TANDY
TRS-80
COMPUTER PRODUCTS

MODEL 2000 MS™-DOS 2.0 CAT. NO. 26-5252

MS™-Assembler

01.00.00

MODEL 2000
Custom manufactured in USA
by Tandy Corporation.

READ ME FIRST

All computer software is subject to change, correction, or improvement as the manufacturer receives customer comments and experiences. Radio Shack has established a system to keep you immediately informed of any reported problems with this software, and the solutions. We have a customer service network including representatives in many Radio Shack Computer Centers, and a large group in Fort Worth, Texas, to help with any specific errors you may find in your use of the programs. We will also furnish information on any improvements or changes that are "cut in" on later production versions.

To take advantage of these services, you must do three things:

(1) Send in the postage-paid software registration card included in this manual immediately. (Postage must be affixed in Canada.)

(2) If you change your address, you must send us a change of address card (enclosed), listing your old address exactly as it is currently on file with us.

(3) As we furnish updates or "patches", and you update your software, you must keep an accurate record of the current version numbers on the logs below. (The version number will be furnished with each update.)

Keep this card in your manual at all times, and refer to the current version numbers when requesting information or help from us. Thank you.

APPLICATIONS SOFTWARE VERSION LOG			OP. SYSTEM VERSION LOG
01.00.00			

Cat. No. 26-5252

MS™-ASSEMBLER

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK AND TANDY
COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK
COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK
FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Radio Shack and Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores and from Radio Shack franchisees and dealers at its authorized location.** The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **Except as provided herein, Radio Shack makes no express warranties, and any implied warranty of merchantability or fitness for a particular purpose is limited in its duration to the duration of the written limited warranties set forth herein.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. **Except as provided herein, Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by "Equipment" or "Software" sold, leased, licensed or furnished by Radio Shack, including, but not limited to, any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of the "Equipment" or "Software". In no event shall Radio Shack be liable for loss of profits, or any indirect, special, or consequential damages arising out of any breach of this warranty or in any manner arising out of or connected with the sale, lease, license, use or anticipated use of the "Equipment" or "Software".**

Notwithstanding the above limitations and warranties, Radio Shack's liability hereunder for damages incurred by customer or others shall not exceed the amount paid by customer for the particular "Equipment" or "Software" involved.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

8086/8088 and 8087 Instructions and Support

Presently MS-Assembler supports only 8086/8088 instructions and operands. The 8087 instructions and operands will be made available at a future date.

Tandy Corporation
Part No. 8759439

MS™-ASSEMBLER

Reference Manual

Contents

Figures

Tables

Introduction

Introduction

System Requirements

The MS-Assembler and the MS-CREF Utility can be used with the Model 2000 with 256K bytes of random access memory and two floppy disk drives or one floppy disk drive and one hard disk drive.

Package Contents

The MS-Assembler package includes one disk and one reference manual.

Software

The MS-Assembler contains the following files on disk:

File	Contents
MASM	MS-Assembler
CREF	Cross-Reference File Utility

About this Manual

The MS-Assembler Reference Manual describes the operation of the MS-Assembler and the mnemonics of the assembly language. This manual assumes that you have a working knowledge of the MS-Assembler, the assembly language and MS-DOS.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

UPPER-CASE

indicates keywords (material that you must type). You may type the keywords in any combination of upper- and lower-case letters. MS-DOS interprets them as upper-case.

(KEYBOARD CHARACTER)

indicates a key you press.

lower case italics

represent words, letters, characters, or values that you supply.

[ ] (square brackets)

indicates optional parameters.

{ } (braces)

indicates you have a choice between two or more entries, one of which must be chosen unless the entries are also enclosed in square brackets.

. . . (ellipsis)

indicates that you may repeat a parameter as many times as you want.

Type all other punctuation exactly as shown in the syntax line.

Learning More About Assembly Level Language

The manual in this package provides complete reference information for your use of MS-Assembler. It does not, however, teach you how to write programs in Assembly language. If you are new to Assembly language or need help in learning to program, we suggest you read any of the following books:

Morse, Stephen P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.

Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA: Osbourne/McGraw-Hill, 1980.

The ASM86 Language Reference Manual. Santa Clara, CA: Intel Corporation, 1981, 1982.

8086/8087/8088 Macro Assembly Language Reference Manual. Santa Clara, CA: Intel Corporation, 1980.

The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1979.

Note:

Some of the information in these books was based on preliminary data and may not reflect the final functional state of the microprocessors.

Features of the MS™-Assembler

Microsoft's MS-Assembler is a powerful program for 8086-based computers. The MS-Assembler incorporates many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from an 8086, 8087, or 8088 microprocessor. Although the MS-Assembler is more complex than most other microcomputer assemblers, it is easy to use.

The MS-Assembler produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked using Microsoft's MS-LINK utility to produce relocatable, executable object code, which you can load anywhere in memory. Thus, the program can execute where it is most efficient, instead of in some fixed range of memory addresses.

In addition, by using relocatable code you can create programs in modules, each of which can be assembled, tested, and perfected individually. Because the MS-Assembler tests and assembles smaller pieces of program code, recording time is shortened. Also, all modules can be error free before being linked into larger modules or into the whole program.

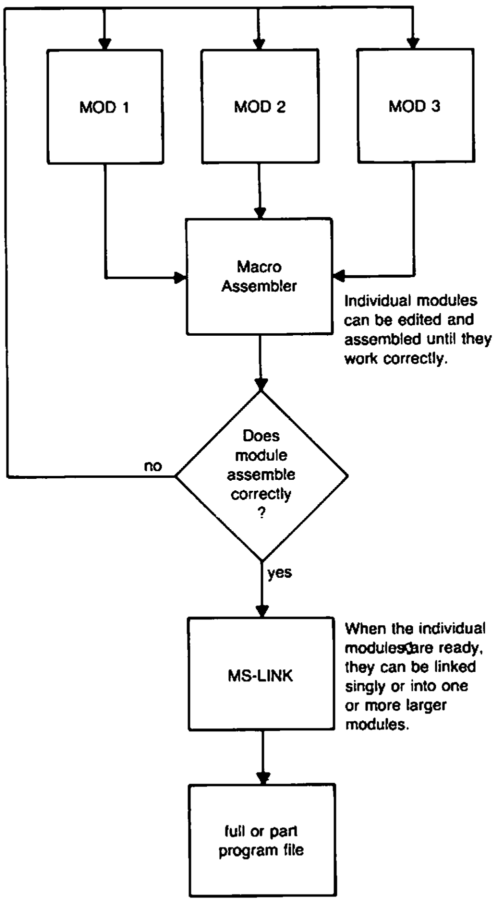

MOD 1
MOD 2
MOD 3
Macro Assembler
Individual modules can be edited and assembled until they work correctly.
Does module assemble correctly ?
no
yes
MS-LINK
When the individual modules are ready, they can be linked singly or into one or more larger modules.
full or part program file

Figure 1-1. The Assembly Process

The MS-Assembler supports Microsoft's complete 8080 macro facility, which is Intel 8080 standard. The macro facility lets you write blocks of code for a set of frequently used instructions. This eliminates the need for recoding these instructions each time they are required in the program.

These blocks of code are called macros. The instructions are the macro definition. Each time you need a set of instructions, you call a macro in the source file. The MS-Assembler expands the macro call by automatically assembling the block of instructions into the program.

The macro call also passes parameters to the MS-Assembler for use during macro expansion. Using macros reduces the size of a source module because the macro definitions are given only once; other occurrences are one-line calls.

You can nest macros; that is, you can call one macro from inside another macro block. The number of macros you can nest is limited only by the size of your computer's memory.

The macro facility includes repeat, indefinite repeat, and indefinite repeat character directives for programming repeat block operations. You can also use the MACRO directive to alter the action of any instruction or directive by using the instruction or directive name as the macro name.

When you place any instruction or directive statement in the program, the MS-Assembler first checks the symbol table it created to see if the instruction or directive is a macro name. If it is, the MS-Assembler "expands" the macro call statement by replacing it with the body of instructions in the macro's definition. If the name is not defined as a macro, the MS-Assembler tries to match the name with an instruction or directive. The MACRO directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.

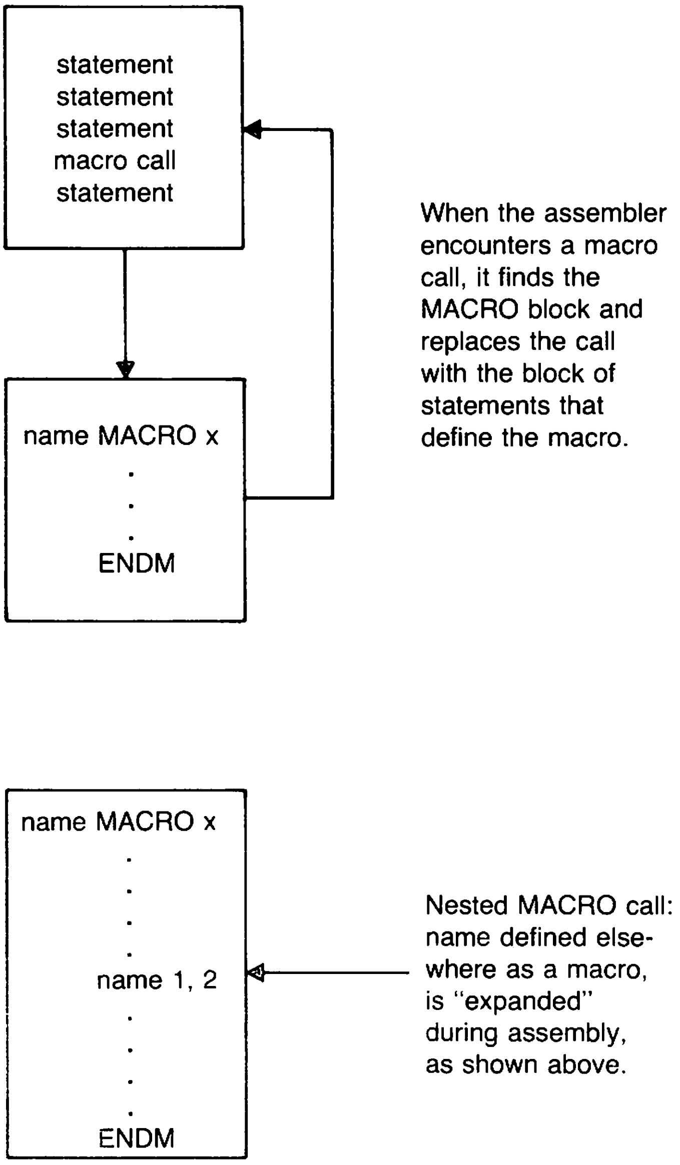
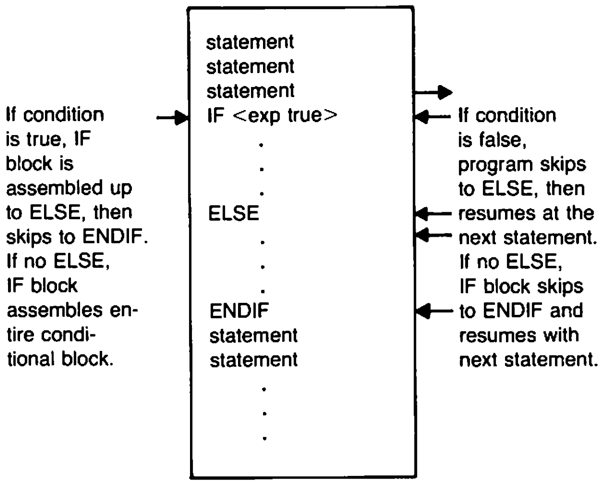

statement
statement
statement
macro call
statement
name MACRO x
.
.
.
ENDM
When the assembler encounters a macro call, it finds the MACRO block and replaces the call with the block of statements that define the macro.
name MACRO x
.
.
.
.
name 1, 2
.
.
.
.
ENDM
Nested MACRO call: name defined else-where as a macro, is "expanded" during assembly, as shown above.

Figure 1-2. Assembler Macros

The MS-Assembler supports an expanded set of conditional directives. Directives for evaluating a variety of assembly conditions can test assembly results and branch where required. Unneeded or unwanted portions of code are left unassembled. The MS-Assembler can test for blank or nonblank arguments, for defined or undefined symbols, for equivalence, and for first assembly pass or second and can compare strings for identity or difference. The conditional directives simplify the evaluation of assembly results and make programming the testing code for conditions easier.

You can also nest conditionals with the MS-Assembler's conditional assembly facility. You can nest a maximum of 255 levels of conditional assembly blocks.


statement
statement
statement
IF <exp true>
ELSE
ENDIF
statement
statement
If condition is true, IF block is assembled up to ELSE, then skips to ENDIF. If no ELSE, IF block assembles entire conditional block.
If condition is false, program skips to ELSE, then resumes at the next statement. If no ELSE, IF block skips to ENDIF and resumes with next statement.

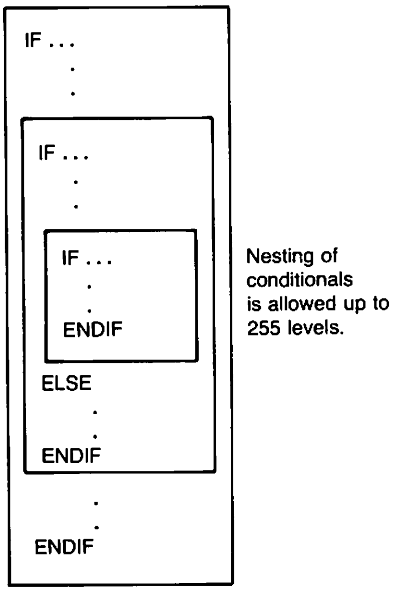
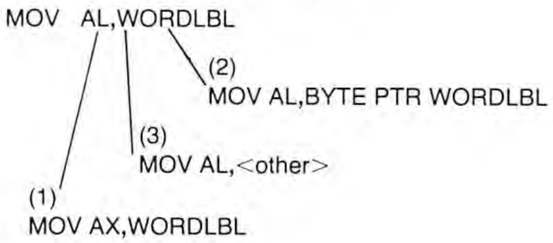


Figure 1-3. Conditional Statements

The MS-Assembler supports all the major 8080 directives found in Microsoft's MS-Assembler for the 8080 processor. Therefore, you can use any conditional, macro, or repeat blocks programmed with the 8080 Macro Assembler with the MS-Assembler for the 8086. You must convert processor instructions and some directives (for example, PHASE, CSEG, DSEG) within the blocks to the 8086 instruction set. All the major MS-Assembler directives (pseudo-ops) for the 8080 that are supported by the MS-Assembler for the 8086 assemble as is, as long as the expressions to the directives are correct for the processor and the program. The syntax of directives is unchanged. The MS-Assembler is upwardly compatible, Macro Assembler for the 8080 processor and with Intel's ASM86(R), except Intel codemacros and macros.

Some 8086 instructions take only one operand type. If you enter a typeless operand for an instruction that accepts only one type of operand (for example, in the instruction PUSH [BX], [BX] has no size, but PUSH only takes a word), the MS-Assembler displays an error message but generates the "correct" code. That is, it always outputs instructions, not just NOP instructions. For example, if you enter:

You may have meant one of three instructions:



The MS-Assembler generates instruction (2), because it assumes that when you specify a register you mean that register and that size; therefore, the other operand is the "wrong size." The MS-Assembler accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some mundane debugging chores. The MS-Assembler still returns an error message, however, because you may have misstated the operand the MS-Assembler assumes is "correct."

Overview of the MS-Assembler Operation

The first task in developing a program is to create a source file. Use EDLIN (the resident editor in Microsoft's MS-DOS operating system) or any other 8086 editor compatible with your operating system to create the source file. The MS-Assembler assumes a default filename extension of .ASM for the source file. Creating the source file involves writing instruction and directive statements that follow the rules and constraints described in Chapters 1-4 in this manual.

When the source file is ready, run the MS-Assembler as described in Chapter 7, "Assembling a Source File." Refer to Appendix H, "MS-Assembler Messages," for explanations of any messages displayed during or immediately after assembly.

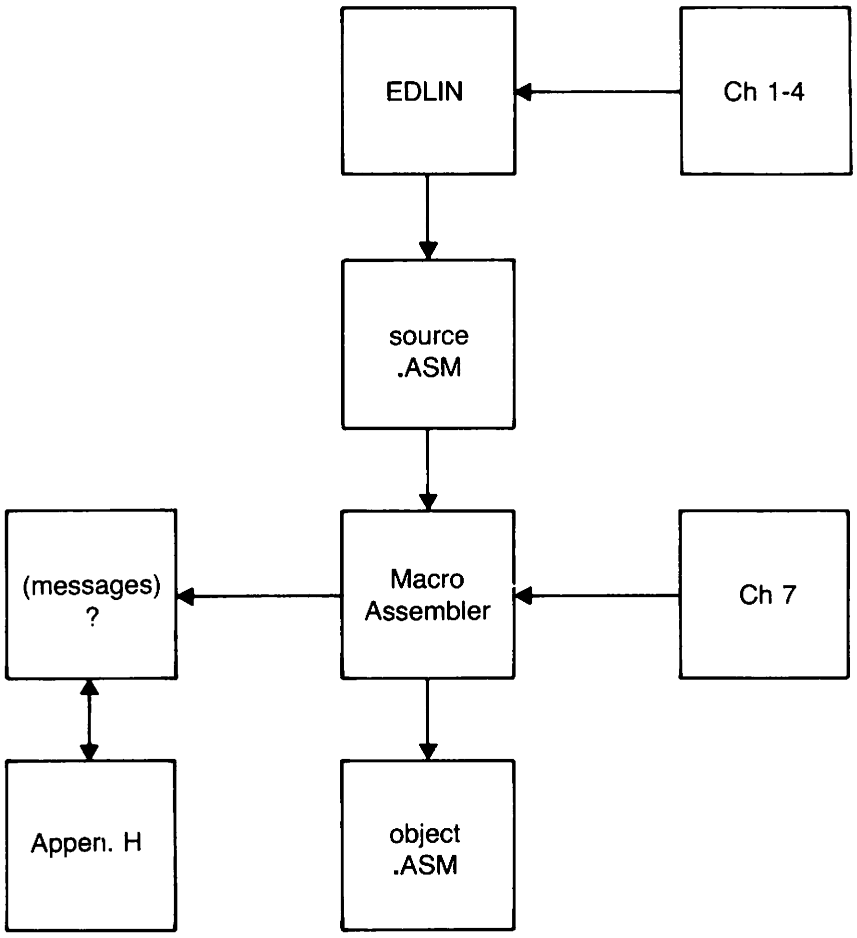


Figure 1.4. Overview of Assembler Operation

The MS-Assembler is a two-pass program. This means that the source file is assembled twice. Slightly different actions occur during each pass. During Pass 1, the MS-Assembler performs the following tasks:

(1) evaluates the statements and expands macro call statements

(2) calculates the amount of code it will generate

(3) builds a symbol table in which it assigns values to all symbols, variables, labels, and macros

During Pass 2, the MS-Assembler performs the following tasks:

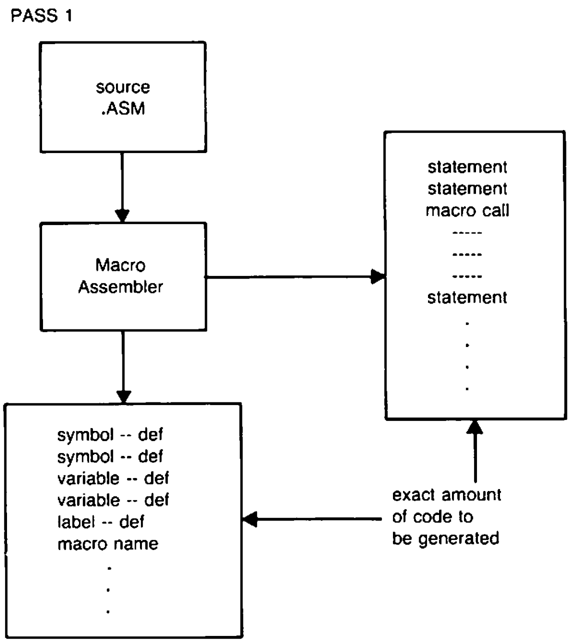
(1) fills in the symbol, variable, label, and expression values from the symbol table

(2) expands macro call statements

(3) sends the relocatable object code into a file with the default filename extension .OBJ

The .OBJ file is suitable for processing with the Microsoft LINK utility (MS-LINK). You can store the .OBJ file as part of your library of object programs and later link it with one or more .OBJ modules by MS-LINK (refer to the MS-LINK utility for further explanation and instructions).

You can also assemble the source file without creating an .OBJ file. The MS-Assembler performs all the tasks listed above but does not send the object code to a disk. Your screen displays only erroneous source statements. This practice is useful for checking the source code for errors. It is faster than creating a .OBJ file because no file is created or written. You can test-assemble modules quickly and correct errors before you put the object code on disk. Modules that assemble without errors do not clutter the disk.



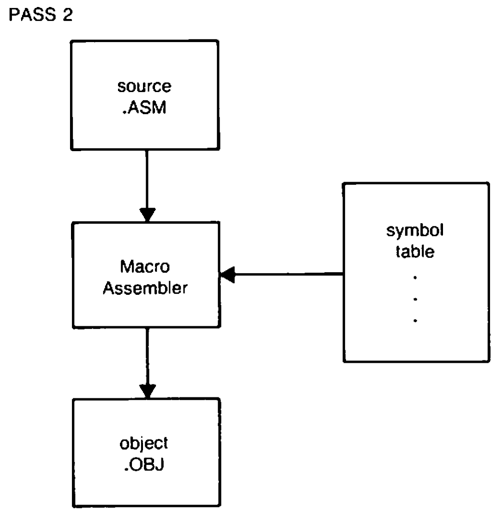


Figure 1-5. Pass 1 and Pass 2

The MS-Assembler creates, on command, a listing file and a cross-reference file. The listing file contains the beginning relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. The listing also contains a symbol table that shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file receives the default filename extension .LST.

The cross-reference file contains a compact representation of variables, labels, and symbols. The cross-reference file receives the default filename extension .CRF. When MS-CREF processes this cross-reference file, the file is converted into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order; followed by the line number in the source program where each is defined; followed by the line numbers where each is used in the program. The final cross-reference listing receives the filename extension .REF.

Figure 1-6 illustrates the files that the MS-Assembler can produce.

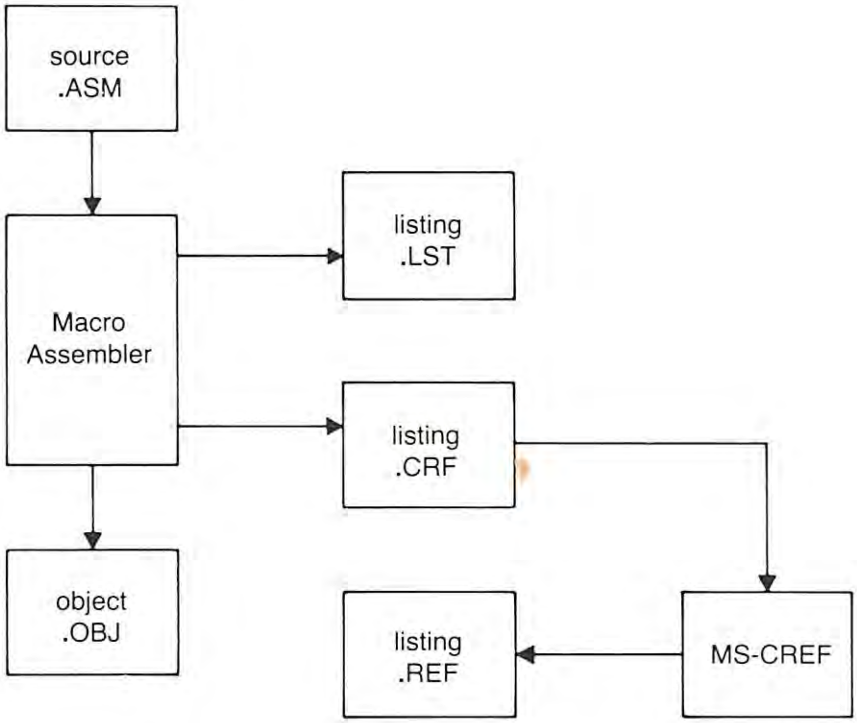


Figure 1-6. Files That the MS-Assembler Produces

Getting Started Session

Getting Started and Sample Session

Preliminary Procedures

This section describes several preliminary procedures, some of which are required and some of which are highly recommended before you begin the sample session or assemble any programs of your own. If you are unfamiliar with any of the MS-DOS procedures mentioned, consult your MS-DOS manual for instructions.

Backing Up Your MS-Assembler Disk

This step is optional but highly recommended.

The first thing you should do when you have unwrapped your MS-Assembler disk is to make copies to work with, saving the original disk for backup. Make the copies using the COPY or DISKCOPY utilities supplied with MS-DOS.

Setting Up Your MS-Assembler Disk

This step is required.

You must have the file COMMAND.COM on the backup of your MS-Assembler disk in order to use your disk in every drive after booting MS-DOS. Therefore, you must copy COMMAND.COM to the backup of your MS-Assembler disk (with the MS-DOS command COPY).

Program Development

This section provides a brief introduction to program development, a multistep process which includes first writing the program, and then assembling, linking, and running it. For a brief explanation of terms that may be unfamiliar, see Section 1.3, "Vocabulary."

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Therefore, before you run a program, some type of translation, from the statements in your program, to the machine language of your microprocessor, must occur. Assemblers are programs that perform this translation.

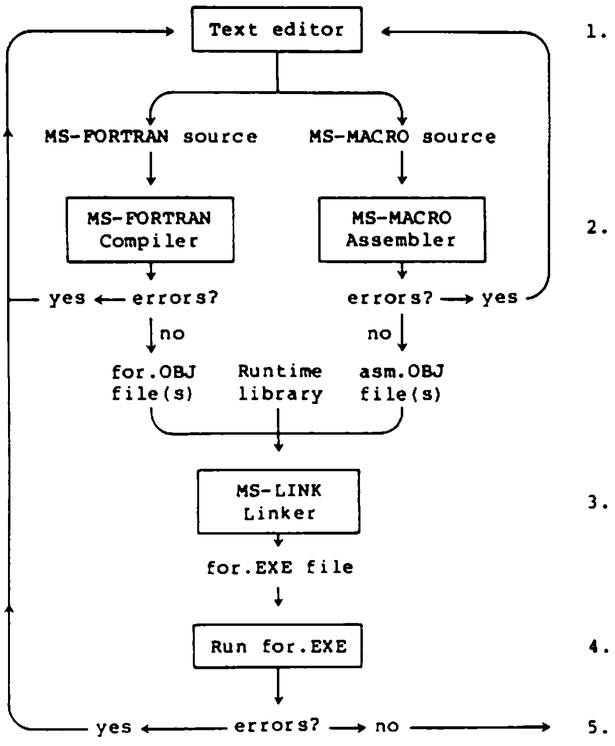
An assembler translates a source program and creates a new file called an object file. The object file contains relocatable machine code that can be placed and run at different absolute locations in memory.

Assembly also associates memory addresses with variables and with the targets of jump statements, so that lists of variables or of labels do not have to be searched during execution of your program.

Before a successfully assembled program can be executed, it must run through MS-LINK. MS-LINK computes the absolute offset addresses for routines and variables in relocatable object modules and then resolves all external references. The linker saves your program on disk as an executable file, ready to run.

You may, at link time, link more than one object module, as well as routines written in a compiler language such as MS-FORTRAN or other high-level languages, and routines in other libraries.

The following illustrates the entire program development process.



Program Development

1. Create and edit the MS-Asssembler source file.

   Program development begins when you write an MS-Assembler program; any general purpose text editor will serve the purpose.

2. Assemble the assembler source.

   Once you have written a program, assemble it with the MS-Assembler.. The assembler flags grammatical errors as it reads your source file. If assembly is successful, the assembler creates a relocatable object file.

   If you have written your own assembly language routines (for example, to increase the speed of execution of a particular algorithm), assemble those routines with the MS-Assembler.

3. Link the assembled OBJ files.

   An assembled object file is not executable and must be run through the MS-LINK utility. Separately compiled subroutines and functions can also be linked to your program at this time.

4. Run the EXE file.

   The linker links all modules needed by your program and produces, as output, an executable run file with .EXE as the extension. This file can be executed by simply typing its filename.

5. Reassemble, relink, and rerun.

   Repeat these processes until your program has successfully assembled, linked, and run without errors.

Vocabulary

This section reviews some of the vocabulary that is commonly used in discussing the steps in program development. The definitions given are intended primarily for use with this manual. Thus, neither the individual definition nor the list of terms is comprehensive.

An MS-Assembler program is more commonly called a "source program" or "source file." The source file is the input file to the assembler. The assembler translates this source and creates, as output, a new file called a "relocatable object file." The source and object files generally have the default extensions .ASM and .OBJ, respectively. After assembling, the object file must be passed through the Linker to produce an executable program or run file. The run file has the extension .EXE.

Some other terms you should know are related to stages in the development and execution of an assembled program. These stages are:

1. Assemble time

   The time during which the assembler is executing and during which it assembles an MS-Assembler source file and creates a relocatable object file.

2. Link time

   The time during which the linker is executing and during which it links together relocatable object files and library files.

3. Runtime

   The time during which an assembled and linked program is executing. By convention, runtime refers to the execution time of your program and not to the execution time of the assembler or the linker.

The following terms pertain to the linking process:

1. Module

   A general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules.

   The object files created by the assembler are said to be "relocatable," that is, they do not contain absolute addresses. Linking produces an "executable" module, that is, one that contains the necessary addresses to proceed with loading and running the program.

2. Routine

   Code, residing in a module, that represents a particular subroutine or function. More than one routine may reside in a module.

3. External reference

   A variable or routine in one given module that is referred to by a routine in another module. The variable or routine is often said to be "defined" in the module in which it resides.

   The linker tries to resolve external references by searching for the declaration of each such reference in other modules. If such a declaration is found, the module in which it resides is selected to be part of the executable module (if it is not already selected) and becomes part of your executable file. These other modules are usually library modules in the runtime library.

   If the variable or routine is found, the address associated with it is substituted for the reference in the first module, which is then said to be "bound." When a variable is not found, it is said to be "undefined" or "unresolved."

4. Relocatable module

   One whose code can be loaded and run at different locations in memory. Relocatable modules contain routines and variables represented as offsets relative to the start of the module. These routines and variables are said to be at "relative" offset addresses.

   When the module is processed by the linker, an address is associated with the start of the module. The linker then computes an absolute offset address that is equal to the associated address plus the relative offset for each routine or variable. These new computed values become the absolute offset addresses that are used in the executable file. Assembled object files and library files are all relocatable modules.

   These offset addresses are still relative to a "segment," which corresponds to an 8086 segment register. Segment addresses are not defined by the linker; rather, they are computed when your program is actually loaded prior to execution.

Sample Session

This manual (including this appendix) is not a tutorial. To learn Assembly Language, see your computer dealer for information on helpful books.

This appendix is for those of you who want to try a session using MS-DOS and MS-Assembler. It demonstrates how to create (write) a program source file (using EDLIN), build a cross reference file (using the MS-CREF utility), and assemble (using MASM), link (using LINKER), and debug (using DEBUG) a program.

This session is for demonstration only. To find out how and why each MS-system works the way it does, you will need to refer to the specific chapters in this manual and the MS-DOS Commands Reference Manual.

In this session you will create two source program files. To do this, use the MS-Editor. However, prior to this it is advisable to copy your MS-Assembler program and CREF Utility to a copy of your MS-DOS diskette. Insert this diskette in Drive A and insert a data dikette in Drive B. In the TRSDOS Ready mode, type:

EDLIN B:(*filespec*) (ENTER)

(filespec = filename, such as "SampfiL1".) For *filespec*, type the filename you want your first file to have.

EDLIN displays:

```
New file
*
```

The asterisk indicates that EDLIN is ready for you to enter a command. To enter the insert mode, so that you can enter text lines into the file, type:

I (ENTER)

EDLIN displays the line number followed by a colon and asterisk.

```
1:*
```

Each line of text that you enter is placed into the text file until you type (F6) (ENTER) or (CTRL) (Z) (ENTER) to end the file.

Insert the following lines into B:SampfiL1:

```
BRANCH_ADDRESSES  SEGMENT
          BRANCH_TABLE_1                       DW ROUTINE_1
                                               DW ROUTINE_2
                                               DW ROUTINE_3
                                               DW ROUTINE_4
                                               DW ROUTINE_5
                                               DW ROUTINE_6
                                               DW ROUTINE_7
                                               DW ROUTINE_8

BRANCH_ADDRESSES  ENDS

PROCEDURE_SELECT  SEGMENT

          ASSUME     CS:PROCEDURE_SELECT,
          ASSUME     DS:BRANCH_ADDRESSES

          MOV        BX,BRANCH_ADDRESSES            ;base-address of
          MOV        DS,BX                          ;segment containing
                                                    ;lists
          LEA        BX,BRANCH_TABLE_1              ;base-address of list
                                                    ;of branch addresses
          MOV        SI,7*TYPE BRANCH_TABLE_1       ;points initially to
                                                    ;last such entry
                                                    ;in list
          MOV        CX,8                           ;loop-counter
                                                    ;allowing 8 shifts
                                                    ;maximum
    L:    SHL        AL,1                           ;shifts high-order
                                                    ;AL bit into CF
          JNC        NOT_YET                        ;if CF = 0, routine
                                                    ;represented by that
                                                    ;bit not desired
          JMP        WORD  PTR [BX][SI]             ;if CF = 1, transfer
                                                    ;to procedure
                                                    ;represented by most
                                                    ;recent bit tested
```

```
NOT_YET: SUB SI, TYPE BRANCH_TABLE_1          ;adjust index
                                              ;register to point
                                              ;to "next"
                                              ;branch-address
          LOOP    L                           ;decrement CX; if
                                              ;CX > 0, transfer to
                                              ;L so as to shift
CONTINUE_MAIN_LINE:                           ;AL and retest
                    .                         ;we reach here only
                    .                         ;if no bit was set
                    .                         ;to indicate a
            ROUTINE_1:                        ;desired routine
                    .
                    .
                    .
            ROUTINE_2:
                    .
                    .
                    .
            ROUTINE_3:
                    .
                    .
                    .
PROCEDURE_SELECT ENDS
```

TO close B:SampfiL1, type:

(CTRL) (Z) (ENTER)

Then to save this file and exit editor type:

(E)

Now you are going to create another new file and enter lines of text to it. Type:

EDLIN B:SampfiL2 (ENTER)

EDLIN displays:

```
New file
*
```

The asterisk indicates that EDLIN is ready for you to enter a command. Enter the insert mode again so that you can enter text lines into B:SampfiL2:

I (ENTER)

Insert the following lines into B:SampfiL2:

```
;The following illustrates the use of interrupt procedures for the 8086. The code sets up six interrupt
;procedures for a hypothetical 8086 system involved in some type of process control application.
;There are 4 sensing devices and two alarm devices, each of which can supply external interrupts to
;the 8086. The different interrupt-handling procedures shown below are arbitrary; that is, the events
;and responses described are not inherent in the 8086 but rather in this hypothetical control
;application. The procedures merely illustrate the diverse possibilities for handling situations of
;varying importance and urgency.

ASSUME CS:INTERRUPT_PROCEDURES, DS:DATA_VAR

DEVICE_1_PORT                   EQU           0F000H
DEVICE_2_PORT                   EQU           0F002H
DEVICE_3_PORT                   EQU           0F004H
DEVICE_4_PORT                   EQU           0F006H
WARNING_LIGHTS                  EQU           0E000H
CONTROL_1                       EQU           0E008H
                                EXTRN CONVERT_VALUE:FAR
                          ;Positioning this EXTRN here indicates
                          ;that CONVERT_VALUE is outside of
                          ;all segments in this module
INTERRUPT_PROC_TABLE SEGMENT BYTE AT 0
                  ORG 08H
                  DD ALARM_1                  ;nonmaskable interrupt
                                              ;type 2

;One 64K area of memory contains pointers to the routines that handle interrupts. This area begins at
;absolute address zero. The address for the routine appropriate to each interrupt type is expected as
;the contents of the double word whose address is 4 times that type. Thus the address for the
;handler of nonmaskable-interrupt type 2 is stored as the contents of absolute location 8. These
;addresses are also called interrupt vectors since they point to the respective procedures.

;The first 32 interrupt types (0-31) are defined or reserved by INTEL,for present and future uses.
;(See the 8086 User's Manual for more detail.) User-interrupt type 32 must therefore use location 128
;(=80h) for its interrupt vector.

                  ORG 80H

                  DD     ALARM_2            ;INTERRUPT TYPE 32
                  DD     DEVICE_1           ;INTERRUPT TYPE 33
                  DD     DEVICE_2           ;INTERRUPT TYPE 34
                  DD     DEVICE_3           ;INTERRUPT TYPE 35
                  DD     DEVICE_4           ;INTERRUPT TYPE 36

INTERRUPT_PROC_TABLE ENDS

DATA_VAR SEGMENT PUBLIC

EXTRN          INPUT_1_VAL:BYTE, OUTPUT_2_VAL:BYTE
EXTRN          INPUT_3_VAL:BYTE, INPUT_4_VAL:BYTE
EXTRN          ALARM_FLAG:BYTE, INPUT_FLAG:BYTE
```

```
;The names above are used by 1 or more of the procedures below, but the location or value referred
;to is located (defined) in a different module. These EXTeRNal references are resolved when the
;modules are linked together, meaning all addresses will then be known. Declaring these EXTRNs
;here indicates what segment they are in.

DATA_VAR ENDS

;The names below are defined later in this module. The PUBLIC directive makes their addresses
;available for other modules to use.

PUBLIC      ALARM_1, ALARM_2, DEVICE_1, DEVICE_2, DEVICE_3
PUBLIC      DEVICE_4

INTERRUPT_PROCEDURES SEGMENT

ALARM_1     PROC        FAR

;The routine for type 2, "ALARM_1", is the most drastic because this interrupt is intended to signal
;disastrous conditions such as power failure. It is nonmaskable; that is, it cannot be inhibited by the
;CLear Interrupts (CLI) instruction.

      MOV         DX,         WARNING_LIGHTS
      MOV         AL,         0FFH
      OUT         DX,AL                               ;turn on all lights
      MOV         DX,         CONTROL_1               ;
      MOV         AL,         38H                     ;turn off
      OUT         DX,AL                               ;machine
      HLT                                             ;stops all processing

ALARM_1     ENDP

ALARM_2     PROC        FAR

            PUSH  DX
            PUSH  AX
            MOV   DX,         WARNING_LIGHTS
            MOV   AL,         1                       ;turn on warning light #1
            OUT   DX,AL                               ;to warn operator of
                                                      ;device
            MOV   ALARM_FLAG, 0FFH                    ;set alarm flag to inhibit
            POP   AX                                  ;later processes which may
                                                      ;now be dangerous
            POP   DX
            IRET                                      ;return from interrupt:
                                                      ;this restores the flags
                                                      ;and returns control to
                                                      ;the interrupted
                                                      ;instruction stream

ALARM_2     ENDP
```

```
DEVICE_1    PROC

           PUSH    DX
           PUSH    AX
           MOV     DX, DEVICE_1_PORT
           IN      AL, DX                        ;get input byte from
           MOV     INPUT_1_VAL, AL               ;device_store value
           MOV     INPUT_FLAG,2                  ;this may alert another
                                                 ;routine or device that
                                                 ;this interrupt and input
                                                 ;occurred
           POP     AX
           POP     DX
           IRET

DEVICE_1    ENDP

DEVICE_2    PROC

           PUSH    DX                            ;when this interrupt type
           PUXH    AX                            ;occurs, the action
                                                 ;necessary is to notify
                                                 ;device_2_port of the
                                                 ;event
           MOV     AL,  OUTPUT_2_VAL             ;get value, to output
           MOV     DX,  DEVICE_2_PORT            ;to device_2_port
           OUT     DX,AL
           POP     AX
           POP     DX
           IRET

DEVICE_2    ENDP

DEVICE_3    PROC
           PUSH    DX                            ;when a device_3 interrupt
           PUSH    AX                            ;occurs, only the lower
                                                 ;byte at the port is of
           MOV     DX, DEVICE_3_PORT             ;value
           IN      AL,DX
           AND     AL,0FH                        ;mask off top four bits
           MOV     INPUT_3_VAL, AL               ;store value for use
           POP     AX                            ;by later routines
                                                 ;in another module
           POP     DX
           IRET

DEVICE_3    ENDP
```

```
DEVICE_4    PROC

            PUSH    DX
            PUSH    CX                        ;a device_ interrupt
            PUSH    AX                        ;provides a value which
            MOV     DX, DEVICE_4_PORT         ;needs immediate
                                              ;conversion by another
            IN      AL,DX                     ;procedure before this
            MOV     CL, AL                    ;interrupt handler can
                                              ;allow it to be used at
                                              ;input_4_val

            CALL    CONVERT_VALUE             ;converts input value in
            MOV     INPUT_4_VAL, AL           ;CL to new result in AL
                                              ;and saves that result in
                                              ;input_4_val

            POP     AX
            POP     CX
            POP     DX
            IRET

DEVICE_4    ENDP

INTERRUPT_PROCEDURES    ENDS

                        END
```

When you have finished creating both source files, exit EDLIN by typing:

CTRL Z ENTER

Then to save this file type:

E

Then type:

MASM ENTER

The assembler is loaded from the diskette, and the first prompt is displayed:

```
Source filename [.ASM]
```

Answer the prompt requesting the source filename with:

B:SampfiL1 ENTER

If you do not specify .ASM, it will be assumed by the assembler.

The assembler then requests the object filename and displays the default value it will use if you do not enter a filename:

```
Object filename [DDD.OBJ]
```

Type:

B:SampfiL1 (ENTER)

The assembler then requests the filename of the listing.

Source Listing [NUL .LST]

If you do not enter a filename for the source, no listing is generated. Since you want to generating a listing, type:

B:SampfiL1 (ENTER)

The assembler then requests the cross-reference filename. This is the cross-reference file which the CREF utility converts into an **alphabetical listing** of the symbols of the file.

Cross reference [NUL .CRF]

If you do not enter a cross-reference filename, no cross-reference file is generated. Since you want to generate a cross reference file, type:

B:SampfiL1 (ENTER)

Note: You can type all of the above responses on the same line as "MASM," if you wish. Type MASM followed by one blank space and then type the responses (the responses must be separated by commas).

Assemble B:SampfiL2 in the same manner as you have assembled SampfiL1 above.

Then type:

CREF (ENTER)

The Cross-Reference Utility is loaded and displays the first prompt, which is a request for the cross-reference filename:

Cross reference [.CRF]

The assembler diskette is no longer needed. Therefore, remove it and replace it with the data diskette containing the files to be converted.

Type:

B:SampfiL1 (ENTER)

The second prompt, a request for the cross-reference listing filename, is displayed.

Listing [crffile.REF]

Type:

B:SampfiL1 (ENTER)

The Cross-Reference Utility proceeds to convert the information in the B:SampfiL1.CRF to an alphabetical reference listing in the file B:SampfiL1.REF.

Note: You can type all the above responses on the same line as "CREF," if you wish. Type CREF followed by one blank space and then type the responses (the responses must be separated by commas).

Convert B:SampfiL2.CRF in the same manner as you have converted B:SampfiL1.CRF above.

When you have finished and assembled the various modules for a particular application, you can link them to form a single composite run time program. The Linker is provided on the MS-DOS diskette for this purpose.

LINK (ENTER)

When the linker is loaded, the first prompt requesting the object files is displayed.

Object Modules [.OBJ]:

This is a request for the list of files that are to be linked. Type:

B:SampfiL1.OBJ B:SampfiL2.OBJ (ENTER)

The second prompt, a request for the full pathname (or filename) of the executable run file, is displayed.

Run File [.EXE]:

Type:

B:SampfiLE.EXE (ENTER)

If you do not enter a pathname, the default value assumed is that of the filename or pathname entered for the first prompt. The third prompt, a request for the name of the listing file that is to contain the memory map, is displayed.

List File [NUL.MAP]:

Type:

B:SampfiLE (ENTER)

If you do not enter a pathname, the default value NUL.MAP is assumed and no listing file (containing the memory map) is created. The fourth prompt, a request for library filenames, is displayed.

Libraries [.LIB]:

This prompt lets you direct the linker to search for libraries which have been created by a library utility. When you have obtained a compatible library utility you may want to search for SampfiLE libraries by typing:

SampfiLE.LIB (ENTER)

Since, it is not necessary to enter any library filenames when using the assembler, you may just press (ENTER).

The Linker Utility proceeds to link the object modules B:SampfiL1.OBJ and B:SampfiL2.OBJ into an executable run file B:SampfiLE.EXE. The Linker utility also produces the listing file B:SampfiLE.MAP containing the memory map.

Note: You can type all the above responses on the same line as "LINK." Type LINK followed by one blank space and then type the responses (the responses must be separated by commas).

To run B:SampfiLE.EXE type:

B:SampfiLE.EXE (ENTER) or only B:SampfiLE (ENTER)

MS-DOS loads and executes the application program B:SampfiLE.EXE and, when finished, control is returned to MS-DOS.

If minor errors are noted, you can alter the executable object file in memory using the DEBUG utility. This eliminates the need to reassemble a program to find out if your corrections have fixed the problem.

When MS-DOS has control, type:

DEBUG (ENTER)

You can now work with the present contents of the registers.

In order to load B:SampfiLE.EXE you must first identify the file to the DEBUG Utility using the command N. Type:

N B:SampfiLE.EXE (ENTER)

and then to load the file, type:

L (ENTER)

B:SampfiLE.EXE is loaded and you can now use any of the DEBUG commands to debug your B:SampfiLE.EXE program.

To terminate debugging, type:

Q

For further details about the DEBUG Utility see the MS-DOS Commands Reference Manual.

Chapter 1

Creating a Source File

1.1 General Facts About Source Files

Creating Your Source File

To create a source file you use an editor, such as EDLIN in Microsoft's MS-DOS. You simply create a program file as you would for any other assembly or high-level programming language. Use the general facts and specific descriptions in this chapter and in chapters 2-4 when creating the file.

Naming Your Source File

A source file must have a name, which may be any name that your operating system recognizes and·the .ASM extension. When you assemble your source file, the MS-Assembler assumes that your source filename has the extension .ASM.

Please note that the MS-Assembler gives the object file it outputs the default extension .OBJ. To avoid confusion or the destruction of your source file, do not give a source file an extension of .OBJ. For similar reasons, do not use the extensions .EXE, .LST, .CRF, and .REF.

Legal Characters

The following are legal characters for your symbol names:

A-Z 0-9 ? @ _ $

The first character of a name can be any character except a number (0-9). The first character of a numeric value, however, must be a number.

The following additional special characters act as operators or delimiters:

:	(colon)—segment override operator
.	(period)—operator for field name of Record or Structure; may be used in a file name only if it is the first character
[ ]	(square brackets)—around register names to indicate value in address in register, not value (data) in register
( )	(parentheses)—operator in DUP expressions and operator to change precedence of operator evaluation
< >	(angle brackets)—operators used to enclose initialization values for Records or Structure, to enclose parameters in IRP macro blocks, and to indicate literals

This manual also uses square brackets and angle brackets for syntax notation in the discussions of the assembler directives (see Section 4.2, "Directives"). When these characters are operators and not syntax notation, we tell you explicitly.

Numeric Notation

The default input radix (number base) for all numeric values is decimal. The output radix for all listings is hexadecimal for code and data items and decimal for line numbers. You can only change the output radix to octal radix by giving the /O switch when the MS-Assembler is run (see Section 7.4, "MS-Assembler Command Switches"). You can change the input radix in two ways:

1. With the .RADIX directive (see Section 4.2.1, "Memory Directives")
2. By special notation appended to a numeric value:

Radix	Range	Notation	Example
Binary	0-1	B	01110100B
Octal	0-7	Q or O	735Q or 621O
Decimal	0-9	none or D	9384 (default) 8149D*
Hexadecimal	0-9 A-F	H	OFFH or 80H**

When using .RADIX 16, remember that numbers ending in B or D will try to use binary or decimal representations. Therefore any number ending in a hexadecimal digit B or D must still have an H suffix for base 16.

* When .RADIX directive changes default radix to not decimal.
**First character must be a number in the range 0-9.

What's in a Source File?

A source file for the MS-Assembler consists of instruction statements and directive statements. Instruction statements consist of 8086 instruction mnemonics and their operands, which command specific processes directly to the 8086 processor. Directive statements are commands to the MS-Assembler to prepare data for use in and by instructions.

Section 1.2 describes statement line format, and Sections 1.3-1.6 and Chapters 2-4 describe the parts of a statement. Statements are usually placed in blocks of code assigned to a specific segment (code, data, stack, extra). The segments may appear in any order in the source file. Within the segments, generally speaking, statements may appear in any order that creates a valid program. Some exceptions to random ordering do exist, and they are discussed under the affected assembler directives.

You must end every segment with an end segment statement (ENDS), every procedure with an end procedure statement (ENDP), and every structure with an end structure statement (ENDS). Likewise, you must end the source file with an END statement that tells the MS-Assembler where to begin executing the program.

Section 3.1, "Memory Organization," describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another and to your programming as a whole. This information is important and helpful for developing your programs. The information is presented in Chapter 3 as a prelude to the discussion of operands and operators.

1.2 Statement Line Format

Statements in source files follow a strict format, which allows some variation.

Directive statements consist of four "fields": Name, Action, Expression, Comment. For example:

```
FOO      DB        0D5E          ;create variable FOO
                                 ;containing the value
                                   0D5EH

Name     Action    Expression    ;Comment
```

Instruction statements usually consist of three "fields": Action, Expression, Comment. For example:

```
         MOV       CX,FOO        ;here's the count number

         Action    Expression    ;Comment
```

An instruction statement may have a Name field under certain circumstances (see Section 1.3, "Names").

1.3 Names

The name field, when present, is the first entry on the statement line. You may begin a name in any column, although normally names are started in Column 1.

You may make names any length. However, the MS-Assembler recognizes only the first 31 characters when assembling your source file.

You also use names with the MACRO directive. All the rules for names in statement lines also apply to MACRO names.

You use names in a statement line to represent code, to represent data, or to represent constants.

To make a name represent code, use:

> <NAME>: followed by a directive, instruction, or nothing at all
>
> <NAMES> LABEL NEAR (for use inside its own segment only)
>
> <NAME> LABEL FAR (for use outside its own segment)
>
> EXTRN <NAME>:NEAR (for use outside its own module but inside its own segment only)
>
> EXTRN <NAME>:FAR (for use outside its own module and segment)

To make a name represent data, use:

> <NAME> LABEL <*size*> (BYTE, WORD, etc.)
>
> <NAME> Dx <*exp*>
>
> EXTRN <NAME>:<*size*> (BYTE, WORD, etc.)

To make a name represent a constant, use:

> <NAME> EQU <*constant*>
>
> <NAME> = <*constant*>
>
> <NAME> SEGMENT <*attributes*>
>
> <NAME> GROUP <*segment-names*>

1.4 Comments

The successful operation of an assembly language program does not depend on comments, but we strongly recommend that you use them.

You must precede every comment on every line with a semicolon. If you want to place a very long comment in your program, you can use the COMMENT directive, which releases you from the required semicolon (see COMMENT in Section 4.2.1, "Memory Directives").

Comments document the processing at particular points in a program and are useful for debugging, for altering code, and for updating code. We recommend that you place comments at the beginning of each segment, procedure, structure, and module and after each line in the code that begins a step in the processing.

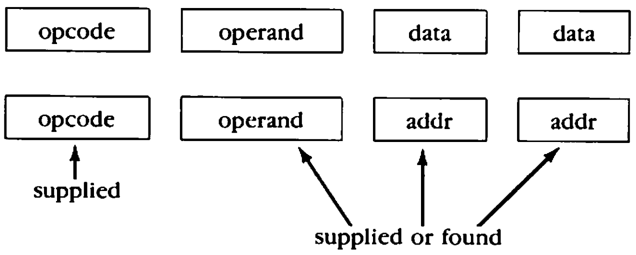
The MS-Assembler ignores comments. Comments do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code.

1.5 Action

The action field contains either an 8086 instruction mnemonic or an MS-Assembler directive. Refer to Section 4.1, "Instructions," for a general discussion and to Appendix D for a list of 8086 instruction mnemonics. The Macro Assembler directives are described in detail in Section 4.2, "Directives."

If the name field is blank, the action field is the first entry in the statement line. In this case, the action may appear in any column, as long as column space remains for the action and expressions fields.

The entry in the action field directs either the processor or the assembler to perform a specific function. Instructions tell the processor to perform some action. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements

(opcode is the action part of an instruction)

Directives give the MS-Assembler directions for I/O, memory organization, conditional assembly, listing and cross-reference control, and definitions.

1.6 Expressions

The expression field contains entries that are operands and/or combinations of operands and operators.

Some instructions take no operands; some take one, and others take two. For two-operand instructions, the expression field consists of a destination operand and a source operand, in that order, separated by a comma. For example:

opcode	dest-operand	source-operand

For one-operand instructions, the operand is a source or a destination operand, depending on the instruction. If you omit one or both of the operands, the instruction carries that information in its internal coding.

Source operands are immediate operands, register operands, memory operands, or attribute operands. Destination operands are register operands and memory operands.

For directives, the expression field usually consists of a single operand. For example:

directive	operand

A directive operand is a data operand, a code (addressing) operand, or a constant, depending on the nature of the directive.

For many instructions and directives, you may connect operands with operators to form a longer operand that looks like a mathematical expression. These operands are called complex operands. Using a complex lets you specify addresses or data derived from several places. For example:

```
MOV     FOO[BX],AL
```

The destination operand is the result of adding the address represented by the variable FOO and the address found in register BX. The processor is instructed to move the value in register AL to the destination calculated from these two operand elements. Another example:

```
MOV     AX,FOO+5[BX]
```

In this case, the source operand is the result of adding the value represented by the symbol FOO plus 5 plus the value found in the BX register.

The MS-Assembler supports the following operands and operators in the expression field (shown in order of precedence):

Operands

- Immediate (incl. symbols)
- Register
- Memory
  - label
  - variables
    - simple
    - indexed
    - structures
- Attribute
  - override
    - PTR
    - :(seg)
    - SHORT
    - HIGH
    - LOW
  - value returning
    - OFFSET
    - SEG
    - THIS
    - TYPE
    - .TYPE
    - LENGTH
    - SIZE
  - record specifying
    - FIELD
    - MASK
    - WIDTH

Operators

- LENGTH, SIZE, WIDTH, MASK, FIELD [ ], ( ), < >
- segment override(:)
- PTR, OFFSET, SEG, TYPE, THIS
- HIGH, LOW
- *, /, MOD, SHL, SHR
- +, −(unary), −(binary)
- EQ, NE, LT, LE, GT, GE
- NOT
- AND
- OR, XOR
- SHORT, .TYPE

NOTE

Some operators can be used as operands or as part of an operand expression. Refer to Sections 3.2, "Operands," and 3.3, "Operators," for details of operands and operators.

Names

Chapter 2

Names: Labels, Variables, and Symbols

Names: Labels, Variables, and Symbols

The MS-Assembler defines and uses names in a number of ways. This chapter discusses the basic methods of defining and using names in statement lines, that is, how to define and use labels, variables, and symbols. Chapters 3-4 present additional uses, and you will discover even more uses as you work with the MS-Assembler.

Names are symbolic representations of values. The values may be addresses, data, or constants.

Names may be any length you choose. However, the MS-Assembler recognizes only the first 31 characters when assembling your source file.

2.1 Labels

Labels are names used as targets for JMP, CALL, and LOOP instructions. The MS-Assembler assigns an address to each label as it is defined. When you use a label as an operand for JMP, CALL, or LOOP, the MS-Assembler can substitute the attributes of the label for the label name, sending processing to the appropriate place.

Labels are defined in four ways:

1. *<name>*:

   Type *name* enclosed in angle brackets and a colon. This defines the name as a NEAR label. You may prefix *<name>*: to any instruction and to all directives that allow a Name field. You may also place *<name>*: on a line by itself.

   Examples:

   ```
   CLEAR_SCREEN:    MOV    AL,20H
   FOO:    DB    0FH
   SUBROUTINE3:
   ```

2.
   ```
   <name>    LABEL    NEAR
   <name>    LABEL    FAR
   ```

   Use the LABEL directive. For further information, see Section 4.2.1, "Memory Directives."

See Type below for a discussion of NEAR and FAR.

Examples:

```
FOO    LABEL    NEAR
GOO    LABEL    FAR
```

3. <*name*>   PROC   NEAR
   <*name*>   PROC   FAR

   Use the PROC directive. For further information, see Section 4.2.1, "Memory Directives."

   NEAR is optional because it is the default if you enter only <*name*> PROC. See Type below for a discussion of NEAR and FAR.

   Examples:

```
REPEAT    PROC    NEAR
CHECKING    PROC
FIND_CHR    PROC    FAR
```

4. EXTRN <*name*>:NEAR
   EXTRN <*name*>:FAR

   Use the EXTRN directive. For further information, see Section 4.2.1, "Memory Directives." See Type below for a discussion of NEAR and FAR.

   Examples:

```
EXTRN FOO:NEAR
EXTRN ZOO:FAR
```

A label has four attributes: segment, offset, type, and the CS ASSUME in effect when the label is defined. Segment is the segment where the label is defined. Offset is the distance from the beginning of the segment to the label's location. Type is either NEAR or FAR.

Segment

Labels are defined inside segments. You must assign a segment to the CS segment register for it to be addressable. You may assign the segment to a group, in which case the group must be addressable through CS. The MS-Assembler requires that a label be addressable through the CS register. Therefore, the segment (or group) attribute of a symbol is the base address of the segment (or group) where it is defined.

Offset

The offset attribute is the number of bytes from the beginning of the label's segment to where the label is defined. The offset is a 16-bit unsigned number.

Type

The two types of labels are NEAR and FAR. Use NEAR labels references from within the segment where the label is defined. NEAR labels may be referenced from more than one module, as long as the references are from a segment with the same name and attributes and have the same CS ASSUME.

Use FAR labels for references from segments with a different CS ASSUME or when there are more than 64K bytes between the label reference and the label definition.

The MS-Assembler generates slightly different code for NEAR and for FAR. NEAR labels supply their offset attribute only (a 2-byte pointer). FAR labels supply both their segment and offset attributes (a 4-byte pointer).

2.2 Variables

Variables are names used in expressions as operands to instructions and directives. A variable represents an address where a specified value may be found.

Variables look much like labels and are defined similarly in some ways; however, the differences are important.

Variables are defined three ways:

1. <*name*> <*define-dir*> ;no colon!
   <*name*> <*struc-name*> <*expression*>
   <*name*> <*rec-name*> <*expression*>

   <*define-dir*> is any of the five Define directives: DB, DW, DD, DQ, DT

   Example:

   ```
   START_MOVE    DW    ?
   ```

   <*struc-name*> is a structure name defined by the STRUC directive.

   <*rec-name*> is a record name defined by the RECORD directive.

   Examples:

   ```
   CORRAL     STRUC
                .
                .
                .
              ENDS
   HORSE      CORRAL     <'SADDLE'>
   ```

   Note: HORSE is the same size as the structure CORRAL.

```
GARAGE      RECORD      CAR:8='P'
SMALL       GARAGE      10 DUP(<'Z'>)
```

Note: SMALL is the same size as the record GARAGE.

See the DEFINE, STRUC, and RECORD directives in Section 4.2.1, "Memory Directives."

2. <*name*> LABEL <*size*>

   Use the LABEL directive with one of the size specifiers. You may specify size in the following ways:

BYTE	—specifies 1 byte
WORD	—specifies 2 bytes
DWORD	—specifies 4 bytes
QWORD	—specifies 8 bytes
TBYTE	—specifies 10 bytes

   Example:

   ```
   CURSOR   LABEL   WORD
   ```

   For further information, see Section 4.2.1, "Memory Directives."

3. EXTRN <*name*>:<*size*>

   Use the EXTRN directive with a size specifier. For further information, see Section 4.2.1, "Memory Directives."

   Example:

   ```
   EXTRN FOO:DWORD
   ```

Variables also have three attributes — segment, offset, and type — as do labels. Segment and Offset are the same for variables as for labels. The Type attribute is different.

Type

The type attribute is the size of the variable's location, as specified when the variable is defined. The size depends on which Define directive or which size specifier was used to define the variable.

Directive	*Type*	*Size*
DB	BYTE	1 byte
DW	WORD	2 bytes
DD	WORD	4 bytes
DQ	QWORD	8 bytes
DT	TBYTE	10 bytes

2.3 Symbols

Symbols are names defined without reference to a Define directive or to code. Like variables, symbols are also used in expressions as operands to instructions and directives.

Symbols are defined three ways:

1. <*name*> EQU <*expression*>

   Use the EQU directive. For further information, see Section 4.2.1, "Memory Directives."

   <*expression*> may be another symbol, an instruction mnemonic, a valid expression, or any other entry (such as text or indexed references).

   Examples:

   ```
   FOO    EQU    7H
   ZOO    EQU    FOO
   ```

2. <*name*> = <*expression*>

   Use the equal sign directive. For further information, see Section 4.2.1, "Memory Directives."

   <*expression*> may be any valid expression.

   Examples:

   ```
   GOO    =    0FH
   GOO    =    $+2
   GOO    =    GOO+FOO
   ```

3. EXTRN <*name*>:ABS

   Use the EXTRN directive with type ABS. For further information, see Section 4.2.1, "Memory Directives."

   Example:

   ```
   EXTRN  BAZ:ABS
   ```

   You must define BAZ by an EQU or = directive to a valid expression.

Chapter 3

Expressions: Operands and Operators

Expressions: Operands and Operators

Expression is'the term used to indicate values on which an instruction or directive performs its functions.

Every expression consists of at least one operand (a value). An expression may consist of two or more operands. Multiple operands are joined by operators. The result is a series of elements that looks like a mathematical expression.

This chapter describes the types of operands and operators that the MS-Assembler supports.

3.1 Memory Organization

Most of your assembly language program is written in segments. In the source file, a segment is a block of code that begins with a SEGMENT directive statement and ends with an ENDS directive. In an assembled and linked file, a segment is any block of code that is addressed through the same segment register and is not more than 64K bytes long.

The MS-Assembler leaves everything relating to segments to MS-LINK. MS-LINK resolves all references. For that reason, the MS-Assembler does not check (because it cannot) to see if your references are entered with the correct distance type. Values such as OFFSET are also left to MS-LINK to resolve.

Although a segment may not be more than 64K bytes long, you may, as long as you observe the 64K limit, divide a segment among two or more modules. (The SEGMENT statement in each module must be the same.)

When the modules are linked, the several segments become one. References to labels, variables, and symbols within each module acquire the offset from the beginning of the whole segment, not just from the beginning of their portion of the whole segment. (All divisions are removed.)

You may gather several segments into a group using the GROUP directive. When you group segments, you tell the MS-Assembler that you want to be able to refer to all these segments as a single entity. (This does not eliminate segment identity, nor does it make values within a particular segment less immediately accessible. It does make value relative to a group base.) The advantage of grouping is that you can refer to data items without worrying about segment overrides or changing segment registers.

References within segments or groups are relative to a segment register. Thus, until linking is complete, the final offset of a reference is relocatable. For this reason, the OFFSET operator does not return a constant. The major purpose of OFFSET is to cause the MS-Assembler to generate an immediate instruction, that is, to use the address of the value instead of the value itself.

A program contains two kinds of references:

1. Code references (JMP, CALL, LOOPxx). These references are relative to the address in the CS register. (You cannot override this assignment.)
2. Data references (all other references). These references are usually relative to the DS register, but you can override this assignment.

When you give a forward reference in a program statement, for example:

MOV AX,<*ref*>

the MS-Assembler first looks for the segment of the reference. It scans the segment registers for the SEGMENT of the reference, then the GROUP (if any) of the reference.

However, the use of the OFFSET operator always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must override this restriction by using the GROUP name and the colon operator. For example:

MOV AX,OFFSET <*group-name*>:<*ref*>

If you set a segment register to a group with the ASSUME directive, then you may also override the restriction on OFFSET by using the register name. For example:

```
        MOV AX,OFFSET DS:<ref>
```

The result of both these statements is the same.

Code labels have four attributes:

1. Segment — to what segment the label belongs
2. Offset — the number of bytes from the beginning of its segment
3. Type — NEAR or FAR
4. CS ASSUME — the CS ASSUME under which the label was coded

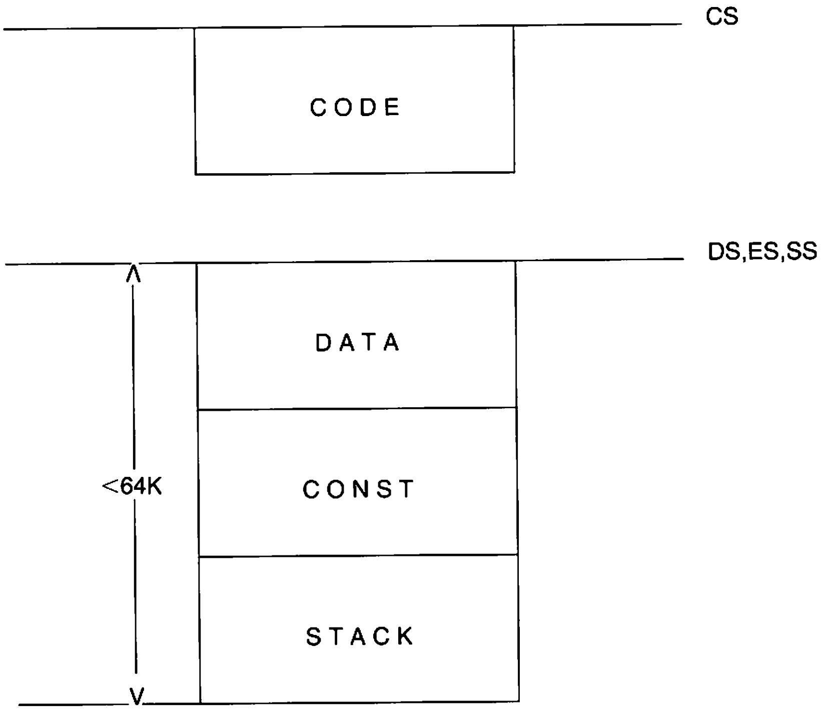
When you enter a NEAR JMP or NEAR CALL, you change the offset (IP) in CS. The MS-Assembler compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they are different, the MS-Assembler returns an error (you must use a FAR JMP or FAR CALL).

When you enter a FAR JMP or FAR CALL, you change both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

For example, a segment is called CODE, and a group (called DGROUP) contains three segments (called DATA, CONST, and STACK). The program statements are:

```
DGROUP    GROUP     DATA,CONST,STACK
          ASSUME    CS:CODE,DS:DGROUP,SS:DGROUP,
                      ES:DGROUP
          MOV       AX,DGROUP
          MOV       DS,AX

;CS initialized by entry; you in-
;itialize DS, do this as soon as
;possible, especially before any
;DS relative references

               .
               .
               .
               .
```

As a diagram, this arrangement could be represented as follows:



Given this arrangement, a statement such as

MOV AX,<*variable*>

causes the MS-Assembler to find the best segment register to reach this variable. (The "best" register is the one that requires no segment overrides.)

A statement such as

MOV AX,OFFSET <*variable*>

tells the MS-Assembler to return the offset of the variable relative to the beginning of the variable's segment.

If this <variable> is in the CONST segment and you want to reference its offset from the beginning of DGROUP, you need a statement such as the following:

MOV AX,OFFSET DGROUP:<*variable*>

The MS-Assembler is a two-pass assembler. During Pass 1, it builds a symbol table and calculates how much code is generated, but does not produce object code. If it finds undefined items (including forward references), it makes assumptions about the reference so that the correct number of bytes is generated. Your screen displays error messages only for those errors involving items that must be defined on Pass 1. No listing is produced unless you include a /D switch when you run the MS-Assembler. The /D switch produces a listing for both passes.

On Pass 2, the MS-Assembler uses the values defined in Pass 1 to generate the object code. Definitions of references during Pass 2 are checked against the Pass 1 value, which is in the symbol table. Also, the amount of code generated during Pass 1 must match the amount generated during Pass 2. If either is different, the MS-Assembler returns a phase error.

Because Pass 1 must keep correct track of the relative offset, some references must be known on Pass 1. If they are not known, the relative offset will not be correct.

The following references must be known on Pass 1:

1. IF/IFE <*expression*>
   If <*expression*> is not known on Pass 1, the MS-Assembler does not know to assemble the conditional block (or which part to assemble if ELSE is used). On Pass 2, the assembler would know and would assemble, resulting in a phase error.
2. <*expression*> DUP(. . .)
   This operand explicitly changes the relative offset; so <*expression*> must be known on Pass 1. The value in parentheses need not be known because it does not affect the number of bytes generated.
3. .RADIX <*expression*>
   Because this directive changes the input radix, constants could have a different value, which could cause the MS-Assembler to evaluate IF or DUP statements incorrectly.

The biggest problem for the MS-Assembler is handling forward references. How can it know the kind of a reference when it still has not seen the definition? This is one of the main reasons for two passes. And, unless the MS-Assembler can tell from the statement containing the forward reference what the size, the distance, or any other of its attributes are, the assembler can only take the safe route (generate the largest possible instruction in some cases, except for segment override or FAR). This results in extra code that does nothing. The MS-Assembler figures this out by Pass 2, but it cannot reduce the size of the instructions without causing an error, so it puts out NOP instructions (90H).

For this reason, the MS-Assembler includes a number of operators that tell the MS-Assembler what size instruction to generate when faced with an ambiguous choice. As a benefit, you can also reduce the size of your program by using these operators to change the nature of the arguments to the instructions.

Examples:

```
MOV AX,FOO ;FOO = forward constant
```

This statement causes the MS-Assembler to generate a move from memory instruction on Pass 1. By using the OFFSET operator, you can cause the MS-Assembler to generate an immediate operand instruction.

```
MOV AX,OFFSET FOO             ;OFFSET says use the
                              ;address of FOO
```

Because OFFSET tells the MS-Assembler to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code.

Similarly, if you have a CALL statement that calls to a label that may be in a different CS ASSUME, you can prevent problems by attaching the PTR operator to the label:

```
CALL FAR PTR <forward-label>
```

At the opposite extreme, you may have a JMP forward that is fewer than 127 bytes. You can save yourself a byte if you use the SHORT operator.

```
JMP SHORT <forward-label>
```

Be sure, however, that the target is within 127 bytes or the MS-Assembler will not find it.

You can use the PTR operator another way to save a byte when using forward references. If you defined FOO as a forward constant, you might enter the statement:

```
MOV [BX],FOO
```

You may want to refer to FOO as a byte immediate. In this case, you could enter either of these statements (they are equivalent):

```
MOV BYTE PTR [BX],FOO

MOV [BX],BYTE PTR FOO
```

These statements tell the MS-Assembler that FOO is a byte immediate. A smaller instruction is generated.

3.2 Operands

The three types of operands are Immediate, Register, and Memory. There is no restriction on combining the types of operands.

The following list shows all the types and the items that constitute them:

- Immediate operands
  - Data items
  - Symbols
- Register operands
- Memory operands
  - Direct
    - Labels
    - Variables
    - Offset (fieldname)
  - Indexed
    - Base register
    - Index register
    - [constant]
    - Displacement
  - Structure

3.2.1 Immediate Operands

Immediate operands are constant values that you supply when you type a statement line. You may type the value either as a data item or as a symbol.

Instructions that take two operands permit an immediate operand as the source operand only (the second operand in an instruction statement). For example:

```
MOV AX,9
```

Data Items

The MS-Assembler recognizes values in forms other than decimal when you append special notation. The default input radix is decimal. The MS-Assembler treats any numeric values you enter without numeric notation appended as a decimal value. These other values include ASCII characters and numeric values.

Data Form	*Format*	*Example*
Binary	xxxxxxxxB	01110001B
Octal	xxxO	735O (letter O)
	xxxQ	412Q
Decimal	xxxxx	65535 (default)
	xxxxxD	1000D (when .RADIX changes input radix to nondecimal)
Hexadecimal	xxxxH	0FFFFH (1st digit must be 0-9)
ASCII	'xx'	'OM' (more than two with DB only;
	"xx"	"OM" both forms are synonymous)
10 real	xx.xxE& + xx	25.23E-7 (floating point format)
16 real	x...xR	8F76DEA9R (1st digit must be 0-9; the total number of digits must be 8, 16, or 20; or 9, 17, 21 if first digit is 0)

Symbols

You may use symbol names equated with some form of constant information (see Section 2.3, "Symbols") as immediate operands. Using a symbol constant in a statement is the same as using a numeric constant. Therefore, using the sample statement above, you could type:

```
    MOV AX,FOO
```

assuming FOO was defined as a constant symbol. For example:

```
    FOO EQU 9
```

3.2.2 Register Operands

The 8086 processor contains a number of registers. These registers are identified by two-letter symbols that the processor recognizes (the symbols are reserved).

The registers are appropriated to different tasks: general registers, pointer registers, counter registers, index registers, segment registers, and a flag register.

The general registers are two sizes: 8 bit and 16 bit. All other registers are 16 bit.

The 16-bit general registers are composed of a pair of 8-bit registers, one for the low byte (bits 0-7) and one for the high byte (bits 8-15). Note, however, that you can use each 8-bit general register independently of its mate. In this case, each 8-bit register contains bits 0-7.

You initialize segment registers, which contain segment base values. You can use the segment register names (CS, DS, SS, ES) with the colon segment override operator to inform the MS-Assembler that an operand is in a different segment than that specified in an ASSUME statement. (For further information, see Section 3.3.1, "Attribute Operators.")

The flag register is one 16-bit register containing nine 1-bit flags (six arithmetic flags and three control flags).

Each register (except segment registers and flags) can be an operand in arithmetic and logical operations.

Register/Memory Field Encoding:

Register Mode

MOD = 11		
R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

EFFECTIVE ADDRESS CALCULATION			
R/M	MOD = 00	MOD = 01	MOD = 10
000	[BX] + [SI]	[BX] + [SI] + D8	[BX] + [SI] + D16
001	[BX] + [DI]	[BX] + [DI] + D8	[BX] + [DI] + D16
010	[BP] + [SI]	[BP] + [SI] + D8	[BP] + [SI] + D16
011	[BP] + [DI]	[BP] + [DI] + D8	[BP] + [DI] + D16
100	[SI]	[SI] + D8	[SI] + D16
101	[DI]	[DI] + D8	[DI] + D16
110	DIRECT ADDRESS	[BP] + D8	[BP] + D16
111	[BX]	[BX] + D8	[BX] + D16

Note: D8 = a byte value; D16 = a word value

Other Registers:

Segment:	CS	code segment
	DS	data segment
	SS	stack segment
	ES	extra segment

Flags:	1-bit arithmetic flags		3 1-bit control flags	
	CF	carry flag	DF	direction flag
	PF	parity flag	IF	interrupt-enable flag
	AF	auxiliary flag	TF	trap flag
	ZF	zero flag		
	SF	sign flag		

Note:

You can also use the BX, BP, SI, and DI registers as memory operands. When these registers are enclosed in square brackets [ ], they are memory operands; when they are not enclosed in square brackets, they are register operands (see Section 3.2.3, "Memory Operands").

3.2.3 Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct the MS-Assembler to an address to find some data or instruction.

A memory operand always consists of an offset from a base address.

Memory operands fit into three categories: those that do not use a register (direct memory operands), those that use a base or index register (indexed memory operands), and structure operands.

Direct Memory Operands

Direct memory operands do not use a register, and they consist of a single offset value. Direct memory operands are labels, simple variables, and offsets.

You can use memory operands as destination operands and as source operands for instructions that take two operands. For example:

```
MOV AX,FOO
MOV FOO,CX
```

Indexed Memory Operands

Indexed memory operands use base and index registers, constants, displacement values, and variables, often in combination. When you combine indexed operands, you create an address expression.

Enclose indexed memory operands in square brackets to indicate indexing (by a register or by registers) or subscripting (for example, FOO[5]). The MS-Assembler treats square brackets as plus signs (+). Therefore,

```
FOO[5] is equivalent to FOO+5
5[FOO] is equivalent to 5+FOO
```

The only difference between square brackets and plus signs occurs when a register name appears inside the square brackets. Then, the operand is indexed.

The types of indexed memory operands are:

Base registers: [BX] [BP]

The default segment register of BP is SS; the default segment register of all others is DS.

Index registers: [DI] [SI]

[*constant*] Immediate in square brackets [8], [FOO]

+Displacement 8-bit or 16-bit value.
Use only with another indexed operand.

You may combine these elements in any order; however, you cannot combine two base registers and two indexed registers.

```
[BX+BP] ;illegal
[SI+DI] ;illegal
```

Some examples of indexed memory operand combinations:

```
[BP+8]
[SI+BX][4]
16[DI+BP+3]
8[FOO]-8
```

More examples of equivalent forms:

```
5[BX][SI]
[BX+5][SI]
[BX+SI+5]
[BX]5[SI]
```

Structure Operands

Structure operands take the form <*variable*>.<*field*>.

<*variable*> is any name you give when coding a statement line that initializes a Structure field. The <variable> may be an anonymous variable, such as an indexed memory operand.

<*field*> is a name defined by a DEFINE directive within a STRUC block. <*field*> is a typed constant.

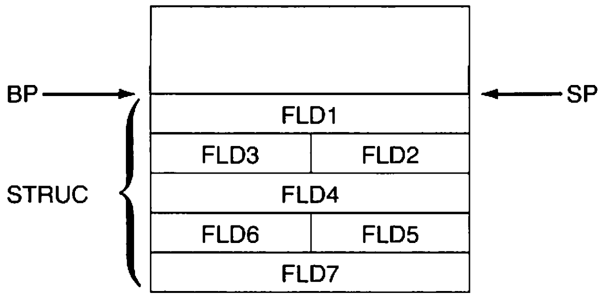
You must include the period (.).

Example:

```
ZOO          STRUC
GIRAFFE      DB  ?
ZOO          ENDS

LONG_NECK   ZOO  <16>

MOV AL,LONG_NECK.GIRAFFE

MOV AL,[BX].GIRAFFE ;anonymous variable
```

The use of structure operands can be helpful in stack operations. If you set up the stack segment as a structure, setting BP to the top of the stack (BP equal to SP), then you can access any value in the stack structure by field name indexed through BP. For example:

```
[BP].FLD6
```



This method makes all values on the stack available all the time, not just the value at the top. Therefore, this method makes the stack a handy place to pass parameters to subroutines.

3.3 Operators

The four types of operators are attribute, arithmetic, relational, and logical.

You use attribute operators with operands to override their attributes, to return the value of the attributes, or to isolate fields of records.

You use arithmetic, relational, and logical operators to combine or compare operands.

3.3.1 Attribute Operators

Attribute operators used as operands perform one of three functions:

Override an operand's attributes

Return the values of operand attributes

Isolate record fields (record specific operators)

The following list shows all the attribute operators by type:

Override operators
- PTR
- colon (:) (segment override)
- SHORT
- THIS
- HIGH
- LOW

Value-returning operators
- SEG
- OFFSET
- TYPE
- .TYPE
- LENGTH
- SIZE

Record specific operators
- Shift count (Field name)
- WIDTH
- MASK

Override Operators

You use these operators to override the segment, offset, type, or distance of variables and labels.

Pointer (PTR) <*expression*>

<*attribute*> PTR <*expression*>

The PTR operator overrides the type (BYTE, WORD, DWORD) or the distance (NEAR, FAR) of an operand.

<*attribute*> is the new attribute; the new type or new distance.

<*expression*> is the operand whose attribute is to be overridden.

The most important and frequent use of PTR is to ensure that the MS-Assembler understands what attribute the expression is supposed to have. This is especially true for the type attribute. Whenever you place forward references in your program, PTR clarifies the distance or type of the expression. This way you can avoid phase errors.

The second use of PTR is to access data by type other than the type in the variable definition. Most often this occurs in structures. If the structure is defined as WORD but you want to access an item as a byte, use PTR as the operator. A much easier method, however, is to enter a second statement that also defines the structure in bytes. This eliminates the need to use PTR for every reference to the structure. (See Section 4.2.1, "Memory Directives.")

Examples:

```
CALL WORD PTR [BX][SI]
MOV BYTE PTR ARRAY

ADD BYTE PTR FOO,9
```

Segment Override (:) (colon)

<segment-register>:<address-expression>
<segment-name>:<address-expression>
<group-name>:<address-expression>

The segment override operator overrides the assumed segment of an address expression (which may be a label, a variable, or other memory operand).

The colon operator helps with forward references by telling the MS-Assembler to what a reference is relative (segment, group, or segment register).

The MS-Assembler assumes that labels are addressable through the current CS register. It also assumes that variables are addressable through the current DS register, or possibly the ES register, by default. If the operand is in another segment and you have not alerted the MS-Assembler through the ASSUME directive, you need to use a segment override operator. If you want to use a nondefault relative base (that is, not the default segment register), you need to use the segment override operator for forward references. If the MS-Assembler can reach an operand through a nondefault segment register, it uses it, but the reference cannot be forward in this case.

<segment-register> is one of the four segment register names: CS, DS, SS, ES.

<segment-name> is a name defined by the SEGMENT directive.

<group-name> is a name defined by the GROUP directive.

Examples:

```
MOV AX,ES:[BX+SI]

MOV CSEG:FAR_LABEL,AX

MOV AX,OFFSET DGROUP:VARIABLE
```

SHORT

SHORT *<label>*

SHORT overrides NEAR distance attributes of labels used as targets for the JMP instruction. SHORT tells the MS-Assembler that the distance between the JMP statement and the *<label>* specified as its operand is not more than 127 bytes either direction.

The major advantage of using the SHORT operator is to save a byte. Normally, the *<label>* carries a 2-byte pointer to its offset in its segment. Because a range of 256 bytes can be handled in a single byte, the SHORT operator eliminates the need for the extra byte (which would carry 00 or FF anyway). Be sure, however, that the target is within 127 bytes of the JMP instruction before using SHORT.

Example:

```
                    JMP SHORT REPEAT
                    .
                    .
                    .
REPEAT:
```

THIS

THIS <*distance*>
THIS <*type*>

The THIS operator creates an operand. The value of the operand depends on which argument you give THIS.

The argument to THIS may be:

1. A distance (NEAR or FAR)

2. A type (BYTE, WORD, or DWORD)

THIS <*distance*> creates an operand with the distance attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

THIS <*type*> creates an operand with the type attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

Examples:

```
TAG EQU THIS BYTE same as TAG LABEL BYTE

SPOT_CHECK = THIS NEAR same as
SPOT_CHECK LABEL NEAR
```

HIGH,LOW

HIGH <*expression*>
LOW <*expression*>

HIGH and LOW are provided for 8080 assembly language compatibility. HIGH and LOW are byte isolation operators.

HIGH isolates the high 8 bits of an absolute 16-bit value or address expression.

LOW isolates the low 8 bits of an absolute 16-bit value or address expression.

Examples:

```
MOV AH,HIGH WORD_VALUE ;get byte with sign bit
MOV AL,LOW 0FFFFH
```

Value-Returning Operators

These operators return the attribute values of the operands that follow them but do not override the attributes.

The value-returning operators take labels and variables as their arguments.

Because variables in the MS-Assembler have three attributes, you need to use value-returning operators to isolate single attributes, as follows:

SEG	isolates the segment base address
OFFSET	isolates the offset value
TYPE	isolates either type or distance

LENGTH and SIZE isolate the memory allocation

SEG

SEG <*label*>
SEG <*variable*>

SEG returns the segment value (segment base address) of the segment enclosing the label or variable.

Example:

```
MOV AX, SEG VARIABLE_NAME
MOV AX, SEG <segment-variable>:<variable>
```

OFFSET

OFFSET <*label*>
OFFSET <*variable*>

OFFSET returns the offset value of the variable or label within its segment (the number of bytes between the segment base address and the address where the label or variable is defined).

You use OFFSET primarily to tell the MS-Assembler that the operand is an immediate operand.

NOTE

OFFSET does not make the value a constant. Only MS-LINK can resolve the final value. OFFSET is not required with uses of the DW or DD directives. The MS-Assembler applies an implicit OFFSET to variables in address expressions following DW and DD.

Example:

```
MOV BX,OFFSET FOO
```

If you use an ASSUME to GROUP, OFFSET does not automatically return the offset of a variable from the base address of the group. Rather, OFFSET returns the segment offset, unless you use the segment override operator (group-name version). If the variable GOB is defined in a segment placed in DGROUP, and you want the offset of GOB in the group, you need to enter a statement such as the following:

```
MOV BX,OFFSET DGROUP:GOB
```

Be sure that the GROUP directive precedes any reference to a group name, including its use with OFFSET.

TYPE

TYPE <*label*>
TYPE <*variable*>

If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the variable type, as follows:

BYTE	=	1
WORD	=	2
DWORD	=	4
QWORD	=	8
TBYTE	=	10
STRUC	=	the number of bytes declared by STRUC

If the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Example:

```
MOV AX,(TYPE FOO_BAR) PTR [BX+SI]
```

.TYPE

.TYPE <*variable*>

The .TYPE operator returns a byte that describes two characteristics of the <*variable*>: (1) the mode, and (2) whether or not it is External. The argument to .TYPE may be any expression (string, numeric, logical). If the expression is invalid, .TYPE returns zero.

The byte that is returned is configured as follows.

The lower two bits are the mode. If the lower two bits are:

0	the mode is Absolute
1	the mode is Program Related
2	the mode is Data Related

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is not External.

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

You usually use .TYPE inside macros, where you may need to test an argument to make a decision regarding program flow, for example, when conditional assembly is involved.

Example:

```
FOO        MACRO  X
           LOCAL  Z
Z          =  .TYPE X
IF         Z...
```

.TYPE tests the mode and type of X. Depending on the evaluation of X, the block of code beginning with IF Z... may be assembled or omitted.

LENGTH

LENGTH <*variable*>

LENGTH accepts only one variable as its argument.

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for that variable.

If the variable is defined by a DUP expression, LENGTH returns the number of type units duplicated, that is, the number that precedes the first DUP in the expression.

If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples:

```
FOO DW 100 DUP(1)

MOV CX,LENGTH FOO   ;get number of elements
                    ;in array
                    ;LENGTH returns 100


BAZ DW 100 DUP(1,10 DUP(?))
```

LENGTH BAZ is still 100, regardless of the expression following DUP.

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE

SIZE <*variable*>

SIZE returns the total number of bytes allocated for a variable.

SIZE returns the product of the value of LENGTH times the value of TYPE.

Example:

```
FOO DW 100 DUP(1)

MOV BX,SIZE FOO   ;get total bytes in array

SIZE = LENGTH  X TYPE
SIZE =   100   X WORD
SIZE =   100   X 2
SIZE = 200
```

Record Specific Operators

You use record specific operators to isolate fields in a record.

Records are defined by the RECORD directive (see Section 4.2.1, "Memory Directives"). A record may be a maximum length of 16 bits. The record is defined by fields, which may be from 1 to 16 bits long. To isolate one of the three characteristics of a record field, use one of the record specific operators, as follows:

Shift-count	Number of bits from low end of record to low end of field (number of bits to right shift the record to lowest bits of record)
WIDTH	The number of bits wide the field or record is (number of bits the field or record contains)
MASK	Value of record if field contains its maximum value and all other fields are zero (all bits in field contain 1; all other bits contain 0)

In the following discussions of the record specific operators, we use these symbols:

FOO	a record defined by the RECORD directive FOO RECORD FIELD1:3,FIELD2:6,FIELD3:7
BAZ	a variable used to allocate FOO BAZ FOO < >

FIELD1, FIELD2, and FIELD3 are the fields of the record FOO.

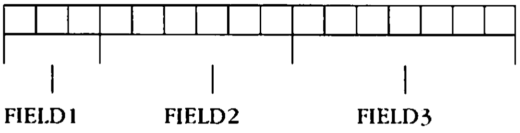
Shift-count - (*record-fieldname*)

<*record-fieldname*>

The shift-count is derived from the record fieldname to be isolated.

The shift-count is the number of bits the field must be shifted right to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (FOO) contains three fields (FIELD1, FIELD2, and FIELD3), the record can be diagrammed as follows:



FIELD1 has a shift-count of 13.
FIELD2 has a shift-count of 7.
FIELD3 has a shift-count of 0.

To isolate the value in one of these fields, enter its name as an operand.

Example:

```
MOV DX,BAZ
MOV CL,FIELD2
SHR DX,CL
```

FIELD2 is now right shifted, ready for access.

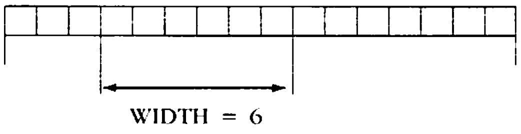
WIDTH

WIDTH <*record-fieldname*>
WIDTH <*record*>

When a <*record-fieldname*> is given as the argument, WIDTH returns the width of a record field as the number of bits in the record field.

When a <*record*> is given as the argument, WIDTH returns the width of a record as the number of bits in the record.

Using the diagram under shift-count, WIDTH can be diagrammed as:



The WIDTH of FIELD1 equals 3.
The WIDTH of FIELD2 equals 6.
The WIDTH of FIELD3 equals 7.

Example:

```
MOV CL,WIDTH FIELD2
```

The number of bits in FIELD2 is now in the count register.

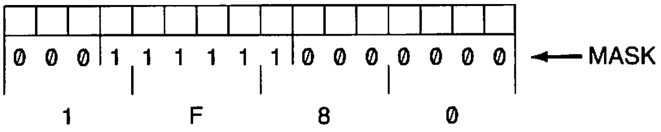
MASK

MASK <*record-fieldname*>

MASK accepts a fieldname as its only argument.

MASK returns a bit-mask defined by 1 for bit positions included by the field and 0 for bit positions not included. The value return represents the maximum value for the record when the field is masked.

Using the diagram for shift-count, MASK can be diagrammed as:



The MASK of FIELD2 equals 1F80H.

Example:

```
MOV DX,BAZ
AND DX,MASK FIELD2
```

FIELD2 is now isolated.

3.3.2 Arithmetic Operators

Eight arithmetic operators provide the common mathematical functions (add, subtract, divide, multiply, modulo, negation), plus two shift operators.

You use the arithmetic operators to combine operands to form an expression that results in a data item or an address.

Except for + and − (binary), operands must be constants.

For plus (+), one operand must be a constant.

For minus (−), the first (left) operand may be a nonconstant, or both operands may be nonconstants. The right must be a constant if the left is a constant.

*	Multiply
/	Divide
MOD	Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolute. Example: `MOV AX,100 MOD 17` The value moved into AX is 0FH (decimal 15).
SHR	Shift Right. SHR is followed by an integer that specifies the number of bit positions the value is to be shifted right. Example: `MOV AX,1100000B SHR 5` The value moved into AX is 11B (03).
SHL	Shift Left. SHL is followed by an integer that specifies the number of bit positions the value is to be shifted left. Example: `MOV AX,0110B SHL 5` The value moved into AX is 011000000B (0C0H).
– (Unary Minus)	Indicates that following value is negative, as in a negative integer.
+	Add. One operand must be a constant; one may be a nonconstant.
–	Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be nonconstants. But the right may be a nonconstant only if the left is also a nonconstant and in the same segment.

3.3.3 Relational Operators

Relational operators compare two constant operands.

If the relationship between the two operands matches the operator, FFFFH is returned.

If the relationship between the two operands does not match the operator, a zero is returned.

You most often use relational operators with conditional directives and conditional instructions to direct program control.

EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less Than or Equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater Than or Equal. Returns true if the left operand is greater than or equal to the right operand.

3.3.4 Logical Operators

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate the logical relationship defined by the logical operator.

You can use logical operators in two ways:

1. To combine operands in a logical relationship. In this case, all bits in the operands have the same value (either 0000 or FFFFH). It is best to use these values for true (FFFFH) and to use for false (0000) the symbols you use as operands, because in conditionals anything nonzero is true.

2. In bitwise operations. In this case, the bits are different, and the logical operators act the same as the instructions of the same name.

NOT	Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.

AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values.
XOR	Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values.

3.3.5 Expression Evaluation: Precedence of Operators

Expressions are evaluated higher precedence operators first, then left to right for equal precedence operators.

You can use parentheses to alter precedence.

For example:

```
MOV AX,101B SHL 2*2 = MOV AX,00101000B

MOV AX,101B SHL (2*2) = MOV AX,01010000B
```

SHL and * are equal precedence. Therefore, their functions are performed in the order the operators are encountered (left to right).

Precedence of Operators

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: parentheses ( )
   angle brackets < >
   square brackets [ ]
   Structure variable operand: <*variable*>.<*field*>
2. Segment override operator: colon (:)
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MOD, SHL, SHR
6. +, − (both unary and binary)

7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT,.TYPE

Actions

Chapter 4

Action: Instructions and Directives

Action: Instructions and Directives

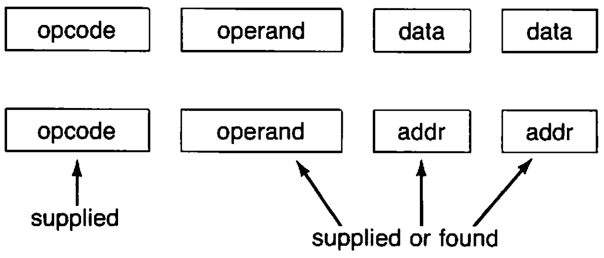
The action field contains either an 8086 instruction mnemonic or an MS-Assembler directive.

Following a name field entry (if any), action field entries may begin in any column. Specific spacing is not required. The only benefit of consistent spacing is improved readability. If a statement does not have a name field entry, the action field is the first entry.

The entry in the action field directs either the processor or the MS-Assembler to perform a specific function.

4.1 Instructions

Instructions tell the command processor to perform some action. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expressions in instruction statements.

(opcode equates to the binary code for the action of an instruction)

This manual does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this information, we recommend that you consult the following texts:

1. Morse, Stephen P. *The 8086 Primer.* Rochelle Park, NJ: Hayden Publishing Co., 1980.
2. Rector, Russell and Alexy, George. *The 8086 Book.* Berkeley, CA: Osbourne/McGraw-Hill, 1980.
3. *The 8086 Family User's Manual.* Santa Clara, CA: Intel Corporation, 1980.

Appendix D contains an alphabetical listing of the instruction mnemonics.

4.2 Directives

Directives give the MS-Assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions.

The directives are divided into groups by the function they perform. Within each group, the directives are described alphabetically.

The groups are:

Memory Directives
: You use directives in this group to organize memory. Because there is no "miscellaneous" group, the memory directives group contains some directives that do not, strictly speaking, organize memory (for example, COMMENT).

Conditional Directives
: You use directives in this group to test conditions of assembly before proceeding with assembly of a block of statements. This group contains all the IF (and related) directives.

Macro Directives
: You use directives in this group to create blocks of code called macros. This group also includes some special operators and directives that are used only inside macro blocks. The repeat directives are considered macro directives for descriptive purposes.

Listing Directives
: You use directives in this group to control the format and, to some extent, the content of listings that the MS-Assembler produces.

Below is an alphabetical list of all directives the MS-Assembler supports:

ASSUME	EVEN	IRPC	.RADIX
	EXITM		RECORD
COMMENT	EXTERN	LABEL	REPT
.CREF		.LALL	
	GROUP	.LFCOND	.SALL
DB		.LIST	SEGMENT
DD	IF		.SFCOND
DQ	IFB	MACRO	STRUC
DT	IFDEF		SUBTTL
DW	IFDIF	NAME	
	IFE		.TFCOND
ELSE	IFIDN	ORG	TITLE
END	IFNB	%OUT	
ENDIF	IFNDEF		.XALL
ENDM		PAGE	.XCREF
ENDP	IF1	PROC	.XLIST
ENDS	IF2	PUBLIC	
EQU	IRP	PURGE	

4.2.1 Memory Directives

ASSUME

ASSUME <*seg-reg*>:<*seg-name*>[, . . .]

or

ASSUME NOTHING

ASSUME tells the MS-Assembler that the symbols in the segment or group can be accessed using this segment register. When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from 1 to 4 arguments to ASSUME.

The valid <*seg-reg*> entries are:

CS, DS, ES, and SS.

The possible entries for <*seg-name*> are:

1. The name of a segment declared with the SEGMENT directive
2. The name of a group declared with the GROUP directive
3. An expression: either SEG <*variable-name*> or SEG <*label-name*> (see SEG operator, Section 3.3)

4. The key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement

If you do not use ASSUME or if you type NOTHING for <*seg-name*>, you must prefix each reference to variables, symbols, labels, and so forth in a particular segment by a segment register. For example, type DS:FOO instead of simply FOO.

Example:

```
ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHING
```

COMMENT

COMMENT<*delim*><*text*><*delim*>

The first nonblank character encountered after COMMENT is the delimiter. The following <*text*> constitutes a comment block that continues until the next occurrence of <*delimiter*>.

COMMENT lets you enter comments about your program without placing a semicolon (;) before each line.

If you use COMMENT inside a macro block, the comment block does not appear on your listing unless you also place the .LALL directive in your source file.

Example:

Using an asterisk as the delimiter, the format of the comment block would be:

```
COMMENT *
any amount of text entered
here as the comment block
 .
 .
 .   *  ;return to normal mode
```

DEFINE BYTE
DEFINE WORD
DEFINE DOUBLEWORD
DEFINE QUADWORD
DEFINE TENBYTES

<*varname*>	DB	<*exp*>[,<*exp*>, . . .]
<*varname*>	DW	<*exp*>[,<*exp*>, . . .]
<*varname*>	DD	<*exp*>[,<*exp*>, . . .]
<*varname*>	DQ	<*exp*>[,<*exp*>, . . .]
<*varname*>	DT	<*exp*>[,<*exp*>, . . .]

You use the DEFINE directives to define variables or to initialize portions of memory.

If you enter the optional <*varname*>, the DEFINE directives define the name as a variable. If <*varname*> has a colon, it becomes a NEAR label instead of a variable. (See Section 2.1, "Labels," and Section 2.2, "Variables.")

The DEFINE directives allocate memory in units specified by the second letter of the directive (each DEFINE directive may allocate one or more of its units at a time).

DB allocates 1 byte (8 bits)
DW allocates 1 word (2 bytes)
DD allocates 2 words (4 bytes)
DQ allocates 4 words (8 bytes)
DT allocates 10 bytes

<*exp*> may be one or more of the following:

1. A constant expression
2. The question mark (?) for indeterminate initialization. Usually you use the question mark to reserve space without placing any particular value into it.
3. An address expression (for DW and DD only)
4. An ASCII string (longer than two characters for DB only)
5. <*exp*>DUP(?)
   When this type of expression is the only argument to a define directive, the define directive produces an uninitialized data block. This expression with the question mark instead of a value results in a smaller object file because only the segment offset is changed to reserve space.
6. <*exp*> DUP(<*exp*>[, . . .])
   This expression, like item 5, produces a data block, but initialized with the value of the second <*exp*>. The first <*exp*> must be a constant greater than zero and must not be a forward reference.

Example — Define Byte (DB):

```
NUM_BASE        DB      16
FILLER          DB      ?             ;initialize with
                                      ;indeterminate value
ONE_CHAR        DB      'M'
MULT_CHAR       DB      'TOM JEROME EDWARD BOB DEAN'
MSG             DB      'MSGTEST',13,10
                                      ;message, carriage return
                                      ;and linefeed
BUFFER          DB      10 DUP(?)     ;indeterminate block
TABLE           DB      100 DUP(5 DUP(4),7)
                                      ;100 copies of bytes
                                      ;with values 4,4,4,4,4,7
NEW_PAGE        DB      0CH           ;form feed character
ARRAY           DB      1,2,3,4,5,6,7
```

Example — Define Word (DW):

```
ITEMS           DW      TABLE,TABLE+10,TABLE+20
SEGVAL          DW      0FFF0H
BSIZE           DW      4 * 128
LOCATION        DW      TOTAL + 1
AREA            DW      100 DUP(?)
CLEARED         DW      50 DUP(0)
SERIES          DW      2 DUP(2,3 DUP(BSIZE))
                        ;two words with the byte values
                        ;2,BSIZE,BSIZE,BSIZE,2,BSIZE,BSIZE,
                        ;BSIZE
DISTANCE        DW      START_TAB -END_TAB
                        ;difference of two labels is a
                        ;constant
```

Example — Define Doubleword (DD):

```
DBPTR             DD        TABLE
                        ;16-bit OFFSET,
                        ;then 16-bit
                        ;SEG base value
SEC_PER_DAY       DD        60*60*24
                        ;arithmetic is performed
                        ;by the assembler
LIST              DD        'XY',2 DUP(?)
HIGH 1            DD        4294967295
                        ;maximum
FLOAT             DD        6.735E2
                        ;floating point
```

Example — Define Quadword (DQ):

```
LONG_REAL         DQ        3.141597
                        ;decimal makes
                        ;it real
STRING            DQ        'AB'
                        ;no more than 2
                        ;characters
HIGH 1            DQ    18446744073709661615
                        ;maximum
LOW 1             DQ        -18446744073709661615
                        ;minimum
SPACER            DQ        2 DUP(?)
                        ;uninit.data
FILLER            DQ        1 DUP(?,?)
                        ;initalized w_/
                        ;indeterminate
                        ;value
HEX_REAL          DQ        0FDCBA9A98765432105R
```

Example — Define Tenbytes (DT):

```
ACCUMULATOR       DT        ?
STRING            DT        'CD'
                        ;no more than 2
                        ;characters
PACKED_DECIMAL    DT        1234567890
FLOATING_POINT    DT        3.1415926
```

END

END [<*exp*>]

The END statement specifies the end of the program.

If <*exp*> is present, it is the start address of the program. If you want to link several modules, only the main module may specify the start of the program with the END <*exp*> statement.

If <*exp*> is not present, then the MS-Assembler does not pass a start address to MS-LINK for that program or module.

Example:

```
END   START          ;START is a label somewhere
                     ;in the program
```

EQU

<*name*> EQU <*exp*>

EQU assigns the value of <*exp*> to <*name*>. If <*exp*> is an external symbol, an error is generated. If <*name*> already has a value, an error is generated. If you want to be able to redefine a <*name*> in your program, use the equal sign (=) directive instead.

In many cases, you can use EQU as a primitive text substitution, like a macro.

<*exp*> may be any one of the following:

1. A symbol. <*name*> becomes an alias for the symbol in <*exp*>. Shown as an Alias in the symbol table.
2. An instruction name. Shown as an Opcode in the symbol table.
3. A valid expression. Shown as a Number or L (label) in the symbol table.
4. Any other entry, including text, index references, segment prefix and operands. Shown as Text in the symbol table.

Example:

```
FOO     EQU     BAZ
                     ;must be defined in this
                     ;module or an error
                     ;results
B       EQU     [BP+8]
                     ;index reference (Text)
P8      EQU     DS:[BP+8]
                     ;segment prefix
                     ;and operand (Text)
CBD     EQU     AAD
                     ;an instruction name
                     ;(Opcode)
ALL     EQU   DEFREC<2,3,4>
                     ;DEFREC = record name
                     ;<2,3,4> = initial values
                     ;for fields of record
EMP     EQU     6
                     ;constant value
FPV     EQU     6.3E7
                     ;floating point (text)
```

Equal Sign

<name = <exp>

<exp> must be a valid expression. It is shown as a Number or L (label) in the symbol table (same as *<exp>* type 3 under the EQU directive above).

The equal sign (=) lets you set and redefine symbols. The equal sign is like the EQU directive, except you can redefine the symbol without generating an error. You may redefine more than once, and a redefinition may refer to a previous definition.

Example:

```
FOO      =      5
                     ;the same as FOO EQU 5
FOO     EQU     6;
                     ;error, FOO cannot be
                     ;redefined by EQU
FOO      =      7
                     ;FOO can be redefined
                     ;only by another =
FOO      =      FOO+3
                     ;redefinition may refer
                     ;to a previous definition
```

EVEN

EVEN

The EVEN directive sends the program counter to an even boundary, that is, to an address that begins a word. If the program counter is not already at an even boundary, the MS-Assembler adds an NOP instruction so that the counter reaches an even boundary.

An error results if you use EVEN with a byte-aligned segment.

Example:

Before: The PC points to 0019 hex (25 decimal)

EVEN

After: The PC points to 1A hex (26 decimal); 0019 hex now contains a NOP instruction

EXTRN

EXTRN <*name*>:<*type*>[, . . .]

<*name*> is a symbol that is defined in another module. <*name*> must have been declared PUBLIC in the module where <*name*> is defined.

<*type*> may be any one of the following, but must be a valid type for <*name*>:

1. BYTE, WORD, or DWORD
2. NEAR or FAR for labels or procedures (defined under a PROC directive)
3. ABS for pure numbers (implicit size is WORD, but includes BYTE)

Placement of the EXTRN directive is significant. If you give the directive with a segment, the MS-Assembler assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments, then use either

```
ASSUME <seg-reg>:SEG <name>
```

or an explicit segment prefix.

NOTE

If a mistake is made and the symbol is not in the segment, MS-LINK takes the offset relative to the given segment, if possible. If the real segment is less than 64K bytes away from the reference, MS-LINK may find the definition. If the real segment is more than 64K bytes away, MS-LINK cannot link the reference and the definition and returns an error message.

Example:

```
In Same Segment:                 In Another Segment:
In Module 1:                     In Module 1:

CSEG    SEGMENT                  CSEGA   SEGMENT
        PUBLIC TAGN                              PUBLIC TAGF
        .                                .
        .                                .
        .                                .
TAGN:                            TAGF:
        .                                .
        .                                .
        .                                .
CSEG    ENDS                     CSEGA   ENDS

In Module 2:                     In Module 2:

CSEG    SEGMENT                          EXTRN TAGF:FAR
        EXTRN TAGN:NEAR                  CSEGB   SEGMENT
        .                                .
        .                                .
        .                                .
        JMP TAGN                                 JMP TAGF
CSEG    ENDS                     CSEGB   ENDS
```

GROUP

<*name*> GROUP <*seg-name*>[, . . .]

The GROUP directive collects the segments named after GROUP (<*seg-name*>s) under one name. MS-LINK uses the GROUP to know which segments to load together. The order in which the segments are named does not influence the order in which they are loaded. The loading order is determined by the CLASS designation of the SEGMENT directive or by the order in which you name object modules in response to the MS-LINK Object Module: prompt.

All segments in a GROUP must fit into 64K bytes of memory. MS-LINK checks this; the MS-Assembler does not.

<*seg-name*> may be one of the following:

1. A segment name, assigned by a SEGMENT directive. The name may be a forward reference.

2. An expression: either SEG <*var*>
   or SEG <*label*>

   Both these entries resolve themselves to a segment name (see SEG operator, Section 3.3).

After you define a group name, you can use the name:

1. As an immediate value:

   ```
   MOV AX,DGROUP
   MOV DS,AX
   ```

   DGROUP is the paragraph address of the base of DGROUP.

2. In ASSUME statements:

   ```
   ASSUME DS:DGROUP
   ```

   You can now use the DS register to reach any symbol in any segment of the group.

3. As an operand prefix (for segment override):

   ```
   MOV   BX,OFFSET DGROUP:FOO
   DW    DGROUP:FOO
   DD    DGROUP:FOO
   ```

   DGROUP: forces the offset to be relative to DGROUP, instead of to the segment in which FOO is defined.

Example (Using GROUP to combine segments):

In Module A:

```
CGROUP      GROUP       XXX,YYY
XXX         SEGMENT
            ASSUME      CS:CGROUP
              .
              .
              .
XXX         ENDS
YYY         SEGMENT
              .
              .
              .
YYY         ENDS
            END
```

In Module B:

```
CGROUP    GROUP      ZZZ
ZZZ       SEGMENT
          ASSUME     CS:CGROUP
          --
          --
          --
ZZZ       ENDS
          END
```

INCLUDE

INCLUDE <*filename*>

The INCLUDE directive inserts source code from an alternate assembly language source file into the current source file during assembly. Use of the INCLUDE directive eliminates the need to repeat an often-used sequence of statements in the current source file.

The <*filename*> is any valid file specification for the operating system. If the device designation is other than the default, the source filename specification must include it. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the INCLUDE directive statement. When end-of-file is reached, assembly resumes with the next statement following the INCLUDE directive.

You may nest INCLUDES (the file inserted with an INCLUDE statement may contain an INCLUDE directive). However, we do not recommend that you use nesting with small systems because of the amount of memory that may be required.

The file specified must exist. If the MS-Assembler does not find the file, your screen displays an error message, and assembly ceases.

On an MS-Assembler listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See Section 7.5, "Formats of Listings and Symbol Tables," for a description of listing file formats.

Example:

```
INCLUDE ENTRY
INCLUDE B:RECORD.TST
```

LABEL

<*name*> LABEL <*type*>

When you define a <*name*> with LABEL, the MS-Assembler associates the current segment offset with <*name*>.

The item is assigned a length of 1.

<*type*> varies depending on the use of <*name*>. You may use <*name*> for code or for data.

1. For code (for example, as a JMP or CALL operand):

   <*type*> may be either NEAR or FAR. You cannot use <*name*> in data manipulation instructions without using a type override.

   If you wish, you can define a NEAR label with <*name*>: form (in this case, do not use the LABEL directive). If you are defining a BYTE or WORD NEAR label, you can place <*name*>: in front of a Define directive.

   When using a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

   Example — For Code:

   ```
   SUBRTF LABEL  FAR
   SUBRT: (first instruction)   ;colon = NEAR label
   ```

2. For data:

   <*type*> may be BYTE, WORD, DWORD, <*structure-name*>, or <*record-name*>. When you use STRUC or RECORD name, <*name*> is assigned the size of the structure or record.

   Example — For Data:

   ```
   BARRAY      LABEL   BYTE
   ARRAY       DW      100 DUP(0)
                .
                .
                .
               ADD    AL,BARRAY[99]       ;ADD 100th byte to AL
               ADD    AX,ARRAY[98]        ;ADD 50th word to AX
   ```

By defining the array two ways, you can access entries either by byte or by word. Also, you can use this method for STRUC. It lets you place data in memory as a table and access it without the offset of the STRUC.

If you define the array in two ways, you do not have to use the PTR operator. Double definitions are especially effective if you access the data in different ways. It is easier to give the array a second name than to remember to use PTR.

NAME

NAME <*module-name*>

<*module-name*> must not be a reserved word. The module name may be any length, but the MS-Assembler recognizes only the first six characters.

The module name is passed to MS-LINK, but the MS-Assembler checks to see if more than one module name has been declared.

Every module has a name, which is derived from:

1. A valid NAME directive statement
2. The first six characters of a TITLE directive statement, if the module does not contain a NAME statement. The first six characters must be legal as a name.

Example:

```
NAME CURSOR
```

ORG

ORG <*exp*>

The location counter is set to the value of <*exp*>, and the MS-Assembler assigns generated code starting with that value.

All names used in <*exp*> must be known on Pass 1. The value of <*exp*> must either evaluate to an absolute or must be in the same segment as the location counter.

Example:

```
ORG          120H          ;2-byte absolute value
              ;maximum=0FFFFH
ORG          $+2           ;skip two bytes
```

Example — ORG to a boundary (conditional):

```
CSEG        SEGMENT    PAGE
BEGIN       =          $
             .
             .
             .
IF ($-BEGIN) MOD 256                  ;if not already on
                                      ;256-byte boundary
            ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
ENDIF
```

See Section 4.2.2, "Conditional Directives," for an explanation of conditional assembly.

PROC

```
<procname>        PROC  [NEAR]
                     or [FAR]
                   .
                   .
                   .
                   RET
<procname>        ENDP
```

The default, if no operand is specified, is NEAR. Use FAR if:

1. The procedure name is an operating system entry point
2. The procedure will be called from code that has another ASSUME CS value

Each PROC block should contain a RET statement.

The PROC directive serves as a structuring device to make your programs more understandable.

The PROC directive, through the NEAR/FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL, and RETs to generate a NEAR or a FAR RET. You use PROC for coding simplification so that you do not have to worry about NEAR or FAR for CALLs and RETs.

A NEAR CALL or RETURN changes the IP but not the CS register. A FAR CALL or RETURN changes both the IP and the CS registers.

Procedures are executed either in line, from a JMP, or from a CALL.

PROCs may be nested, which means that they are put in line.

Combining the PUBLIC directive with a PROC statement (both NEAR and FAR) lets you make external CALLs to the procedure or to make other external references to the procedure.

Example:

```
                PUBLIC     FAR_NAME
FAR_NAME                  PROC        FAR
                CALL       NEAR_NAME
                RET

FAR_NAME                  ENDP

                PUBLIC     NEAR_NAME
NEAR_NAME                 PROC        NEAR
               .
               .
               .
              RET
NEAR_NAME                 ENDP
```

You can call the second subroutine above directly from a NEAR segment (that is, a segment addressable through the same CS and within 64K):

```
CALL NEAR_NAME
```

A FAR segment (that is, any other segment that is not a NEAR segment) must call the first subroutine, which then calls the second (an indirect call):

```
CALL FAR_NAME
```

PUBLIC

PUBLIC <*symbol*>[, . . .]

Place a PUBLIC directive statement in any module that contains symbols you want to use in other modules without defining the symbol again. PUBLIC makes the listed symbol(s), which are defined in the module where the PUBLIC statement appears, available for use by other modules to be linked with the module that defines the symbol(s). This information is passed to MS-LINK.

<*symbol*> may be a number, a variable, or a label (including PROC labels).

<*symbol*> may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes.

Example:

```
            PUBLIC      GETINFO
GETINFO     PROC        FAR
            PUSH        BP          ;save caller's register
            MOV         BP,SP       ;get address parameters
                                    ;body of subroutine
            POP         BP          ;restore caller's reg
            RET                     ;return to caller
GETINFO     ENDP
```

Example — illegal PUBLIC:

```
            PUBLIC PIE_BALD,HIGH_VALUE
PIE_BALD                EQU         3.1416
HIGH_VALUE EQU 999999999
```

.RADIX

.RADIX <*exp*>

The default input base (or radix) for all constants is decimal. The .RADIX directive lets you change the input radix to any base in the range 2 to 16.

<*exp*> is always in decimal radix, regardless of the current input radix.

Example:

```
   MOV         BX,0FFH
   .RADIX      16
   MOV         BX,0FF
```

The two MOVs in this example are identical.

The .RADIX directive does not affect the generated code values placed in the .OBJ, .LST, or .CRF output files.

The .RADIX directive does not affect the DD, DQ, or DT directives. Numeric values entered in the expression of these directives are always evaluated as decimal unless a data type suffix is appended to the value.

Example:

```
    .RADIX 16
NUM_HAND       DT    773     ;773 = decimal
HOT_HAND       DQ    773Q    ;773 = octal here only
COOL_HAND      DD    773H    ;now 773 = hexadecimal
```

RECORD

<recordname> RECORD
<fieldname>:*<width>*[= *<exp>*],[. . .]

<fieldname> is the name of the field. *<width>* specifies the number of bits in the field defined by *<fieldname>*. *<exp>* contains the initial (or default) value for the field. You may not include forward references in a RECORD statement.

<fieldname> becomes a value that you can use in expressions. When you use *<fieldname>* in an expression, its value is the shift-count to move the field to the far right. Using the MASK operator with the *<fieldname>* returns a bit mask for that field.

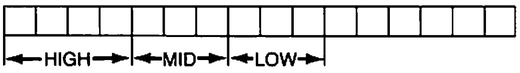
<width> is a constant in the range 1 to 16 that specifies the number of bits contained in the field defined by *<fieldname>*. The WIDTH operator returns this value. If the total width of all declared fields is larger than 8 bits, then the MS-Assembler uses two bytes. Otherwise, it uses only one byte.

The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

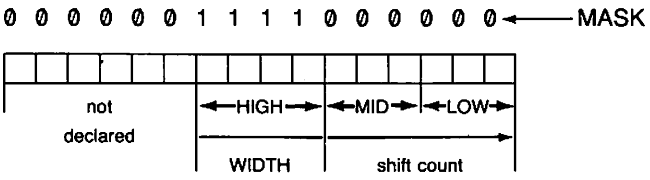
Example:

```
FOO RECORD HIGH:4,MID:3,LOW:3
```

Initially, the bit map would be:



Total bits >8 means use a word; but total bits <16 means right shift, place undeclared bits at high end of word. Thus:



<exp> contains the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character as the *<exp>*.

Example:

```
HIGH:7='Q'
```

To initialize records, use the same method as that for DB. The format is:

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

        or

    [<name<> <recordname> [<exp>
DUP(<[exp][, . . . ]>)
```

The name is optional. When given, name is a label for the first byte or word of the record storage area.

The recordname is the name used as a label for the RECORD directive.

The [*exp*] (both forms) contains the values you want placed into the fields of the record. In the latter case, the parentheses and angle brackets are required only around the second [*exp*] (following DUP). If [*exp*] is left blank, either the default value applies (the value given in the indeterminate (when not initialized in the original record definition). For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields.

For example:

```
    FOO <, ,7>
```

From the previous example, the 7 would be placed into the LOW field of the record FOO. The fields HIGH and MID would be left as declared (in this case, uninitialized).

You may use records in expressions (as an operand) in the form:

```
    recordname<[value[, . . . ]]>
```

The value entry is optional. The angle brackets must be coded as shown, even if the optional values are not given. A value entry is the value to be placed into a field of the record. For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields, as shown above.

Example:

```
FOO       RECORD    HIGH:5,MID:3,LOW:3
            .
            .
            .
BAX       FOO       <>   ;leave undeterminate  here
JANE      FOO       10 DUP(<16,8>)   ;HIGH=16,
                    ;MID=8, LOW=?
            .
            .
            .
          MOV       DX,OFFSET JANE[2]
                    ;get beginning record
                    ;address
          AND       DX,MASK MID
          MOV       CL,MID
          SHR       DX,CL
          MOV       CL,WIDTH MID
```

SEGMENT

<segname> SEGMENT [<*align*>] [<*combine*>]
[<'*class*'>]

.
.
.

<segname> ENDS

At runtime, all instructions that generate code and data are in (separate) segments. Your program may be a segment, part of a segment, several segments, parts of several segments, or a combination of these. If a program has no SEGMENT statement, an MS-LINK error (invalid object) results at link time.

The <*segment name*> must be unique and legal. The segment name must not be a reserved word.

<*align*> may be PARA (paragraph — default), BYTE, WORD, or PAGE.

<*combine*> may be PUBLIC, COMMON, AT <*exp*>, STACK, MEMORY, or no entry (which defaults to not combinable).

<*class*> name is used to group segments at link time.

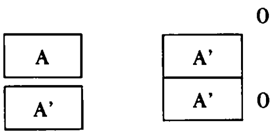
All three operands are passed to MS-LINK.

The *alignment* type tells MS-LINK on what kind of boundary you want the segment to begin. The first address of the segment for each alignment type is:

PAGE — address is xxx00H (low byte is 0)
PARA — address is xxxx0H (low nibble is 0)
bit map — |x|x|x|x|0|0|0|0|
WORD — address is xxxxeH (e = even number;low bit is 0)
bit map — |x|x|x|x|x|x|x|0|
BYTE — address is xxxxxH (place anywhere)

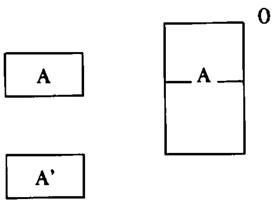
The *combine* type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or Private)



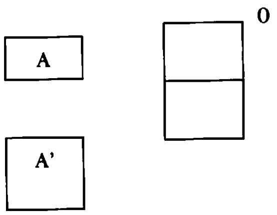
Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

Public and Stack



Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine type stack is treated the same as public. However, the Stack Pointer is set to the first address of the first stack segment. MS-LINK requires at least one stack segment.)

Common



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Memory

The memory combine type causes the segment(s) to be placed as the highest segments in memory. The first memory combinable segment encountered is placed as the highest segment in memory. Subsequent segments are treated the same as Common segments.

NOTE

This feature is not supported by MS-LINK. MS-LINK treats Memory segments the same as Public segments.

AT <*exp*>

The segment is placed at the PARAGRAPH address specified in <*exp*>. The expression may not be a forward reference. Also, you may not use the AT type to force loading at fixed addresses. Rather, the AT combine type lets you define labels and variables at fixed offsets within fixed areas of storage, such as ROM or the vector space in low memory.

NOTE

This restriction is imposed by MS-LINK and MS-DOS.

You must enclose *class* names (any legal name) in quotation marks.

You may nest segment definitions. When segments are nested, the MS-Assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, the MS-Assembler takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. You may not use overlapping segments.

For example:

```
A        SEGMENT      |         | A SEGMENT
            .         |         |    .
            .         |         |    .
            .         |         |    .
B        SEGMENT      |         | A ENDS
            .         |         | B           SEGMENT
            .         |  ---->  |    .
            .         |         |    .
B        ENDS         |         |    .
            .         |         | B           ENDS
            .         |         | A           SEGMENT
            .         |         |    .
A        ENDS         |         |    .
                                |    .
                                | A ENDS
```

The following arrangement is not allowed:

```
    A           SEGMENT
                  .
                  .
    B           SEGMENT
                  .
                  .
    A           ENDS          ;This is illegal!
                  .
                  .
    B           ENDS
```

Example:

In module A:

```
SEGA        SEGMENT     PUBLIC 'CODE'
            ASSUME      CS:SEGA
              .
              .
              .
SEGA        ENDS
            END
```

In module B:

```
SEGA        SEGMENT     PUBLIC 'CODE'
            ASSUME      CS:SEGA
              .        ;MS-LINK adds this segment to same named
              .        ;segment in module A (and others) if class
              .        ;name is the same.
SEGA        ENDS
            END
```

STRUC

```
<structurename>             STRUC
                              .
                              .
                              .
<structurename>             ENDS
```

The STRUC directive is very much like RECORD, except STRUC has a multiple byte capability. The allocation and initialization of a STRUC block are the same as for RECORDs.

Inside the STRUC/ENDS block, you may allocate space with the Define directives (DB, DW, DD, DQ, DT). The Define directives and Comments set off by semicolons (;) are the only statement entries allowed inside a STRUC block.

Any label on a Define directive inside a STRUC/ENDS block becomes a <*fieldname*> of the structure. (This is how structure fieldnames are defined.) Initial values given to fieldnames in the STRUC/ENDS block are default values for the various fields. Some field values can be overridden; others cannot. A simple field, a field with only one entry (but not a DUP expression), can be overridden. A multiple field, a field with more than one entry, cannot be overridden. For example:

```
FOO     DB      1,2             ;cannot be
                                ;overridden
BAZ     DB      10 DUP(?)       ;cannot be
                                ;overridden
ZOO     DB      5               ;can be
                                ;overridden
```

If the <*exp*> following the Define directive contains a string, it may be overridden by another string. However, if the overriding string is shorter than the initial string, the MS-Assembler pads with spaces. If the overriding string is longer, the MS-Assembler truncates the extra characters.

Usually, structure fields are used as operands in some expression. The format for a reference to a structure field is:

<variable>.<field>

<*variable*> represents an anonymous variable, usually set up when the structure is allocated. To allocate a structure, use the structure name as a directive with a label (the anonymous variable of a structure reference) and any override values in angle brackets:

```
FOO         STRUCTURE
              .
              .
              .
FOO         ENDS
GOO         FOO       <,7,,'JOE'>
```

.<*field*> represents a label given to a DEFINE directive inside a STRUC/ENDS block (you must code the period). The value of <*field*> will be the offset within the addressed structure.

Example:

To define a structure:

```
S   STRUC
FIELD1      DB      1,2             ;cannot be
                                    ;overridden
FIELD2      DB      10 DUP(?)       ;cannot be
                                    ;overridden
FIELD3      DB      5               ;can be overridden
FIELD4      DB      'DOBOSKY'       ;can be overridden
S           ENDS
```

The Define directives in this example define the fields of the structure, and the order corresponds to the order values that are given in the initialization list when the structure is allocated. Every Define directive statement line inside a STRUC block defines a field, whether or not the field is named.

To allocate the structure:

```
DBAREA      S       <, ,7,'ANDY'>       ;overrides 3rd and
                                        ;4th fields only
```

To refer to a structure:

```
            MOV         AL,[BX].FIELD3
            MOV         AL,DBAREA.FIELD3
```

4.2.2 Conditional Directives

Conditional directives let you design blocks of code that test for specific conditions.

All conditionals follow the format:

```
IFxxxx [argument]
 .
 .
 .
[ELSE
 .
 .
 . ]
ENDIF
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Otherwise, an "Unterminated conditional" message is generated at the end of each pass. An ENDIF without a matching IF causes a Code 8, "Not in conditional block" error.

Each conditional block may include the optional ELSE directive, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a Code 7, "Already had ELSE clause" error.

You may nest conditionals up to 255 levels. Any argument to a conditional must be known on Pass 1 to avoid Phase errors and incorrect evaluation. For IF and IFE, the expression must involve values that were previously defined, and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, Pass 1 considers the name undefined, but it is defined on Pass 2.

The MS-Assembler evaluates the conditional statement to TRUE (which equals any nonzero value), or to FALSE (which equals 0000H). If the evaluation matches the condition defined in the conditional statement, the MS-Assembler either assembles the whole conditional block or, if the conditional block contains the optional ELSE directive, assembles from IF to ELSE; the ELSE to ENDIF portion of the block is ignored. If the evaluation does not match, the MS-Assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE directive, assembles only the ELSE to ENDIF portion; the IF to ELSE portion is ignored.

The following is a list of MS-Assembler conditional directives:

IF <*exp*>

> If <*exp*> evaluates to nonzero, the statements within the conditional block are assembled.

IFE <*exp*>

> If <*exp*> evaluates to 0, the statements in the conditional block are assembled.

IF1 Pass 1 Conditional

> If the MS-Assembler is in Pass 1, the statements in the conditional block are assembled. IF1 takes no expression.

IF2 Pass 2 Conditional

> If the MS-Assembler is in Pass 2, the statements in the conditional block are assembled. IF2 takes no expression.

IFDEF <*symbol*>

> If the <*symbol*> is defined or has been declared External, the statements in the conditional block are assembled.

IFNDEF <*symbol*>

If the <*symbol*> is not defined or not declared External, the statements in the conditional block are assembled.

IFB <*arg*>

You must enclose <*arg*> with angle brackets.

If the <*arg*> is blank (none given) or null (two angle brackets with nothing in between <>), the statements in the conditional block are assembled.

You normally use IFB (and IFNB) inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy is replaced by a parameter passed by the macro call. If the macro call does not specify a parameter to replace the dummy following IFB, the expression is blank, and the block is assembled. (IFNB is the opposite case.) For further information, see Section 4.2.3, "Macro Directives."

IFNB <*arg*>

You must enclose <*arg*> with angle brackets.

If <*arg*> is not blank, the statements in the conditional block are assembled.

You normally use IFNB (and IFB) inside macro blocks. The expression following the IFNB directive is typically a dummy symbol. When the macro is called, the dummy is replaced by a parameter passed by the macro call. If the macro call specifies a parameter to replace the dummy following IFNB, the expression is not blank, and the block is assembled. (IFB is the opposite case.) For further information, see Section 4.2.3, "Macro Directives."

IFIDN <*arg1*>,<*arg2*>

You must enclose <*arg1*> and <*arg2*> with angle brackets.

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

You normally use IFIDN (and IFDIF) inside macro blocks. The expression following the IFIDN directive is typically two dummy symbols. When the macro is called, the dummys are replaced with parameters passed by the macro call. If the macro call specifies two identical parameters to replace the dummys, the block is assembled. (IFDIF is the opposite case.) For further information, see Section 4.2.3, "Macro Directives."

IFDIF <*arg1*>,<*arg2*>

You must enclose <*arg1*> and <*arg2*> with angle brackets.

If the string <*arg1*> is different from the string <*arg2*>, the statements in the conditional block are assembled.

You normally use IFDIF (and IFIDN) inside macro blocks. The expression following the IFDIF directive is typically two dummy symbols. When the macro is called, the dummys are replaced by parameters passed by the macro call. If the macro call specifies two different parameters to replace the dummys, the block is assembled. (IFIDN is the opposite case.)

ELSE

The ELSE directive lets you generate alternate code when the opposite condition exists. You may use ELSE with any conditional directive. You may use only one ELSE for each IFxxxx conditional directive. ELSE takes no expression.

ENDIF

This directive terminates a conditional block. You must give an ENDIF directive for every IFxxxx directive used. ENDIF takes no expression. ENDIF closes the most recent, unterminated IF.

4.2.3 Macro Directives

Macro directives let you write blocks of code that can be repeated without recoding. The blocks of code begin with either the macro definition directive or one of the repetition directives, and they end with the ENDM directive. You may use all macro directives inside a macro block. The number of macros you may nest is limited only by memory.

The macro directives of the MS-Assembler include:

macro definition:
    MACRO

termination:
    ENDM
    EXITM

unique symbols within macro blocks:
    LOCAL

undefine a macro:
    PURGE

repetitions:

REPT	(repeat)
IRP	(indefinite repeat)
IRPC	(indefinite repeat character)

The macro directives also include some special macro operators:

& (ampersand)

;; (double semicolon)

! (exclamation mark)

% (percent sign)

Macro Definition

<*name*> MACRO [<*dummy*> , . . .]

```
        .
        .
        .
        ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line constitute the body of the macro, or the macro's definition.

<*name*> is like a label and conforms to the rules for forming symbols. After you define the macro, you use <*name*> to invoke the macro.

You form a <*dummy*> just as you form any other name. A <*dummy*> is a place holder that is replaced by a parameter in a one-for-one text substitution when you use the macro block. You should include all <*dummy*>s used inside the macro block on this line. The number of <*dummy*>s is limited only by the length of a line. If you specify more than one <*dummy*>, you must separate them with commas. The MS-Assembler interprets a series of <*dummy*>s the same as any list of symbol names.

NOTE

A <*dummy*> is always recognized exclusively as a dummy. Even if a register name (such as AX or BH) is used as a <*dummy*>, it is replaced by a parameter during expansion.

One alternative is to list no <*dummy*>s:

```
    <name> MACRO
```

This type of macro block lets you call the block repeatedly, even if you do not want or need to pass parameters to the block. In this case, the block contains no <*dummy*>s.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the MS-Assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable, or symbol. When the MS-Assembler evaluates a statement, it first looks at the macro table it builds during Pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: The MS-Assembler evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, the MS-Assembler assembles the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way, for example, MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

```
<name> [<parameter> , . . . ]
```

<name> is the *<name>* of the macro block. A *<parameter>* replaces a *<dummy>* on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, you must separate them with commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, the MS-Assembler passes all items inside the angle brackets as a single parameter. For example:

```
FOO 1,2,3,4,5
```

passes five parameters to the macro, but

```
FOO <1,2,3,4,5>
```

passes only one.

The number of parameters in the macro call statement need not be the same as the number of *<dummy>*s in the MACRO definition. If there are more parameters than *<dummy>*s, the extras are ignored. If there are fewer, the extra *<dummy>*s are made null. The assembled code includes the macro block after each macro call statement.

Example:

```
GEN       MACRO     XX,YY,ZZ
          MOV       AX,XX
          ADD       AX,YY
          MOV       ZZ,AX
          ENDM
```

If you then enter a macro call statement:

```
GEN       DUCK,DON,FOO
```

the MS-Assembler generates the statements:

```
          MOV       AX,DUCK
          ADD       AX,DON
          MOV       FOO,AX
```

On your program listing, these statements are preceded by a plus sign (+) to indicate that they came from a macro block.

End Macro

ENDM

ENDM tells the MS-Assembler that the MACRO or Repeat block is ended.

You terminate every MACRO, REPT, IRP, and IRPC with the ENDM directive. Otherwise, the "Unterminated REPT/IRP/IRPC/MACRO" message is generated at the end of each pass. An unmatched ENDM also causes an error.

If you wish to be able to exit a MACRO or repeat block before expansion is complete, use EXITM.

Exit Macro

EXITM

You use the EXITM directive inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. You usually use EXITM in conjunction with a conditional directive.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

Example:

```
FOO     MACRO   X
X       =       0
        REPT    X
X       =       X+1
        IFE     X-0FFH ;test X
        EXITM   ;if true, exit REPT
        ENDIF
        DB      X
        ENDM
        ENDM
```

LOCAL

LOCAL <*dummy*>[,<*dummy*>. . .]

You may use the LOCAL directive only inside a macro definition block. A LOCAL statement must precede all other types of statements in the macro definition.

When LOCAL is executed, the MS-Assembler creates a unique symbol for each <*dummy*> and substitutes that symbol for each occurrence of the <*dummy*> in the expansion. You usually use these unique symbols to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the MS-Assembler range from ??0000 to ??FFFF. Avoid using the form ??nnnn for your own symbols.

Example:

```
0000             FUN       SEGMENT
                           ASSUME CS:FUN,DS:FUN
                 FOO       MACRO        NUM,Y
                           LOCAL        A,B,C,D,E
                 A:        DB           7
                 B:        DB           8
                 C:        DB           Y
                 D:        DW           Y+1
                 E:        DW           NUM+1
                           JMP          A
                           ENDM
                           FOO          0C00H,0BEH
0000    07       + ??0000:              DB       7
0001    08       + ??0001:              DB       8
0002    BE       + ??0002:              DB       0BEH
0003    00BF     + ??0003:              DW       0BEH+1
0005    0C01     + ??0004:              DW       0C00H+1
0007    EB F7    +         JMP          ??0000
                           FOO          03C0H,0FFH
0009    07       + ??0005:              DB       7
000A    08       + ??0006:              DB       8
000B    FF       + ??0007:              DB       0FFH
000C    0100     + ??0008:              DW       0FFH+1
000E    03C1     + ??0009:              DW       03C0H+1
0010    EB F7    +         JMP          ??0005
0012               FUN     ENDS
                           END
```

Notice that the MS-Assembler has substituted LABEL names in the form ??nnnn for the instances of the dummy symbols.

PURGE

PURGE <*macro-name*>[, . . .]

PURGE deletes the definition of the macro(s) listed after it.

PURGE provides three benefits:

1. It frees text space of the macro body.
2. It returns any instruction mnemonics or directives that were redefined by macros to their original function.

3. It lets you "edit out" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method lets you use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you will not use in this program.

   It is not necessary to PURGE a macro before redefining it. Simply place another MACRO statement in your program, reusing the macro name.

Example:

```
INCLUDE MACRO.LIB
PURGE   MAC1
MAC1              ;tries to invoke purged macro
                  ;returns a syntax error
```

Repeat Directives

The directives in this group let the operations in a block of code be repeated for the number of times you specify. The major differences between the Repeat directives and MACRO directive are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed. You can use the macro block in many different programs by simply entering a macro call statement.
2. MACRO lets parameters be passed to the macro block when you call a MACRO; hence, you can change parameters.

You must assign repeat directive parameters as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be performed for every program execution, then Repeat directives are convenient. With the MACRO directive, you must call in the MACRO each time it is needed.

Note that you must match each Repeat directive with the ENDM directive to terminate the repeat block.

Repeat

```
REPT <exp>
 .
 .
 .
ENDM
```

Repeat block of statements between REPT and ENDM <*exp*> times. <*exp*> is evaluated as a 16-bit unsigned number. If <*exp*> contains an External symbol or undefined operands, an error is generated.

Example:

```
                    X       =       0
                            REPT    10        ;generates
                                              ;DB 1 - DB
10
                    X       =       X+1
                            DB      X
                            ENDM
```

assembles as:

```
0000                X       =       0
                            REPT    10        ;generates
                                              ;DB 1 - DB
10
                    X       =       X+1
                            DB      X
                            ENDM
0000'      01       +       DB      X
0001'      02       +       DB      X
0002'      03       +       DB      X
0003'      04       +       DB      X
0004'      05       +       DB      X
0005'      06       +       DB      X
0006'      07       +       DB      X
0007'      08       +       DB      X
0008'      09       +       DB      X
0009'      0A       +       DB      X
                            END
```

Indefinite Repeat

IRP <*dummy*>,<*parameters inside angle brackets*>

.
.
.

ENDM

You must enclose parameters (any legal symbol, string, numeric, or character constant) in angle brackets. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of <*dummy*> in the block. If a parameter is null (that is, <>), the block is processed once with a null parameter.

Example:

```
IRP        X,<1,2,3,4,5,6,7,8,9,10>
DB         X
ENDM
```

This example generates the same bytes (DB 1 to DB 10) as the REPT example.

When you use IRP inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

```
FOO        MACRO       X
           IRP         Y,<X>
           DB          Y
           ENDM
           ENDM
```

When the macro call statement

```
FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion becomes:

```
IRP        Y,<1,2,3,4,5,6,7,8,9,10>
DB         Y
ENDM
```

The angle brackets around the parameters are removed, and all items are passed as a single parameter.

Indefinite Repeat Character

IRPC <*dummy*>,<*string*>

.

.

.

ENDM

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <*dummy*> in the block.

Example:

```
IRPC       X,0123456789
DB         X+1
ENDM
```

This example generates the same code (DB 1 to DB 10) as the two previous examples.

Special Macro Operators

You can use several special operators in a macro block to select additional assembly functions.

& Ampersand concatenates text or symbols. (You may not use the ampersand in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by an ampersand. To form a symbol from text and a dummy, put an ampersand between them.

For example:

```
ERRGEN     MACRO      X
ERROR&X:             PUSH       BX
           MOV        BX,'&X'
           JMP        ERROR
           ENDM
```

The call ERRGEN A will then generate:

```
ERRORA:    PUSH       BX
           MOV        BX,'A'
           JMP        ERROR
```

In the MS-Assembler, the ampersand does not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, you must supply as many ampersands as there are levels of nesting.

For example:

Correct form / *Incorrect form*

```
FOO    MACRO    X                 FOO    MACRO    X
       IRP      Z,<1,2,3>                IRP      Z,<1,2,3>
X&&Z   DB       Z                 X&Z    DB       Z
       ENDM                              ENDM
       ENDM                              ENDM
```

When called, for example, by FOO BAZ, the expansion would be (correctly in the left column, incorrectly in the right):

1. MACRO build, find <*dummy*>s and change to *d1*

```
           IRP      Z,<1,2,3>             IRP      Z,<1,2,3>
d1&Z DB             Z         d1Z         DB       Z
           ENDM                        ENDM
```

2. MACRO expansion, substitute parameter text for *d1*

```
        IRP     Z,<1,2,3>          IRP      Z,<1,2,3>
BAZ&Z   DB        ZBAZZ     DB  Z
        ENDM                     ENDM
```

3. IRP build, find dummys and change to *d1*

```
BAZ&d1        DB      d1      BAZZ       DB       d1
```

4. IRP expansion, substitute parameter text for *d1*

```
BAZ1      DB     1        BAZZ      DB     1
BAZ2      DB     2        BAZZ      DB     2 <-+
BAZ3      DB     3        BAZZ      DB     3   |
                                               |
                     +-------------------------+
                     |__;here it's an error,
                        ;multi-defined symbol
```

<text>

If you enclose text with angle brackets, the MS-Assembler treats the text as a single literal. If you place parameters to a macro call inside angle brackets or place the list of parameters following the IRP directive inside angle brackets, the following occur:

1. All text within the angle brackets is seen as a single parameter, even if commas are used.

2. Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows.

One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you must supply as many sets of angle brackets around parameters as there are levels of nesting.

;; In a macro or repeat block, a comment preceded by two semicolons is not saved as a part of the expansion.

The default listing condition for macros is .XALL (see Section 4.2.4, "Listing Directives," below). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments inside macro and repeat blocks are saved and listed. This can be the cause of an "out of memory error." To avoid this error, place double semicolons before comments inside macro and repeat blocks, unless you specifically want a comment to be retained.

! You may enter an exclamation point in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The only time you use a percent sign in a macro argument is to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference, with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must evaluate to an absolute (nonrelocatable) constant.

Example:

```
PRINTE    MACRO     MSG,N
          %OUT      * MSG,N *
          ENDM
SYM1      EQU       100
SYM2      EQU       200
          PRINTE    <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement causes the string (SYM1 + SYM2) to be substituted for the dummy N. The result is:

```
%OUT  * SYM1 + SYM2 = (SYM1 + SYM2) *
```

When you place % in front of the parameter, the MS-Assembler generates:

```
%OUT  * SYM1 + SYM2 = 300 *
```

4.2.4 Listing Directives

Listing directives perform two general functions: format control and listing control. Format control directives let the programmer insert page breaks and direct page headings. Listing directives turn on and off the listing of all or part of the assembled file.

PAGE

PAGE [<*length*>][,<*width*>]
PAGE [+]

PAGE with no arguments or with the optional [,+] argument causes the assembler to start a new output page. The MS-Assembler puts a form feed character in the listing file at the end of the page.

The PAGE directive with either the length or width arguments does not start a new listing page.

The value of <*length*>, if included, becomes the new page length (measured in lines per page) and must be in the range 10 to 255. The default page length is 50 lines per page.

The value of <*width*>, if included, becomes the new page width (measured in characters) and must be in the range 60 to 132. The default page width is 80 characters.

The plus sign (+) increments the major page number and resets the minor page number to 1. Page numbers are in the form major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Example:

```
.
.
.
PAGE +    ;increment major,set minor to 1
.
.
.
PAGE 58,60   ;page length=58 lines,
             ;width=60 characters
```

TITLE

TITLE <*text*>

TITLE specifies a title to be listed on the first line of each page. The <*text*> may be a maximum of 60 characters. If you give more than one TITLE, an error results. The MS-Assembler recognizes the first six characters of the title, if legal, as the module name, unless you give a NAME directive.

Example:

```
TITLE PROG1 — 1st Program
.
.
.
```

If you do not use the NAME directive, the module name is now PROG1—1st Program. This title text appears at the top of every page of the listing.

SUBTITLE

SUBTTL <*text*>

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The MS-Assembler truncates the <*text*> after 60 characters.

You may have any number of SUBTTLs in a program. Each time the MS-Assembler encounters SUBTTL, it replaces the <*text*> from the previous SUBTTL with the <*text*> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <*text*>.

Example:

```
SUBTTL SPECIAL I/O ROUTINE
  .
  .
  .
SUBTTL
  .
  .
  .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

%OUT

%OUT <*text*>

The screen displays the text during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

%OUT outputs on both passes. If you want only one printout, use the IF1 or IF2 directive, depending on which pass you want displayed. See Section 4.2.2, "Conditional Directives," for descriptions of the IF1 and IF2 directives.

Example:

```
%OUT *Assembly half done*
```

The MS-Assembler sends this message to the terminal screen when encountered.

```
IF1
%OUT *Pass 1 started*
ENDIF
```

```
IF2
%OUT *Pass 2 started*
ENDIF
```

.LIST
.XLIST

.LIST lists all lines with their code (the default condition).

.XLIST suppresses all listing.

If you specify a listing file following the Listing: prompt, the MS-Assembler prints a listing file with all the source statements.

When the MS-Assembler encounters .XLIST in the source file, it does not list source and object code. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing directives. Nothing is listed, even if another listing directive (other than .LIST) is encountered.

Example:

```
.
.
.
.XLIST    ;listing suspended here
.
.
.
.LIST    ;listing resumes here
```

.SFCOND

.SFCOND suppresses portions of the listing that contain conditional false expressions.

.LFCOND

.LFCOND ensures the listing of conditional expressions that evaluate false. This is the default condition.

.TFCOND

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch when the MS-Assembler is running. When /X is used, .TFCOND causes false conditionals to list. When /X is not used, .TFCOND suppresses false conditionals.

.XALL

.XALL is the default.

.XALL lists source code and object code produced by a macro, but does not list source lines that do not generate code.

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code. It does not list comments preceded by two semicolons (;;).

.SALL

.SALL suppresses listing of all text and object code produced by macros.

.CREF
.XCREF

.CREF
.XCREF [<*variable list*>]

.CREF is the default condition. .CREF remains in effect until the MS-Assembler encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until the MS-Assembler encounters .CREF. Use .XCREF to suppress the creation of cross-references in selected portions of the file. Use .CREF to restart the creation of a cross-reference file after using the .XCREF directive.

If you include one or more variables following .XCREF, these variables will not be placed in the listing or cross-reference file. All other cross-referencing, however, is not effected by a .XCREF directive with arguments. Separate the variables with commas.

Neither .CREF nor .XCREF without arguments takes effect unless you specify a cross-reference file when running the MS-Assembler. .XCREF <*variable list*> suppresses the variables from the symbol table listing regardless of the creation of a cross-reference file.

Example:

```
.XCREF CURSOR,FOO,GOO,BAZ,ZOO
      ;these variables will not be
      ;in the listing or cross-reference file
```

CREF

Chapter 5

Cross-Reference Utility

Cross-Reference Utility (CREF)

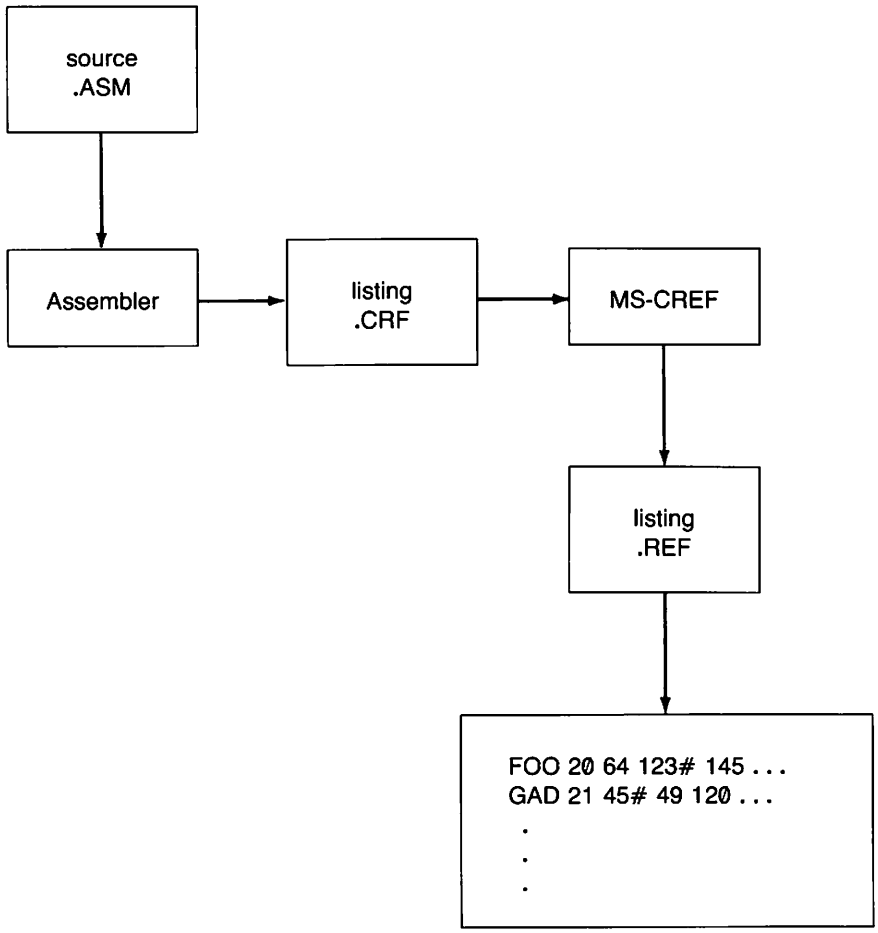
5.1 Overview of MS-CREF

The MS-CREF Cross-Reference Utility can help you in debugging your assembly language programs. With MS-CREF you can output an alphabetical listing of all the symbols to a special file created by your assembler. This listing lets you quickly locate all occurrences of any symbol in your source program by line number.

To use MS-CREF, you must first create a cross-reference file with the assembler. MS-CREF then converts this cross-reference file (which has the filename extension .CRF) into an alphabetical listing of the symbols in the file. (The cross-reference listing file is given the default filename extension .REF.)

Beside each symbol in the listing, MS-CREF lists the line numbers where the symbol occurs in the source program. A pound sign (#) indicates the line number where the symbol is defined.

Figure 7 illustrates the MS-CREF operation.



5.2 Running MS-CREF

Before you can use MS-CREF to create the cross-reference listing, you must first create a cross-reference file using your assembler. This step is described in the next section.

5.2.1 Creating a Cross-Reference File

A cross-reference file is created during an assembly session. To create a cross-reference file, use the MS- Assembler and answer the fourth command prompt with the name of the cross-reference file you want to create.

The fourth assembler prompt is:

```
    Cross-reference [NUL.CRF]:
```

If you do not type a filename in response to this prompt, or if you use the default response, the assembler will not create a cross-reference file. Therefore, you must type a filename if you want to create a cross-reference file.

You may also specify which drive or device you want the file saved on, and the filename extension (if different from .CRF).

You are now ready to use MS-CREF to convert the cross-reference file produced by the assembler into a cross-reference listing.

5.2.2 How to Start MS-CREF

MS-CREF may be started two ways. By the first method, you type the commands in response to individual prompts. By the second method, you type all commands on the line used to start MS-CREF.

Method 1: Prompts

To start MS-CREF using prompts, type:

```
    CREF
```

MS-CREF is loaded into memory. Then, MS-CREF displays two prompts.

Command Prompts

Cross reference [.CRF]:

Type the name of the cross-reference file you want MS-CREF to convert to a cross-reference listing. The filename is the name you specified when you directed the assembler to produce the cross-reference file.

MS-CREF assumes that the filename extension is .CRF. If you do not specify a filename extension when you type the cross-reference filename, MS-CREF will look for a file with the name you specify and the filename extension .CRF. If your cross-reference file has a different extension, specify that extension when you type the filename.

Refer to section 5.5, "Format of MS-CREF Compatible Files," for a description of what MS-CREF expects to see in the cross-reference file. You will need this information only if your cross-reference file was not produced by MS-Assembler.

Listing [crffile.REF]:

Type the name you want the cross-reference listing file to have. MS-CREF will automatically give the cross-reference listing the filename extension .REF.

If you want your cross-reference listing to have the same filename as the cross-reference file but with the filename extension .REF, simply press the (ENTER) key when the Listing: prompt appears. If you want your cross-reference listing file to be named anything else, or to have any other filename extension, you must type a response following the Listing: prompt.

If you want the listing file placed on a drive or device other than the default drive, specify that drive or device when you type your response to the Listing: prompt.

Method 2: Command Line

To start MS-CREF using the command line, type:

```
CREF <crffile>,<listing>
```

where <crffile> and <listing> are responses to the command prompts:

<crffile> is the name of the cross-reference file produced by your assembler. MS-CREF assumes that the filename extension is .CRF. You may override this default by specifying a different extension. If the file named for the <crffile> does not exist, MS-CREF displays the message:

```
Fatal I/O Error 110
in File: <crffile>.CRF
```

MS-CREF is aborted and the operating system prompt appears.

<listing> is the name of the file you want to contain the cross-reference listing of symbols in your program.

To select the default filename and extension for the listing file, type a semicolon after the <crffile> name. (Refer to the "Command Characters" section for more information on how to use the semicolon.)

Once you have entered the command line, MS-CREF is loaded into memory. MS-CREF then converts your cross-reference file into a cross-reference listing.

Examples:

```
CREF FUN;
```

This example causes MS-CREF to process the cross-reference file FUN.CRF and to produce a listing file named FUN.REF.

To give the listing file a different filename, extension, or destination, simply specify it when you type the command line.

```
CREF FUN,B:WORK.ARG
```

This example causes MS-CREF to process the cross-reference file named RUN.CRF and to produce a listing file named WORK.ARG, which will be placed on the disk in Drive B.

5.3 Command Characters

MS-CREF provides two command characters.

Semicolon

Use a single semicolon (;), followed immediately by a carriage return, at any time after responding to the Cross reference: prompt to select the default response to the Listing: prompt. This feature saves time and eliminates the need to answer the Listing: prompt.

If you use the semicolon, MS-CREF gives the listing file the filename of the cross-reference file and the default filename extension .REF.

Example:

```
Cross reference [ .CRF]: FUN;
```

MS-CREF will process the cross-reference file named FUN.CRF and output a listing file named FUN.REF.

CONTROL-C

Use CONTROL-C at any time to abort the MS-CREF session. If you make a mistake (for example, typing the wrong filename or incorrectly spelling a filename), you must press CONTROL-C to exit MS-CREF, and then restart MS-CREF. If you have typed the error but you have not pressed the ENTER key, you may delete the erroneous characters for that line.

5.4 Format of Cross-Reference Listings

The cross-reference listing is an alphabetical list of all the symbols in your program. Each page begins with the title of the program module. Then the symbols are listed. Following each symbol name is a list of the line numbers where the symbol occurs in your program. The line number for the definition has a pound sign (#) appended to it.

Example Of Cross-Reference Listing

```
ENTX       PASCAL entry for initializing programs comes from
                                              TITLE directive

Symbol Cross-Reference     (# is definition)           Cref-1
AAAXQQ ...........  37#     38

BEGHQQ ...........  83      84#    154    176
BEGOQQ ...........  33     162
BEGXQQ ........... 113     126#    164    223

CESXQQ ...........  97      99#    129
CLNEQQ ...........  67      68#
CODE                37     182
CONST ............ 104     104     105    110
CRCXQQ ...........  93      94#    210    215
CRDXQQ ...........  95      96#    216
CSXEQQ ...........  65      66#    149
CURHQQ ...........  85      86#    155

DATA .............  64#     64     100    110
DGROUP ........... 110#    111     111    111    127    153    171    172
DOSOFF ...........  98#    198     199
DOSXQQ ........... 184     204#    219

ENDHQQ ...........  87      88#    158
ENDOQQ ...........  33#    195
ENDUQQ ...........  31#    197
ENDXQQ ........... 184     194#
ENDYQQ ...........  32#    196
ENTGQQ ...........  30#    187
ENTXCM ........... 182#    183     221

FREXQQ ........... 169     170#    178

HDRFQQ ...........  71      72#    151
HDRVQQ ...........  73      74#    152
HEAP .............  42      44     110
HEAPBEG ..........  54#    153     172
HEAPLOW ..........  43     171

INIUQQ ...........  31     161
```

```
MAIN_START-
  UP.................. 109#    111      180
MEMORY........... 42       48#     48      49     109     110

PNUXQQ........... 69       70      150

RECEQQ........... 81       82#

REFEQQ........... 77       78#
REPEQQ........... 79       80#
RESEQQ........... 75       76#    148
ENTX        PASCAL entry for initializing programs

Symbol Cross-Reference       (# is definition)                 Cref-2

SKTOP............... 59#
SMLSTK............ 135      137#
STACK............... 53#      53       60     110
STARTMAIN ...... 163      186#    200
STKBQQ............ 89       90#    146
STKHQQ............ 91       92#    160
```

5.5 Format of MS-CREF Compatible Files

MS-CREF will process files other than those generated by MS-Assembler, as long as the file conforms to the valid MS-CREF format.

5.6 MS-CREF File Processing

MS-CREF reads a stream of bytes from the cross-reference file (or source file), sorts them, then outputs them as a printable listing file (the .REF file). The symbols are held in memory as a sorted tree. References to the symbols are held in a linked list.

MS-CREF keeps track of line numbers in the source file by the number of end-of-line characters it encounters. Therefore, every line in the source file must contain at least one end-of-line character (see the chart later in this section).

MS-CREF places a heading at the top of every page of the listing. The name MS-CREF uses is passed by your assembler from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol (see chart below). If MS-CREF encounters more than one title symbol in the source file, it will use the last title read for all page headings. If MS-CREF does not encounter a title symbol in the file, the title line on the listing will be blank.

5.7 Format of Source Files

MS-CREF uses the first three bytes of the source file as format specification data. The rest of the file is processed as a series of records that either begin or end with a byte that identifies the type of record.

First Three Bytes

The PAGE directive in your assembler, which takes arguments for page length and line length, will pass the following information to the cross-reference file:

First Byte
The number of lines to be printed per page (page length range is from 1 to 255 lines).

Second Byte
The number of characters per line (line length range is from 1 to 132 characters).

Third Byte
The Page Symbol (07) that tells MS-CREF that the two preceding bytes define listing page size.

If MS-CREF does not find these first three bytes in the file, it uses default values for page size (page length is 58 lines; line length is 80 characters).

Control Symbols

The two tables below show the types of records that MS-CREF recognizes and the byte values and placement it uses to recognize record types.

Records have a control symbol (which identifies the record type) either as the first byte of the record or as the last byte.

Records That Begin with a Control Symbol

Byte Value*	Control Symbol	Subsequent Bytes
01	Reference symbol	Record is a reference to a symbol name (1 to 80 characters)
02	Define symbol	Record is a definition of a symbol name (1 to 80 characters)
04	End-of-line	(none)
05	End-of-file	1AH

Records That End with a Control Symbol

Byte Value*	Control Symbol	Preceding Bytes
06	Title defined	Record is title text (1 to 80 characters)
07	Page length/ line length	One byte for page length followed by one byte for line length

*For all record types, the byte value represents a control character, as follows:

01	Control-A
02	Control-B
04	Control-D
05	Control-E
06	Control-F
07	Control-G

The Control Symbols are defined as follows:

Reference symbol
: Record contains the name of a symbol that is referenced. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

Define symbol
: Record contains the name of a symbol that is defined. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

End-of-line
: Record is an end-of-line symbol character only (04H or Control-D).

End-of-file
: Record is the end-of-file character (1AH).

Title defined
: ASCII characters of the title are to be printed at the top of each listing page. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used for the title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the listing will be left blank.

Page length/line length

The first byte of the record contains the number of lines to be printed per page (range is from 1 to 255 lines). The second byte contains the number of characters to be printed per page (range is from 1 to 132 characters). The default page length is 58 lines. The default line length is 80 characters.

The following table illustrates CRF file record contents by byte and length of record.

Summary of CRF File Record Contents

Byte Contents	Length of Record
01 symbol_name	2-81 bytes
02 symbol_name	2-81 bytes
04	1 byte
05 1A	2 bytes
title_text 06	2-81 bytes
PL LL 07	3 bytes

Instruction Sets

Chapter 6

8086/8088 and 8087 Instructions

8086/8088 and 8087 Instructions and Support

MS-Assembler supports standard Intel 8086/8088 and 8087 instructions and operands. This section contains these instructions and operands. An alphabetical list of the instructions and opcodes can be found in Appendix C of this manual.

There are two switches that are used when running MS-Assembler with an 8087. These switches are /R (for Real) and /E (for Emulate). The /R and /E switches are described below.

Switch	Function
/R	Use the /R switch when the code being produced by MS-Assembler is going to be run on a *real* 8087 machine (not an emulated machine). Code produced with the /R switch will only run on real 8087 machines.
/E	Use the /E switch when the code being produced by MS-Assembler is going to be run on an *emulated* 8087 machine. Code produced with the /E switch will also run on real 8087 machines with the appropriate emulator library.

Be sure to use /E when using Model 2000.

The emulator library is provided with some MS-DOS language products. It contains specific 8087 emulation routines. Refer to your language compiler user's guide for information on the emulator library that has been provided. If your code is going to run on an *emulated* 8087 machine, you must specify the appropriate emulator library when you link your code with MS-LINK. If the library is not specified, MS-LINK will return errors for those unresolved symbols that are defined in the emulator library.

The 8086/8088 Instruction Set

Instruction Statement Formats

The format for the instruction statement was introduced in Chapter 4. The format is shown below:

```
[label:] [prefix] mnemonic [operand [, operand]]
```

This chapter describes the 8086/8087/8088 instruction set. The instruction set consists of a set of mnemonics that select different machine operations. The instruction set encyclopedia at the end of this chapter describes each of these mnemonics, their operations, and allowed operands.

Addressing Modes

The 8086 instruction set provides several different ways to address operands. Most two-operand instructions allow either memory or a register to serve as one operand, and either a register or a constant within the instruction to serve as the other operand. Memory to memory operations are excluded.

Operands in memory may be addressed *directly* with a 16-bit offset address, or *indirectly* with *base* (BX or BP) and/or *index* (SI or DI) registers added to an optional 8- or 16-bit displacement *constant*. This constant can be the name of a variable or a pure number. When a name is used, the displacement constant is the variable's offset (see Chapter 4).

The result of a two-operand operation may be directed to either memory or a register. Single-operand operations are applicable uniformly to any operand except immediate constants. Virtually all 8086 operations may specify either 8- or 16-bit operands.

Memory Operands

Operands residing in memory may be addressed in four ways:

- Direct 16-bit offset address
- Indirect through a base register, BX or BP, optionally with an 8- or 16-bit displacement
- Indirect through an index register, SI or DI, optionally with an 8- or 16-bit displacement
- Indirect through the sum of one base register and one index register, optionally with an 8- or 16-bit displacement.

The location of an operand in an 8086 register or in memory is specified by up to three fields in each instruction. These fields are the mode field (mod), the register field (reg), and the register/memory field (r/m). When used, they occupy the second byte of the instruction sequence. This byte is referred to as the Modrm byte of the instruction.

The mode field occupies the two most significant bits, 7 and 6, of the byte, and specifies how the r/m field (bits 2, 1, 0) is used in locating the operand. The r/m field can name a register that holds the operand or can specify an addressing mode (in combination with the mod field) that points to the location of the operand in memory. The reg field occupies bits 5, 4, and 3 following the mode field, and can specify that one operand is either an 8-bit register or a 16-bit register. In some instructions, this reg field gives additional bits of information specifying the instruction, rather than only encoding a register.

Description

The effective address (EA) of the memory operand is computed according to the *mod* and *r/m* fields:

```
if mod  = 00 then DISP = 0*, disp-low and disp-high are absent
if mod  = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod  = 10 then DISP = disp-high:disp-low
if r/m  = 000 then EA = (BX) + (SI) + DISP
if r/m  = 001 then EA = (BX) + (DI) + DISP
if r/m  = 010 then EA = (BP) + (SI) + DISP
if r/m  = 011 then EA = (BP) + (DI) + DISP
if r/m  = 100 then EA = (SI) + DISP
if r/m  = 101 then EA = (DI) + DISP
if r/m  = 110 then EA = (BP) + DISP*
if r/m  = 111 then EA = (BX) + DISP

*except if mod = 00 and r/m = 110 then
 EA = disp-high: disp-low
```

Instructions referencing 16-bit objects interpret EA as addressing the low-order byte; the word is addressed by EA+1,EA.

Encoding

mod reg r/m	disp-low	disp-high

Segment Override Prefixes

General register BX and pointer register BP may serve as base registers. When BX is the base the operand by default resides in the current Data Segment and the DS register is used to compute the physical address of the operand. When BP is the base, the operand by default resides in the current Stack Segment and the SS segment register is used to compute the physical address of the operand. When both base and index registers are used, the operand by default resides in the segment determined by the base register, i.e., BX means DS is used, BP means SS is used. When an index register alone is used, the operand by default resides in the current Data Segment. The physical address of most other memory operands is by default computed using the DS segment register (exceptions are noted below). These assembler-default segment register selections may be overridden by preceding the referencing instruction with a segment override prefix.

Description

The segment register selected by the *reg* field of a segment prefix is used to compute the physical address for the instruction this prefix precedes. This prefix may be combined with the LOCK and/or REP prefixes, although the latter has certain requirements and consequences—see REP.

Encoding

`001 reg 110`

reg is assigned according to the following table:

	Segment
00	ES
01	CS
10	SS
11	DS

Exceptions

The physical addresses of all operands addressed by the SP register are computed using the SS segment register, which may not be overridden. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden.

Register Operands

The four 16-bit general registers and the four 16-bit pointer and index registers may serve interchangeably as operands in nearly all 16-bit operations. Three exceptions to note are multiply, divide, and some string operations, which use the AX register implicitly. The eight 8-bit registers of the HL group may serve interchangeably in 8-bit operations. Multiply, divide, and some string operations use AL implicitly.

Description

Register operands may be indicated by a distinguished field, in which case REG will represent the selected register, or by an encoded field, in which case EA will represent the register selected by the r/m field. Instructions without a "w" bit always refer to 16-bit registers (if they refer to any register at all); those with a "w" bit refer to either 8- or 16-bit registers according to "w".

Encoding

General Registers:

Distinguished Field:

`     reg` or `  reg  `

for mode = 11 EA = r/m (a register):

`11 reg`

REG is assigned according to the following table:

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

Instructions that reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

```
FLAGS  X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)
```

where X is undefined.

Immediate Operands

All two-operand operations except multiply, divide, and the string operations allow one source operand to appear within the instruction as immediate data. Sixteen-bit immediate operands that have a high-order byte that is the sign extension of the low-order byte may be abbreviated to eight bits.

Three points about immediate operands:

- Immediate operands always *follow* addressing mode displacement constants (when present) in the instruction.
- The low-order byte of 16-bit immediate operands always precedes the high-order byte.
- The 8-bit immediate operands of instructions with s:w = 11 are sign-extended to 16-bit values.

String Instructions and Memory References

Table 6-1 shows the mnemonics of the string instructions that can be coded without operands (MOVSB, MOVSW, etc.) or with operands (MOVS, etc.).

The string instructions are unusual in several respects:

1. Before coding a string instruction, you must:
   - Load SI with the offset of the source string.
   - Load DI with the offset of the destination string.
2. One of the forms of REP (REP, REPZ, REPE, REPNE, REPNZ) can be coded immediately preceding (but separated from by at least one blank) the primitive string operation mnemonic (thus, REPNZ SCASW is one possibility). This specifies that the string operation is to be repeated the number of times determined by CX. (Refer to instruction descriptions.)
3. Each can be coded with or without symbolic memory operands.
   - If symbolic operands are coded, the assembler can check the addressability of the operands.

Table 6-1. String Instruction Mnemonics

Operation Being Performed	Mnemonic if Operand Is Byte String	Mnemonic if Operand Is Word String	Mnemonic if Symbolic Operands Are Coded*
Move	MOVSB	MOVSW	MOVS
Compare	CMPSB	CMPSW	CMPS
Load AL/AX	LODSB	LODSW	LODS
Store from AL/AX	STOSB	STOSW	STOS
Compare to AL/AX	SCASB	SCASW	SCAS
Block Input	INSB	INSW	INS
Block Output	OUTSB	OUTSW	OUTS

*If symbolic operands are coded, the assembler can check their addressability. Also, their TYPEs determine the opcode generated.

- Anonymous references that use the hardware defaults should be coded using the operand-less forms (e.g. MOVSB, MOVSW), to avoid the cumbersome (but otherwise required):

  ```
  MOVS ES:BYTE PTR [DI], [SI]
  ```

  as opposed to the simple:

  ```
  MOVSB
  ```

- Anonymous references that do not use the hardware defaults require both segment and type to be explicitly specified:

  ```
  MOVS ES:BYTE PTR [DI], SS:[SI]
  ```

- Never use [BX] or [BP] addressing modes with string instructions.

4. If the instruction mnemonic is coded without operands (e.g., MOVSB, MOVSW), then the segment register defaults are as follows:
   - SI defaults to an offset in the segment addressed by DS,
   - DI is required to be an offset in the segment addressed by ES.

   Thus, the direction of data flow for the default case in which no operands are specified is from the segment addressed by DS to the segment addressed by ES.
5. If the instruction mnemonic is coded with operands (e.g. MOVS, CMPS), the operands can be anonymous (indirect) or they can be variable references.

Example:

```
DESTSTRING  EQU ES:BYTE PTR [DI]

SRCSTRING   EQU  DS:BYTE PTR [SI]

ASSUME CS:CODE, DS:DATA, ES:DATA1

DATA SEGMENT

SRCARRAY  DB  10 DUP (1)

DATA ENDS

DATA1 SEGMENT

DESTARRAY  DB  10 DUP (?)

DATA1 ENDS

CODE SEGMENT
        MOV  AX, DATA
        MOV  DS, AX       ;INIT DS
        MOV  AX, DATA1
        MOV  ES, AX       ;INIT ES

        MOV  SI, OFFSET SRCARRAY
        MOV  DI, OFFSET DESTARRAY

                          ;INIT POINTER REGISTERS

        MOV  CX, 10       ;NUMBER OF ELEMENTS
        REP MOVS DESTSTRING, SRCSTRING
```

```
                               ;PERFORM STRING OPERATION
          .
          .
          .
CODE ENDS
```

Mnemonic Synonyms

There are some machine operations that can have different mnemonics. The different mnemonics are all synonyms in that they refer to the same machine instructions. They are supplied by the assembler to allow you to think of the operation in terms that are more helpful for your task. Many of the conditional jump instructions have more than one mnemonic. When used after a compare, the conditional jump mnemonic can express the type of compare or the result of the compare in terms of the flags that were set. For example,

```
CMP  DEST, SRC
JE   LAB1          ;jump if dest is equal to source
```

or

```
CMP  DEST, SRC
JZ   LAB1          ;jump if zero flag set (dest = src)
```

In both cases, the same instruction will be encoded for the jump. Programmers familiar with other assembly languages that use conditional jump mnemonics that refer to flags may be more comfortable using this form. However, the first form that expresses the relationship the compare is checking between the operands is more expressive.

Organization of the Instruction Set

Instructions are described in this section in six functional groups:

- Data transfer
- Arithmetic
- Logic
- String manipulation
- Control transfer
- Processor control

Each of the first three groups mentioned in the preceding list is further subdivided into an array of codes that specify whether the instruction is to act upon immediate data, register or memory locations, whether 16-bit words, or 8-bit bytes are to be processed, and what addressing mode is to be employed. All of these codes are listed and explained in detail, but you do not have to code each one individually. The context of your program automatically causes the assembler to generate the correct code. There are three general categories of instructions within each of the three functional groups mentioned:

1. Register or memory space to or from register
2. Immediate data to register or memory
3. Accumulator to or from registers, memory, or ports

Data Transfer

Data transfer operations are divided into four classes:

1 general purpose
2 accumulator-specific
3 address-object
4 flag

None affect flag settings except SAHF and POPF.

General Purpose Transfers

Four general purpose data transfer operations are provided. These may be applied to most operands, though there are specific exceptions. The general purpose transfers (except XCHG) are the only operations that allow a segment register as an operand.

— MOV performs a byte or word transfer from the source (rightmost) operand to the destination (leftmost) operand.

— PUSH decrements the SP register by two and then transfers a word from the source operand to the stack element currently addressed by SP.

— POP transfers a word operand from the stack element addressed by the SP register to the destination operand and then increments SP by 2.

— XCHG exchanges the byte or word source operand with the destination operand. The segment registers may not be operands of XCHG.

Accumulator-Specific Transfers

Three accumulator-specific transfer operations are provided:

— IN transfers a byte (or word) from an input port to the AL register (or AX register). The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K input ports.

— OUT is similar to IN except that the transfer is from the accumulator to the output port.

— XLAT performs a table lookup byte translation. The AL register is used as an index into a 256-byte table addressed by the BX register. The byte operand so selected is transferred to AL.

Address-Object Transfers

Three address-object transfer operations are provided:

— LEA (load effective address) transfers the offset address of the source operand to the destination operand. The source operand must be a memory operand and the destination operand must be a 16-bit general, pointer, or index register.

— LDS (load pointer into DS) transfers a "pointer-object" (i.e., a 32-bit object containing an offset address and a segment address) from the source operand (which must be a doubleword memory operand) to a pair of destination registers. The segment address is transferred to the DS segment register. The offset address is transferred to the 16-bit general, pointer, or index register that you coded.

— LES (load pointer into ES) is similar to LDS except that the segment address is transferred to the ES segment register.

Flag Register Transfers

Four flag register transfer operations are provided:

— LAHF (load AH with flags) transfers the flag registers SF, ZF, AF, PF, and CF (the 8080 flags) into specific bits of the AH register.

— SAHF (store AH into flags) transfers specific bits of the AH register to the flag registers, SF, ZF, AF, PF, and CF.

— PUSHF (push flags) decrements the SP register by two and transfers all of the flag registers into specific bits of the stack element addressed by SP.

— POPF (pop flags) transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by two.

Arithmetic

The 8086/8088 provides the four basic mathematical operations in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard twos complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer). Correction operations are provided to allow arithmetic to be performed directly on unpacked decimal digits or on packed decimal representations.

Flag Register Settings

Six flag registers are set or cleared by arithmetic operations to reflect certain properties of the result of the operation. They generally follow these rules (see also Appendix C):

— CF is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise, CF is cleared.

— AF is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise, AF is cleared.

— ZF is set if the result of the operation is zero; otherwise, ZF is cleared.

— SF is set if the high-order bit of the result of the operation is set; otherwise, SF is cleared.

— PF is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise, PF is cleared (odd parity).

— OF is set if the operation results in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa; otherwise, OF is cleared.

Addition

Five addition operations are provided:

— ADD performs an addition of the source and destination operands and returns the result to the destination operand.

— ADC (add with carry) performs an addition of the source and destination operands, adds one if the CF flag is found previously set, and returns the result to the destination operand.

— INC (increment) performs an addition of the source operand and one, and returns the result to the operand.

— AAA (unpacked BCD (ASCII) adjust for addition) performs a correction of the result in AL of adding two unpacked decimal operands, yielding an unpacked decimal sum.

— DAA (decimal adjust for addition) performs a correction of the result in AL of adding two packed decimal operands, yielding a packed decimal sum.

Subtraction

Seven subtraction operations are provided:

— SUB performs a subtraction of the source from the destination operand and returns the result to the destination operand.

— SBB (subtract with borrow) performs a subtraction of the source from the destination operand, subtracts one if the CF flag is found previously set, and returns the result to the destination operand.

— DEC (decrement) performs a subtraction of one from the source operand and returns the result to the operand.

— NEG (negate) performs a subtraction of the source operand from zero and returns the result to the operand.

— CMP (compare) performs a subtraction of the source destination operand, causing the flags to be affected, but does not return the result.

— AAS (unpacked BCD (ASCII) adjust for subtraction) performs a correction of the result in AL of subtracting two unpacked decimal operands, yielding an unpacked decimal difference.

— DAS (decimal adjust for subtraction) performs a correction of the result in AL of subtracting two packed decimal operands, yielding a packed decimal difference.

Multiplication

Three multiplication operations are provided:

— MUL performs an unsigned multiplication of the accumulator (AL or AX) and the source operand, returning a double length result to the accumulator and its extension (AL and AH for 8-bit operation, AX and DX for 16-bit operation). CF and OF are set if the top half of the result is non-zero.

— IMUL (integer multiply) is similar to MUL except that it performs a signed multiplication. CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

— AAM (unpacked BCD (ASCII) adjust for multiply) performs a correction of the result in AX of multiplying two unpacked decimal operands, yielding an unpacked decimal product.

Division

There are three division operations provided and two sign-extension operations to support signed division:

— DIV performs an unsigned division of the accumulator and its extension (AL and AH for 8-bit operation, AX and DX for 16-bit operation) by the source operand and returns the single length quotient to the accumulator (AL or AX), and returns the single length remainder to the accumulator extension (AH or DX). The flags are undefined. Division by zero generates an interrupt of type 0.

— IDIV (integer division) is similar to DIV except that it performs a signed division.

— AAD (unpacked BCD (ASCII) adjust for division) performs a correction of the dividend in AL before dividing two unpacked decimal operands, so that the result will yield an unpacked decimal quotient.

— CBW (convert byte to word) performs a sign extension of AL into AH.

— CWD (convert word to double word) performs a sign extension of AX into DX.

Logic

The 8086/8088 provides the basic logic operations for both 8- and 16-bit operands.

Single-Operand Operations. Three-single-operand logical operations are provided:

— NOT forms the one's complement of the source operand and returns the result to the operand. Flags are not affected.

— Shift operations of four varieties are provided for memory and register operands: SHL (shift logical left), SHR (shift logical right), SAL (shift arithmetic left), and SAR (shift arithmetic right). Single bit shifts, and variable bit shifts with the shift count taken from the CL register are available. The CF flag becomes the last bit shifted out, OF is defined only for shifts with count of 1, and is set if the final sign bit value differs from the previous value of the sign bit, and PF, SF, and ZF are set to reflect the resulting value.

— Rotate operations of four varieties are provided for memory and register operands: ROL (rotate left), ROR (rotate right), RCL (rotate through CF left), and RCR (rotate through CF right). Single bit rotates, and variable bit rotates with the rotate count taken from the CL register, are available. The CF flag becomes the last bit rotated cut; OF is defined only for shifts with count of 1, and is set if the final sign bit value differs from the previous value of the sign bit.

Two-Operand Operations

Four two-operand logical operations are provided. The CF and OF flags are cleared on all operations; SF, PF, and ZF reflect the result.

— AND performs the bitwise logical conjunction of the source and destination operand and returns the result to the destination operand.

— TEST performs the same operations as AND, causing the flags to be affected but does not return the result.

— OR performs the bitwise logical inclusive disjunction of the source and destination operand and returns the result to the destination operand.

— XOR performs the bitwise logical exclusive disjunction of the source and destination operand and returns the result to the destination operand.

String Manipulation

One-byte instructions perform various primitive operations for the manipulation of byte and word strings (sequences of bytes or words). Any primitive operation can be performed repeatedly in hardware by preceding its instruction with a repeat prefix (see REP). The single-operation forms may be combined to form complex string operations with repetition provided by iteration operations.

Hardware Operation Control

All primitive string operations use the SI register to address the source operands. The DI register is used to address the destination operands that reside in the current extra segment. If the DF flag is cleared, the operand pointers are incremented after each operation, once for byte operations and twice for word operations. If the DF flag is set, the operand pointers are decremented after each operation. See Processor Control for setting and clearing DF.

Any of the primitive string operation instructions may be preceded with a one-byte prefix indicating that the operation is to be repeated until the operation count in CX is satisfied. The test for completion is made prior to each repetition of the operation. Thus, an initial operation count of zero in CX will cause zero executions of the primitive operation.

The repeat prefix byte also designates a value to compare with the ZF flag. If the primitive operation is one that affects the ZF flag, and the ZF flag is unequal to the designated value after any execution of the primitive operation, the repetition is terminated. This permits the scan operation, for example, to serve as a scan-while or a scan-until.

During the execution of a repeated primitive operation, the operand index registers (SI and DI) and the operation count register (CX) are updated after each repetition, whereas the instruction pointer will retain the offset address of the repeat prefix byte (assuming it immediately precedes the string operation instruction). Thus, an interrupted repeated operation will be correctly resumed when control returns from the interrupting task.

Using more than one prefix on an instruction is processor dependent. Please refer to the User's Manual for your processor for further information.

Primitive String Operation

Five primitive string operations are provided:

— MOVS (MOVSB, MOVSW) transfers a byte (or word) operand from the source (rightmost) operand to the destination (leftmost) operand. As a repeated operation, this provides for moving a string from one location in memory to another.

— CMPS (CMPSB, CMPSW) subtracts the rightmost byte (or word) operand from the leftmost operand and affects the flags but does not return the result. As a repeated operation, this provides for comparing two strings. With the appropriate repeat prefix it is possible to determine after which string element the two strings become unequal, thereby establishing an ordering between the strings.

— SCAS (SCASB, SCASW) subtracts the destination byte (or word) operand from AL (or AX) and affects the flags but does not return the result. As a repeated operation, this provides for scanning for the occurrence of, or departure from, a given value in the string.

— LODS (LODSB, LODSW) transfers a byte (or word) operand from the source operand to AL (or AX). This operation ordinarily would not be repeated.

— STOS (STOSD, STOSW) transfers a byte (or word) operand from AL (or AX) to the destination operand. As a repeated operation, this provides for filling a string with a given value.

In all the cases above, the source operand is addressed by SI and the destination operand is addressed by DI. Only in CMPB/CMPW does the DI-indexed operand appear as the rightmost operand.

Software Operation Control

The repeat prefix provides for rapid iteration in a hardware-repeated string operation. The iteration control operations (see LOOP) provide this same control for implementing software loops to perform complex string operations. These iteration operations provide the same operation count update, operation completion test, and ZF flag tests that the repeat prefix provides.

By combining the primitive string operations and iteration control operations with other operations, it is possible to build sophisticated yet efficient string manipulation routines. One instruction that is particularly useful in this context is XLAT. It permits a byte fetched from one string to be translated before being stored in a second string, or before being operated upon in some other fashion. The translation is performed by using the value in the AL register as an index into a table pointed at by the BX register. The translated value obtained from the table then replaces the value initially in the AL register (see XLAT).

Control Transfer

Four classes of control transfer operations may be distinguished: calls, jumps, and returns; conditional transfers; iteration control; and interrupts.

All control transfer operations cause the program execution to continue at some new location in memory, possibly in a new code segment. Conditional transfers are provided for targets in the range −128 to +127 bytes from the transfer.

Calls, Jumps, and Returns

Two basic varieties of calls, jumps, and returns are provided—those that transfer control within the current code segment, and those that transfer control to an arbitrary code segment, which then becomes the current code segment. Both direct and indirect transfers are supported; indirect transfers make use of the standard addressing modes as described in above.

The three transfer operations are described below.

— CALL pushes the offset address of the next instruction onto the stack (in the case of an inter-segment transfer the CS segment register is pushed first) and then transfers control to the target operand.

— JMP transfers control to the target operand.

— RET transfers control to the return address saved by a previous CALL operation, and optionally may adjust the SP register so as to discard stacked parameters.

Intra-segment direct calls and jumps specify a self-relative direct displacement, thus allowing *position independent code*. A shortened jump instruction is available for transfers in the range −128 to +127 bytes from the instruction for code compaction.

Conditional Jumps

The conditional transfers of control perform a jump contingent upon various Boolean functions of the flag registers. The destination must be within a −128 to +127 byte range of the instruction. Table 6-2 shows the available instructions, the conditions associated with them, and their interpretation.

Table 6-2. 8086/8087 Conditional Transfer Operations

Instruction	Condition	Interpretation
JE or JZ	ZF = 1	"equal" or "zero"
JL or JNGE	(SF xor OF) = 1	"less" or "not greater or equal"
JLE or JNG	((SF xor OF) or ZF) = 1	"less or equal" or "not greater"
JB or JNAE or JC	CF = 1	"below" or "not above or equal" or "carry"
JBE or JNA	(CF or ZF) = 1	"below or equal" or "not above"
JP or JPE	PF = 1	"parity" or "parity even"
JO	OF = 1	"overflow"
JS	SF = 1	"sign"
JNE or JNZ	ZF = 0	"not equal" or "not zero"
JNL or JGE	(SF xor OF) = 0	"not less" or "greater or equal"
JNLE or JG	((SF xor OF) or ZF) = 0	"not less or equal" or "greater"
JNB or JAE or JNC	CF = 0	"not below" or "above or equal" or "no carry"
JNBE or JA	(CF or ZF) = 0	"not below or equal" or "above"
JNP or JPO	PF = 0	"not parity" or "parity odd"
JNO	OF = 0	"not overflow"
JNS	SF = 0	"not sign"

*"Above" and "below" refer to the relation between two unsigned values, while "greater" and "less" refer to the relation between two signed values.

Iteration Control

The iteration control transfer operations perform leading- and trailing-decision loop control. The destination of iteration control transfers must be within a −128 to +127 byte range of the instruction. These operations are particularly useful in conjunction with the string manipulation operations.

There are four iteration control transfer operations provided:

— LOOP decrements the CX ("count") register by one and transfers if CX is not zero.

— LOOPZ (also called LOOPE) decrements the CX register by one and transfers if CX is not zero and the ZF flag is set (loop while zero or loop while equal).

— LOOPNZ (also called LOOPNE) decrements the CX register by one and transfers if CX is not zero and the ZF flag is cleared (loop while not zero or loop while not equal).

— JCXZ transfers if the CX register is zero.

Interrupts

Program execution control may be transferred by means of operations similar in effect to that of external interrupts. All interrupts perform a transfer by pushing the flag registers onto the stack (as in PUSHF), and then performing an indirect inter-segment call through an element of an interrupt transfer vector located at absolute locations 0 through 3FFH. This vector contains a four-byte element for each of up to 256 different interrupt types.

Three interrupt transfer operations provided.

— INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. A one-byte form of this instruction is available for interrupt type 3.

— INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 if the OF flag is set (trap on overflow). If the OF flag is cleared, no operation takes place.

— IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

Processor Control

Various instructions and mechanisms are provided for control and operation of the processor and its interaction with its environment.

Flag Operations

There are seven operations provided that operate directly on individual flag registers.

— CLC clears the CF flag.

— CMC complements the CF flag.

— STC sets the CF flag.

— CLD clears the DF flag, causing the string operations to auto-increment the operand pointers.

— STD sets the DF flag, causing the string operations to auto-decrement the operand pointers.

— CLI clears the IF flag, disabling external interrupts (except for the non-maskable external interrupt).

— STI sets the IF flag, enabling external interrupts after the execution of the next instruction.

Processor Halt

The HLT instruction causes the 8086 processor to enter its halt state. The halt state is cleared by an enabled external interrupt or RESET.

Processor Wait

The WAIT instruction causes the processor to enter a wait state if the signal on its TEST pin is not asserted. The wait state may be interrupted by an enabled external interrupt. When this occurs the saved code location is that of the WAIT instruction, so that upon return from the interrupting task, the wait state is re-entered. The wait state is cleared and execution resumed when the TEST signal is asserted. Execution resumes without allowing external interrupts until after the execution of the next instruction. This instruction allows the processor to synchronize itself with external hardware.

Processor Escape

The ESC instruction provides a mechanism by which other processors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 processor does no operation for the ESC instruction other than to access a memory operand.

Bus Lock

A special one-byte prefix may precede any instruction causing the processor to assert its bus-lock signal for the duration of the operation caused by that instruction. This has use in multiprocessing applications (see LOCK).

Single Step

When the TF flag register is set, the processor generates a type 1 interrupt after the execution of each instruction. During interrupt transfer sequences caused by any type of interrupt, the TF flag is cleared after the push-flags step of the interrupt sequence. No instructions are provided for setting or clearing TF directly. Rather, the flag register image saved on the stack by a previous interrupt operation must be modified, so that the subsequent interrupt return operation (IRET) restores TF set. This allows a diagnostic task to single-step through a task under test, while still executing normally itself.

If the single-stepped instruction itself clears the TF flag, the type 1 interrupt will still occur upon completion of the single-stepped instruction. If the single-stepped instruction generates an interrupt or if an enabled external interrupt occurs prior to the completion of the single-stepped instruction, the type 1 interrupt sequence will occur after the interrupt sequence of the generated or external interrupt, but before the first instruction of the interrupt service routine is executed.

The 8086/8088 hardware protects the execution of the instruction immediately following a POP or a MOV to a segment register instruction from any kind of interrupt, including type 1 interrupts used to single-step. When single-stepping through a task under test, the single-step interrupt is not recognized until the instruction following the POP or MOV to a segment register instruction is executed.

Example

```
TEST_TASK     SEGMENT
              ASSUME     CS:TEST_TASK
INSTRUC1:     POP        DS
INSTRUC2:     POP        BX
INSTRUC3:     ADD        AX,[BX]
      .
      .
      .
TEST_TASK     ENDS
```

When single-stepping through TEST_TASK, INSTRUC1 steps to INSTRUC3 since the single-step interrupt is not recognized by the 8086/8088 until the instruction following the POP to the DS segment register (POP BX) is executed.

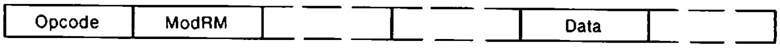
Instruction Description Formats

The formats presented in the individual instruction descriptions and briefly discussed here reflect the assembly language processed by the 8086/8087/8088 Macro Assembler (ASM86).

Format Boxes

The individual instruction descriptions show first a format box such as the following example.

Mem/Reg * Immediate to Reg



These are byte-wise representations of the object code generated by the assembler and are interpreted as follows:

- Opcode is the 8-bit opcode for the instruction. The actual opcode generated is defined in the "Opcode" column of the instruction table that follows each format box.
- ModRM is the byte that specifies the operands of the instruction. It contains a 2-bit mode field (MOD), a 3-bit register field (REG), and a 3-bit Register or Memory (R/M) field.
- Dashed blank boxes following the ModRM box are for any displacement required by the mode field.
- Data is for a byte of immediate data.
- A dashed blank box following a Data box is used whenever the immediate operand is a word quantity.

Instruction Detail Tables

Following each format box, an instruction detail table shows the opcode, the number of clocks required for the operation to take place, the actual operation performed, and a coding example for each variant of the instruction.

The instruction detail table for the instruction IMUL is shown below. The examples in the table are neither complete nor restrictive; anyplace there is a memory operand, any of the seven memory addressing modes can be used.

Opcode	Clocks	Operation	Coding Example
F6	80-98	AX ← AL · Reg 8	IMUL BL
F6	(86-104) + EA	AX ← AL · Mem 8	IMUL BYTESOMETHING
F7	128-154	DX:AX ← AX · Reg 16	IMUL BX
F7	(134-160) + EA	DX:AX ← AX · Mem 16	IMUL WORDSOMETHING

Flags

The flags produced by each instruction are represented by a table such as the following:

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

The top line in the table represents the individual flags, and the lower line shows the effect on each flag by the instruction. The letters, numbers and symbols used in the table are defined as follows:

Flag	Definition
O	Overflow
D	Direction (used in string ops)
I	Interrupt Enable (1=enabled)
T	Single Step Trap Flag (causes interrupt 1 after next instruction)
S	Sign
Z	Zero
A	Auxiliary Carry (used primarily in BCD ops)
P	Parity
C	Carry

Effect Code	Effect
X	Modified by the instruction; result depends on operands.
-	Not modified.
U	Undefined after the instruction.
1	Set to 1 by the instruction.
0	Set to 0 by the instruction.

Table 6-3. Symbols

8086/8088 Descriptor	Meaning
AX	Accumulator (16-bit)
AH	Accumulator (high-order byte)
AL	Accumulator (low-order byte)
BX	Register BX (16-bit), which may be split and addressed as two 8-bit registers.
BH	High-order byte of register BX.
BL	Low-order byte of register BX.
CX	Register CX (16-bit), which may be split and addressed as two 8-bit registers.
CH	High-order byte of register CX.
CL	Low-order byte of register CX.
DX	Register DX (16-bit), which may be split and addressed as two 8-bit registers.
DH	High-order byte of register DX.
DL	Low-order byte of register DX.
SP	Stack pointer (16-bit)
BP	Base pointer (16-bit)
IP	Instruction Pointer (16-bit)
Flags	16-bit register space, in which nine flags reside.
DI	Destination Index register (16-bit)
SI	Stack Index register (16-bit)
CS	Code Segment register (16-bit)
DS	Data Segment register (16-bit)
ES	Extra Segment register (16-bit)
SS	Stack Segment register (16-bit)

Table 6-3. Symbols (Cont'd.)

8086/8088 Descriptor	Meaning
REG8	The name or encoding of an 8-bit CPU register location.
REG16	The name or encoding of an 16-bit CPU register location.
LSRC, RSRC	Refer to operands of an instruction, generally left source and right source when two operands are used. The leftmost operand is also called the destination operand, and the rightmost is called the source operand.
reg	A field that specifies REG8 or REG16 in the description of an instruction.
EA	Effective address (16-bit)
r/m	Bits 2, 1, and 0 of the MODRM byte used in accessing memory operands. This 3-bit field defines EA, in conjunction with the mode and w fields.
mode	Bits 7 and 6 of the MODRM byte. This 2-bit field defines the addressing mode.
w	A 1-bit field in an instruction, identifying byte instructions (w=0), and word instructions (w=1)
d	A 1-bit field in an instruction, "d" identifies direction, i.e. whether a specified register is source or destination.
(...)	Parentheses mean the contents of the enclosed register or memory location.
(BX)	Represents the contents of register BX, which can mean the address where an 8-bit operand is located. To be so used in an assembler instruction, BX must be enclosed only in square brackets.
((BX))	Means this 8-bit operand, the contents of the memory location pointed at by the contents of register BX. This notation is only descriptive for use in this chapter. It cannot appear in source statements.
(BX) + 1, (BX)	Means the address (of a 16-bit operand) whose low-order 8-bits reside in the memory location pointed at by the contents of register BX and whose high-order 8-bits reside in the next sequential memory location, (BX) + 1.
((BX) + 1, (BX))	Means the 16-bit operand that resides there.
Concatenation, e.g., ((DX) + 1: (DX))	Means a 16-bit word that is the concatenation of two 8-bit bytes, the low-order byte in the memory location pointed at by DX and the high-order byte in the next sequential memory location.
addr	Address (16-bit) of a byte in memory.
addr-low	Least significant byte of an address.
addr-high	Most significant byte of an address.
addr + 1: addr	Addresses of two consecutive bytes in memory, beginning at addr.
data	Immediate operand (8-bit if w=0; 16-bit if w=1).
data-low	Least significant byte of 16-bit data word.
data-high	Most significant byte of 16-bit data word.
disp	Displacement
disp-low	Least significant byte of 16-bit displacement.
disp-high	Most significant byte of 16-bit displacement.
←	Assignment
+	Addition
−	Subtraction
•	Multiplication
/	Division
%	Modulo
&	And
|	Inclusive or
||	Exclusive or

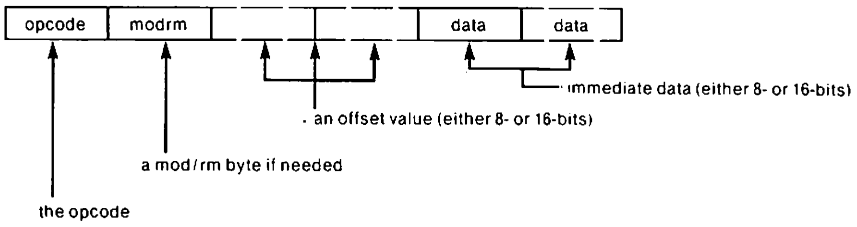
Table 6-4. Effective Address Calculation Time

EA Components		Clocks*
Displacement Only		6
Base or Index Only	(BX BP, SI, DI)	5
Displacement + Base or Index	(BX, BP, SI, DI)	9
Base + Index	BP + DI, BX + SI	7
	BP + SI, BX + DI	8
Displacement + Base + Index	BP + DI + DISP BX + SI + DISP	11
	BP + SI + DISP BX + DI + DISP	12

*Add 2 clocks for segment override

Sample 8086/8088 Instruction

Format


opcode
modrm
data
data
immediate data (either 8- or 16-bits)
an offset value (either 8- or 16-bits)
a mod/rm byte if needed
the opcode

Opcode	Clocks	Operation	Coding Example
(the value of the opcode byte)	(number of clocks required)	(the machine operation)	MNEMONIC

Operation

(A description of the machine operation.)

Flags

O D I T S Z A P C

(shows the effect on the flags)

Description

(Describes the use/operation of the instruction.)

ASCII Adjust for Addition

Format

Opcode

Opcode	Clocks	Operation	Coding Example
37	4	adjust AL, flags, AH	AAA

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do;
  AL ← AL + 6
  AH ← AH + 1
  CF ← AF ← 1
end;
AL ← AL & 0FH
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	X	U	X

Description

AAA is used to correct the result of adding two unpacked BCD digits in the AL register. After the normal byte addition, AAA tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then a carry bit is added to the AH register, and the low nibble of AL is increased by 6 to produce the decimal digit. AL is masked to 4 bits whether an adjustment was performed or not, thus always leaving an unpacked BCD result in the low nibble of AL. High nibble data does not affect the corrected result of the addition, so ASCII digits can be added correctly by following the AAA with an OR AL,30H to restore the result to an ASCII character. The digit carry, in AH, is not affected by this restoration.

ASCII Adjust for Division

Format

Long —— Opcode

Opcode	Clocks	Operation	Coding Example
D5,0A	60	Adjust AL, AH prior to division	AAD

Operation

AL ← AL + (AH * 0AH)
AH ← 0

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	U	X	U

Description

AAD is used to prepare 2 unpacked BCD digits (least significant in AL, most significant in AH) for a division operation that will yield an unpacked result. This is accomplished by multiplying AH by 10 and adding the product to AL. Then AH is zeroed, leaving AX with the binary equivalent of the original unpacked 2-digit number.

ASCII Adjust for Multiplication

Format

Long — — Opcode

Opcode	Clocks	Operation	Coding Example
D4,0A	83	Adjust AL, AH alter multiplication	AAM

Operation

AH ← (AL / 0AH)
AL ← (AL MOD 0AH)

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	U	X	U

Description

AAM is used to produce 2 unpacked BCD digits (least significant in AL, most significant in AH) after a multiplication of 2 unpacked digits. This is accomplished by dividing the binary product in AL by ten. The quotient is left in AH as the most significant digit, and the remainder is left in AL as the least significant digit.

ASCII Adjust for Subtraction

Format

Opcode

Opcode	Clocks	Operation	Coding Example
3F	4	adjust AL, flags, AH	AAS

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do:
  AL ← AL - 6
  AH ← AH - 1
  CF ← AF ← 1
end:
AL ← AL & 0FH
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	X	U	X

Description

AAS is used to correct the result of subtracting two unpacked BCD digits in the AL register. After the normal byte subtraction, AAS tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If the AF is set or the low nibble of AL is greater than 9, then a borrow bit is subtracted from AH, and the low nibble of AL is decreased by 6 to produce the proper decimal digit. AL is masked to 4 bits whether an adjustment was performed or not, thus always leaving an unpacked BCD result in the low nibble of AL. High nibble data does not affect the corrected result of the subtraction, so ASCII digits can be subtracted correctly by following the AAS with an OR AL,30H to restore the result to an ASCII character. The digit borrow, in AH, is not affected by this restoration.

ADC

Integer Add With Carry

Format

Memory/Reg + Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
12	3	Reg8 ← CF + Reg 8 + Reg8	ADC BL,CL
12	9 + EA	Reg8 ← CF + Reg8 + Mem8	ADC BL,BYTESOMETHING
13	3	Reg16 ← CF + Reg16 + Reg16	ADC BX,CX
13	9 + EA	Reg16 ← CF + Reg16 + Mem16	ADC BX,WORDSOMETHING
10	16 + EA	Mem8 ← CF + Mem8 + Reg8	ADC BYTESOMETHING,BL
11	16 + EA	Mem16 ← CF + Mem16 + Reg16	ADC WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
14	4	AL ← CF + AL + Immed8	ADC AL,5
15	4	AX ← CF + AX + Immed16	ADC AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*-(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← CF + Reg8 + Immed8	ADC BL,32
80	17 + EA	Mem8 ← CF + Mem8 + Immed8	ADC BYTESOMETHING,32
81	4	Reg16 ← CF + Reg16 + Immed16	ADC BX,1234H
81	17 + EA	Mem16 ← CF + Mem16 + Immed16	ADC WORDSOMETHING,1234H
83	4	Reg16 ← CF + Reg16 + Immed8	ADC BX,32
83	17 + EA	Mem16 ← CF + Mem16 + Immed8 (Immed8 is sign-extended before add in last 2 cases)	ADC WORDSOMETHING,32

Operation

LeftOpnd ← CF + LeftOpnd + RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

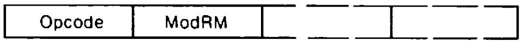
Description

The sum of two operands and the initial state of the carry flag replaces the left operand.

Integer Addition

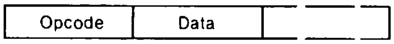
Format

Memory/Reg + Reg



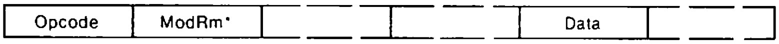
Opcode	Clocks	Operation	Coding Example
02	3	Reg8 ← Reg8 + Reg8	ADD BL.CL
02	9 + EA	Reg8 ← Reg8 + Mem8	ADD BL.BYTESOMETHING
03	3	Reg16 ← Reg16 + Reg16	ADD BX.CX
03	9 + EA	Reg16 ← Reg16 + Mem16	ADD BX.WORDSOMETHING
00	16 + EA	Mem8 ← Mem8 + Reg8	ADD BYTESOMETHING.BL
01	16 + EA	Mem16 ← Mem16 + Reg16	ADD WORDSOMETHING.BX

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
04	4	AL ← AL + Immed8	ADD AL.5
05	4	AX ← AX + Immed16	ADD AX.400H

Immed to Memory/Reg



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 + Immed8	ADD BL.32
80	17 + EA	Mem8 ← Mem8 + Immed8	ADD BYTESOMETHING.32
81	4	Reg16 ← Reg16 + Immed16	ADD BX.1234H
81	17 + EA	Mem16 ← Mem16 + Immed16	ADD WORDSOMETHING.1234H
83	4	Reg16 ← Reg16 + Immed8	ADD BX.32
83	17 + EA	Mem16 ← Mem16 + Immed8 (Immed8 is sign-extended before add in last 2 cases)	ADD WORDSOMETHING.32

Operation

LeftOpnd ← LeftOpnd + RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

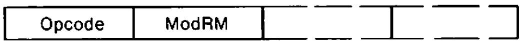
The sum of two operands replaces the left operand.

AND

Logical AND

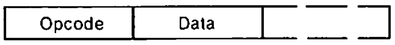
Format

Memory/Reg with Reg



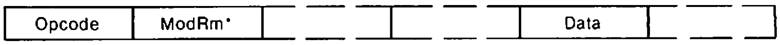
Opcode	Clocks	Operation	Coding Example
22	3	Reg8 ← Reg8 AND Reg8	AND BL,CL
22	9 + EA	Reg8 ← Reg8 AND Mem8	AND BL,BYTESOMETHING
23	3	Reg16 ← Reg16 AND Reg16	AND BX,CX
23	9 + EA	Reg16 ← Reg16 AND Mem16	AND BX,WORDSOMETHING
20	16 + EA	Mem8 ← Mem8 AND Reg8	AND BYTESOMETHING,BL
21	16 + EA	Mem16 ← Mem16 AND Reg16	AND WORDSOMETHING,BX

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
24	4	AL ← AL AND Immed8	AND AL,4
25	4	AX ← AX AND Immed16	AND AX,400H

Immed to Memory/Reg



*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 AND Immed8	AND BL,3FH
80	17 + EA	Mem8 ← Mem8 AND Immed8	AND BYTESOMETHING,3FH
81	4	Reg16 ← Reg16 AND Immed16	AND BX,3FFH
81	17 + EA	Mem16 ← Mem16 AND Immed16	AND WORDSOMETHING,3FFH

Operation

LeftOpnd ← LeftOpnd and RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The result of a bitwise logical AND of the two operands replaces the left operand. The carry and overflow flags are cleared.

Call

Format

Within segment or group, IP relative

Opcode	DispL	DispH

Opcode	Clocks	Operation	Coding Example
E8	19	IP ← IP + Disp16 —(SP) ← return link	CALL NEAR LABEL FOO

Within segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
FF	16	IP ← Reg16 —(SP) ← return link	CALL SI
FF	21 + EA	IP ← Mem16 —(SP) ← return link	CALL WORD PTR [SI]
FF	21 + EA	IP ← Mem16 —(SP) ← return link	CALL POINTER_TO_FRED

Operation

```
if IP-relative then do;
  IP ← IP + Disp16;
  —(SP) ← return link;
else do;
  IP ← (EA);
  —(SP) ← return link;
end if;
```

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

There are two types of within-segment or group calls: one that is IP-relative and is specified by the use of a NEAR label as the target address, and one in which the target address is taken from a register or variable pointer without modification (i.e., is NOT IP-relative). In the first case, the 16-bit displacement is relative to the first byte of the next instruction.

The second case is specified when the operand is any (16-bit) general, base, or index register—as in CALL AX, CALL BP, or CALL DI, respectively—or when the operand is a word-variable, as in CALL WORD PTR [BP] or CALL OPEN_ROUTINE[BX] (assuming that OPEN_ROUTINE is declared a word array or structure element). When the effective address is a variable, as in the preceding two examples, DS is the implied segment register for all EA's not using BP.

The return link, which is pushed to the TOS during the CALL, is the address of the instruction following the CALL.

Inter-segment or group, Direct

Opcode	offset	offset	segbase	segbase

Opcode	Clocks	Operation	Coding Example

9A	28	CS ← segbase
		IP ← offset
			CALL FAR LABEL FOO

Operation

CS ← segbase;
IP ← offset;
—(SP) ← return link;

Flags

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

Inter-segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example

FF	37 + EA	CS ← segbase
		IP ← offset
			CALL DWORD PTR FOO

Operation

CS ← (EA + 2);
JP ← (EA);

Flags

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

Description

An intersegment or group (long or far) CALL will transfer control by replacing both the values in CS and IP. This effectively transfers control to another segment or group by changing both the base (paragraph number) and offset values.

Convert Byte to Word

Format

Opcode

Opcode	Clocks	Operation	Coding Example
98	2	convert byte in AL to word in AX	CBW

Operation

```
if (AL AND 80H) = 80H then do:
  AH ← 0FFh
else do:
  AH ← 0
end;
```

Flags

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

Description

CBW converts the byte in AL to a word in AX by sign extension of AL through AH. No flags are affected.

CLC

Clear Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F8	2	clear the carry flag	CLC

Operation

CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	0

Description

CLC clears the carry flag, CF. No other flags are affected.

Clear Direction Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FC	2	clear direction flag	CLD

Operation

DF ← 0

Flags

```
O D I T S Z A P C
- 0 - - - - - - -
```

Description

CLD clears the direction flag, DF. No other flags are affected.

Clear Interrupt Enable Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FA	2	clear interrupt flag	CLI

Operation

IF ← 0

Flags

```
O D I T S Z A P C
- - 0 - - - - - -
```

Description

CLI clears the interrupt enable flag, IF. No other flags are affected.

Complement Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F5	2	complement carry flag	CMC

Operation

```
if CF = 1 then do;
  CF ← 0
else do;
  CF ← 1
end;
```

Flags

```
O D I T S Z A P C
- - - - - - - - X
```

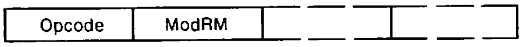
Description

CMC complements the carry flag, CF. No other flags are affected.

Compare Two Operands

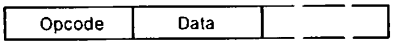
Format

Memory/Reg with Reg



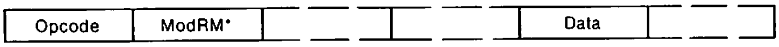
Opcode	Clocks	Operation	Coding Example
3A	3	flags ← Reg8 - Reg8	CMP BL,CL
3A	9 + EA	flags ← Reg8 - Mem8	CMP BL,BYTESOMETHING
3B	3	flags ← Reg16 - Reg16	CMP BX,CX
3B	9 + EA	flags ← Reg16 - Mem16	CMP BX,WORDSOMETHING
38	9 + EA	flags ← Mem8 - Reg8	CMP BYTESOMETHING,BL
39	9 + EA	flags ← Mem16 - Reg16	CMP WORDSOMETHING,BX

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
3C	4	flags ← AL - Immed8	CMP AL,5
3D	4	flags ← AX - Immed16	CMP AX,400H

Immed to Memory/Reg



*—(Reg field = 111)

Opcode	Clocks	Operation	Coding Example
80	4	flags ← Reg8 - Immed8	CMP BL,32
80	10 + EA	flags ← Mem8 - Immed8	CMP BYTESOMETHING,32
81	4	flags ← Reg16 - Immed16	CMP BX,1234H
81	10 + EA	flags ← Mem16 - Immed16	CMP WORDSOMETHING,1234H
83	4	flags ← Reg16 - Immed8	CMP BX,32
83	10 + EA	flags ← Mem16 - Immed8 (Immed 8 is sign-extended before sub in last 2 cases)	CMP WORDSOMETHING,32

Operation

flags ← LeftOpnd - RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

The flags are set by the subtraction of the right operand from the left operand. Neither operand is modified. A table of signed and unsigned comparisons supported by conditional jumps is provided under the 'Jcond' heading of this chapter.

Convert Word to Doubleword

Format

Opcode

Opcode	Clocks	Operation	Coding Example
99	5	convert word in AX to doubleword in DX:AX	CWD

Operation

```
if (AX AND 8000H) = 8000H then do:
  DX ← 0FFFFH
else do:
  DX ← 0
end;
```

Flags

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

Description

CWD converts the word in AX to a doubleword in DX:AX by sign extension of AX through DX. No flags are affected.

DAA

Decimal Adjust for Addition

Format

Opcode

Opcode	Clocks	Operation	Coding Example
27	4	adjust AL, flags, AH	DAA

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do;
  AL ← AL + 6
  AF ← 1
end;
if AL > 9F or CF = 1 then do;
  AL ← AL + 60H
  CF ← 1
end;
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	X	X	X

Description

DAA is used to correct the result of adding two bytes, each of which contains two packed BCD digits, in order to produce a packed decimal result. After the normal byte addition in AL, DAA tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then the low nibble of AL is increased by 6 to produce the correct decimal digit, and the high nibble of AL is incremented, effecting the digit carry.

Whether this first adjustment is made or not, a second adjustment is made if AL is greater than 9FH or if the CF is set, indicating a carry out of the high digit. In this case, 60H is added to AL and the CF is set.

Decimal Adjust for Subtraction

Format

Opcode

Opcode	Clocks	Operation	Coding Example
2F	4	adjust AL, flags, AH	DAS

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do;
  AL ← AL - 6
  AF ← 1
end;
if AL > 9F or CF = 1 then do;
  AL ← AL - 60H
  CF ← 1
end;
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	X	X	X

Description

DAS is used to correct the result of subtracting two bytes, each of which contains two packed BCD digits, in order to produce a packed decimal result. After the normal byte subtraction in AL, DAS tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then the low nibble of AL is reduced by 6 to produce the correct decimal digit.

Whether this first adjustment is made or not, a second adjustment is made if AL is greater than 9FH or the CF is set, indicating a borrow out of the high digit. In this case, 60H is subtracted from AL and the CF is set.

DEC

Decrement by 1

Format

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
48+reg	2	Reg16 ← Reg16 - 1	DEC BX

Memory/Byte Register

Opcode	ModRM*		

*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
FE	3	Reg8 ← Reg8 - 1	DEC BL
FE	15+EA	Mem8 ← Mem8 - 1	DEC BYTESOMETHING
FF	15+EA	Mem16 ← Mem16 - 1	DEC WORDSOMETHING

Operation

Operand ← Operand - 1

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

Description

The operand is decremented by 1.

Unsigned Division

Format

Memory/Reg with AX or DX:AX

Opcode	ModRM*		

*—(Reg field = 110)

Opcode	Clocks	Operation	Coding Example
F6	80-90	AH,AL ← AX / Reg8	DIV BL
F6	(86-96) + EA	AH,AL ← AX / Mem8	DIV BYTESOMETHING
F7	144-162	DX,AX ← DX:AX / Reg16	DIV BX
F7	(150-168) + EA	DX,AX ← DX:AX / Mem16	DIV WORDSOMETHING

Operation

```
if byte-operation then do;
  if AX / divisor > 0FFH then INT 0;
  else do;
    AL ← AX / divisor          /* unsigned division */
    AH ← AX MOD divisor        /* unsigned modulo */
  end if;
else do;                       /* word-operation */
  if DX:AX / divisor > 0FFFFH then INT 0
  else do;
    AX ← DX:AX / divisor       /* unsigned division */
    DX ← DX:AX MOD divisor     /* unsigned modulo */
  end if;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	U	U	U

Description

Depending on the opcode, either a word in AX is divided by a byte found in a register or memory location, or a doubleword in DX:AX is divided by a word register or memory location. A doubleword dividend is stored with its high word in DX and low word in AX, and the results are: DX gets the unsigned modulo, and AX gets the unsigned quotient. For a word dividend (byte divisor), the dividend is in AX and the results are: AH gets the unsigned modulo, and AL gets the unsigned quotient. In either case, if the result is too big to fit in the designated register (AX or AL) then an interrupt of type 0 is performed to allow the overflow to be handled.

ESC

Escape

Format

Opcode + i	ModRM		

Opcode	Clocks	Operation	Coding Example
D8+i	8+EA	data bus ← (EA)	ESC 6,ARRAY
D8+i	2	data bus ← (EA)	ESC 20,AL

Operation

if mod ≠ 11 then data bus ← (EA)
if mod = 11 then no operation

Flags

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

Description

The ESC instruction provides a mechanism by which other processors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 processor does no operation for the ESC instruction other than to access a memory operand and place it on the bus.

Halt

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F4	2	halt operation	HLT

Operation

cease operation;

Flags

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

Description

The HLT instruction causes the 8086/8088 processor to enter its halt state. The halt state is cleared by an enable interrupt or reset.

IDIV

Signed Division

Format

Memory/Reg with AX or DX:AX

Opcode	ModRM*		

*—(Reg field =111)

Opcode	Clocks	Operation	Coding Example
F6	101-112	AH,AL ← AX / Reg8	IDIV BL
F6	(107-118) + EA	AH,AL ← AX / Mem8	IDIV BYTESOMETHING
F7	165-184	DX,AX ← DX:AX / Reg16	IDIV BX
F7	(171-190) + EA	DX,AX ← DX:AX / Mem16	IDIV WORDSOMETHING

Operation

```
if byte-operation then do;
  if AX / divisor > 7FH or AX / divisor ← 80H then INT 0;
  else do;
    AL ← AX / divisor          /* signed division */
    AH ← AX MOD divisor        /* signed modulo */
  end if;
else do;                       /* word-operation */
  if DX:AX / divisor > 7FFFH or DX:AX / divisor ← 8000H then INT 0;
  else do;
    AX ← DX:AX / divisor       /* signed division */
    DX ← DX:AX MOD divisor     /* signed modulo */
  end if;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	U	U	U	U	U

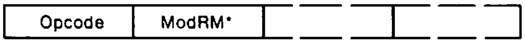
Description

Depending on the opcode, either a word in AX is divided by a byte in a register or memory location, or a dword in DX:AX is divided by a word register or memory location. A dword dividend is stored with its high word in DX and low word in AX, and the results are: DX gets the signed modulo, and AX gets the signed quotient. For a word dividend (byte divisor) the dividend is in AX, and the results are: AH gets the signed modulo, and AL gets the signed quotient. In either case, if the result is too big to fit in the designated register (AX or AL) then an interrupt of type 0 is performed to allow the overflow to be handled.

Signed Multiplication

Format

Memory/Reg with AL or AX



*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
F6	80-98	AX ← AL * Reg8	IMUL BL
F6	(86-104) + EA	AX ← AL * Mem8	IMUL BYTESOMETHING
F7	128-154	DX:AX ← AX * Reg16	IMUL BX
F7	(134-160) + EA	DX:AX ← AX * Mem16	IMUL WORDSOMETHING

Operation

```
if byte-operation then do;          /* byte operation, word result */
  AX ← AL * (Mem8 or Reg8);
  if AH is a sign extension of AL then CY ← OF ← 0;
  else CY ← OF ← 1;
else if word-operation then do;     /* word-operation, dword result */
  DX:AX ← AX * (Mem16 or Reg16);
  if DX is a sign extension of AX then CY ← OF ← 0;
  else CY ← OF ← 1;
else do;                            /* immed-operation, word result */
  Reg16 ← Immed16 * (Mem16 or Reg16);
  if product fits in destination register then CY ← OF ← 0;
  else CY ← OF ← 1;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

Description

There are two types of integer (signed) multiplication in the ASM86, distinguishable by the types of operands and the precision of the result:

1. Multiply a byte memory or register operand by a byte in AL, producing a word result in AX (called 'byte-operation, word result' above).
2. Multiply a word memory or register operand by a word in AX, producing a dword result in DX:AX (called 'word-operation, dword result' above).

Input Byte, Word

Format

Fixed port

Opcode	Port

Opcode	Clocks	Operation	Coding Example
E4	10	AL ← Port8	IN AL,BYTEPORTNUMBER
E5	10	AX ← Port8	IN AL,BYTEPORTNUMBER

Variable port

Opcode

Opcode	Clocks	Operation	Coding Example
EC	8	AL ← Port16(in DX)	IN AL,DX
ED	8	AX ← Port16(in DX)	IN AX,DX

Operation

```
if fixed-port then
  portnumber in instruction;
  0 ≤ portnumber ≤ 0FFH;
else
  portnumber in DX;
  0 ≤ portnumber ≤ 0FFFFH;
end if;
if byte-input then AL ← ioport[portnumber];
else AX ← ioport[portnumber];
```

Flags

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

Description

IN transfers a byte or word from the specified input port to AL or AX. Use of the fixed port format allows access to ports 0 through FF, and encodes the port number in the instruction. To use the variable port format you load the DX register with a 16 bit port number and then code the mnemonic 'DX' in place of a constant port number. This format allows access to 64k ports.

INC

Increment By 1

Format

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
40 + reg	2	Reg16 ← Reg16 + 1	INC BX

Memory/Byte Register

Opcode	ModRM*		

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
FE	3	Reg8 ← Reg8 + 1	INC BL
FE	15 + EA	Mem8 ← Mem8 + 1	INC BYTESOMETHING
FF	15 + EA	Mem16 ← Mem16 + 1	INC WORDSOMETHING

Operation

Operand ← Operand + 1

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

Description

The operand is incremented by 1.

Interrupt

Format

Opcode	type

Opcode	Clocks	Operation	Coding Example
CC	52	Interrupt 3	INT 3
CD	51	Interrupt 'type'	INT 5
CE	53 or 4	Interrupt 4 if FLAGS.OF = 1, else NOP	INTO

Operation

```
SP ← SP - 2
—(SP) ← FLAGS
IF ← 0
TF ← 0
SP ← SP - 2
—(SP) ← CS
CS ← TYPE * 4 + 2
SP ← SP - 2
—(SP) ← IP
IP ← TYPE * 4
```

Flags

O	D	I	T	S	Z	A	P	C
-	-	0	0	-	-	-	-	-

Description

INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. The one-byte form of this instruction generates a type 3 interrupt.

INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 (location 10H) if the OF flag is set (trap on overflow). If the OF flag is clear, no operation takes place.

IRET

Return from Interrupt

Format

Opcode

Opcode	Clocks	Operation	Coding Example
CF	24	Return from interrupt	IRET

Operation

IP ← (SP) + +
SP ← SP + 2
CS ← (SP) + +
SP ← SP + 2
FLAGS ← (SP) + +
SP ← SP + 2

Flags

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

Description

IRET returns control to an interrupted routine by transferring control to the return address saved by a previous interrupt operation and restoring the saved flag registers (as in POPF).

Jump on Condition

Operation

```
if condition is true then do;
  sign-extend displacement to 16 bits;
  IP ← IP + sign-extended displacement;
  end if;
```

Format

Opcode	Disp

Opcode	Clocks	Operation	Coding Example	
77	16 or 4	jump if above	JA	TARGETLABEL (CF OR ZF)=0
73	16 or 4	jump if above or equal	JAE	TARGETLABEL CF=0
72	16 or 4	jump if below	JB	TARGETLABEL CF=1
76	16 or 4	jump if below or equal	JBE	TARGETLABEL (CF OR ZF)=1
72	16 or 4	jump if carry set	JC	TARGETLABEL CF=1
74	16 or 4	jump if equal	JE	TARGETLABEL ZF=1
7F	16 or 4	jump if greater	JG	TARGETLABEL ((SF XOR OF) OR ZF)=0
7D	16 or 4	jump if greater or equal	JGE	TARGETLABEL (SF XOR OF)=0
7C	16 or 4	jump if less	JL	TARGETLABEL (SF XOR OF)=1
7E	16 or 4	jump if less or equal	JLE	TARGETLABEL ((SF XOR OF) OR ZF)=1
76	16 or 4	jump if not above	JNA	TARGETLABEL (CF OR ZF)=1
72	16 or 4	jump if neither above nor equal	JNAE	TARGETLABEL CF=1
73	16 or 4	jump if not below	JNB	TARGETLABEL CF=0
77	16 or 4	jump if neither below nor equal	JNBE	TARGETLABEL (CF OR ZF)=0
73	16 or 4	jump if no carry	JNC	TARGETLABEL CF=0
75	16 or 4	jump if not equal	JNE	TARGETLABEL ZF=0
7E	16 or 4	jump if not greater	JNG	TARGETLABEL ((SF XOR OF) OR ZF)=1
7C	16 or 4	jump if neither greater nor equal	JNGE	TARGETLABEL (SF XOR OF)=1
7D	16 or 4	jump if not less	JNL	TARGETLABEL (SF XOR OF)=0
7F	16 or 4	jump if neither less nor equal	JNLE	TARGETLABEL ((SF XOR OF) OR ZF)=0
71	16 or 4	jump if no overflow	JNO	TARGETLABEL OF=0
7B	16 or 4	jump if no parity	JNP	TARGETLABEL PF=0
79	16 or 4	jump if positive	JNS	TARGETLABEL SF=0
75	16 or 4	jump if not zero	JNZ	TARGETLABEL ZF=0
70	16 or 4	jump if overflow	JO	TARGETLABEL OF=1
7A	16 or 4	jump if parity	JP	TARGETLABEL PF=1
7A	16 or 4	jump if parity even	JPE	TARGETLABEL PF=1
7B	16 or 4	jump if parity odd	JPO	TARGETLABEL PF=0
78	16 or 4	jump if sign	JS	TARGETLABEL SF=1
74	16 or 4	jump if zero	JZ	TARGETLABEL ZF=1
E3	18 or 6	jump if CX is zero (does not test flags)	JCXZ	TARGETLABEL

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

Conditional jumps (except for JCXZ, explained below) test the flags, which presumably have been set in some meaningful way by a previous instruction. Because there are, in many instances, several meaningful and useful ways to interpret a particular state of the flags, ASM86 allows different mnemonics for each interpretation to resolve to the same op-code. This means that some op-codes are, in effect, synonyms for others. As an example, consider that a programmer who has just compared a character to another in AL might wish to jump if the two were equal (JE), while another who had just ANDed AX with a bit field mask would prefer to consider only whether the result was zero or not (he would use JZ, a synonym for JE).

JCXZ differs from the other conditional jumps in that it actually tests the contents of the CX register for zero, rather than interrogating the flags. This instruction is useful following a conditionally repeated string operation (REPE SCASB for example) or conditional loop instruction (such as LOOPNE TARGETLABEL), both of which may terminate for either of two reasons. These instructions implicitly use a limiting count in the CX register, and looping (or repeating) ends either when the CX register goes to zero or when the condition specified in the instruction (flags indicating equals in both of the above cases) occurs. JCXZ is useful when the two terminations must be handled differently.

In every case, if the condition specified in the conditional jump is true, the signed displacement byte is sign extended to a word and added to the IP, which has been updated to point to the first byte of the next instruction. This limits the range of the conditional jump to 127(decimal) bytes beyond and 126 bytes before the instruction (remember, the IP was incremented by 2 to point to the next instruction before the displacement was added).

Jump

Format

Within segment or group, IP relative

Opcode	DispL	DispH

Opcode	Clocks	Operation	Coding Example
E9	15	IP ← IP + Disp16	JMP NEAR_LABEL_FOO
EB	15	IP ← IP + Disp8 (Disp8 sign-extended)	JMP SHORT NR_LAB_FOO

Within segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
FF	11	IP ← Reg16	JMP SI
FF	18+EA	IP ← Mem16	JMP WORD PTR [SI]
FF	18+EA	IP ← Mem16	JMP POINTER_TO_FRED

Operation

```
if IP-relative then do;
  if short then sign-extend Disp8 to Disp16;
  IP ← IP + Disp16;
else do;
  IP ← (EA);
end if;
```

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

There are two types of within-segment jumps: one which is IP-relative and is specified by the use of a NEAR label as the target address; and one in which the target address is taken from a register or variable pointer without modification (i.e. is NOT IP-relative). In the first case, the displacement—which is relative to the first byte of the next instruction—may be either a full word or a byte which will be sign-extended to a word.

The second case is specified when the operand is any (16-bit) general, base, or index register—as in JMP AX, JMP BP, or JMP DI, respectively—or when the operand is a word-variable, as in JMP WORD PTR [BP], or JMP CS:CASE_TABLE[BX] (assuming that CASE_TABLE was defined as an array of word pointers). When the effective address is a variable, as in the preceding two examples, DS is the implied segment register for all EA's not using BP. Note especially the difference between JMP BX and JMP [BX]. In the first jump the new IP is taken from a register, while in the second it comes from a word variable which is pointed at by the register.

Inter-segment or group, Direct

Opcode	offset	offset	segbase	segbase

Opcode	Clocks	Operation	Coding Example

EA	15	CS ← segbase
		IP ← offset
			JMP FAR_LABEL_FOO

Operation

CS ← segbase
IP ← offset

Flags

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

Inter-segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example

FF	24 + EA	CS ← segbase
		IP ← offset
			JMP CASE_TABLE[BX]

Operation

CS ← EA.segbase;
IP ← EA.offset;

Flags

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

Description

The long jumps transfer control using both an offset and paragraph number (segbase), which may be either included in the instruction itself or found in a DWORD variable.

Load AH From Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9F	4	copy low byte of flags word to AH	LAHF

Operation

```
AH ← SF:ZF:X:AF:X:PF:X:CF
/* 'x' indicates non-specified bit value */
```

Flags

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

Description

The Sign, Zero, Auxiliary carry, Parity, and Carry Flags are transferred to AH in the following format:

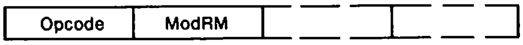
SF goes to AH bit7
ZF goes to AH bit6
AF goes to AH bit4
PF goes to AH bit2
CF goes to AH bit0

The remaining bits are indeterminate.
No flags are altered.

LDS/LES

Load Pointer to DS/ES and Register

Format



Opcode	Clocks	Operation	Coding Example
C4	16+EA	dword pointer at EA goes to reg16 (1st word) and ES (2nd word)	LES BX,DWORDPOINTER
C5	16+EA	dword pointer at EA goes to reg16 (1st word) and DS (2nd word)	LDS BX,DWORDPOINTER

Operation

```
Reg16 ← Mem16 @ EA               /* offset part of Virtual Address DWord */
DS (or ES) ← Mem16 @ EA+2        /* selector part of Virtual Address DWord */
```

Flags

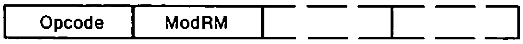
O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Description

The double word in the memory location designated by the effective address and 3 successive bytes is treated as two word operands. The first of these in EA:EA+1 is the offset part of the pointer and is loaded into the designated word-register. The second word, at EA+2:EA+3, is the paragraph number (segment base) of the address, and is loaded into the DS or ES register.

Load Effective Address

Format



Opcode	Clocks	Operation	Coding Example
8D	2+EA	Reg16 ← EA	LEA BX,SOMEVARIABLE [SI]

Operation

```
if EA = register then UDtrap;
else Reg 16 ← offset(EA)
```

Flags

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

Description

The effective address of the memory operand is put in the specified register. You should use this instruction only if EA requires run time calculation, i.e., has indexing with index or base register. Otherwise, you should use MOV reg, OFFSET variable.

LOCK

Assert Bus Lock

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F0	2	assert the bus lock next instruction	LOCK XCHG AX,SEMAPHORE

Operation

None.

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

A special one-byte lock prefix may precede any instruction. It causes the processor to assert its bus-lock signal for the duration of the operation caused by the instruction. In multiple processor systems with shared resources it is necessary to provide mechanisms to enforce controlled access to those resources. Such mechanisms, while generally provided through software operating systems, require hardware assistance. A sufficient mechanism for accomplishing this is a *locked exchange* (also known as test-and-set-lock).

It is assumed that external hardware, upon receipt of that signal, will prohibit bus access for other bus masters during the period of its assertion.

The instruction most useful in this context is an exchange register with memory. A simple software lock may be implemented with the following code sequence:

```
Check:  MOV   AL,1     ;set AL to 1 (implies locked)
  LOCK  XCHG  Sema,AL  ;test and set lock
        TEST  AL, AL   ;set flags based on AL
        JNZ   Check    ;retry if lock already set
        .
        .
        MOV   Sema,0   ;clear the lock when done
```

The LOCK prefix may be combined with the segment override and/or REP prefixes, although the latter has certain problems. (See REP.)

Loop Control

Format

Opcode	Disp

Opcode	Clocks	Operation	Coding Example
E1	18 or 6	dec CX; loop if equal and CX not 0	LOOPE TARGETLABEL
E0	19 or 5	dec CX; loop if not equal and CX not 0	LOOPNE TARGETLABEL
E1	18 or 6	dec CX; loop if zero and CX not 0	LOOPZ TARGETLABEL
E0	19 or 5	dec CX; loop if not zero and CX not 0	LOOPNZ TARGETLABEL
E2	17 or 5	dec CX; loop if CX not 0	LOOP TARGETLABEL

Operation

```
CX ← CX - 1;
if (condition is true) and (CX <> 0) then do;
  sign-extend displacement to 16 bits;
  IP ← IP + sign-extended displacement;
end if;
```

Flags

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

Description

The LOOP instructions are intended to provide iteration control and combine loop index management with conditional branching. To use the LOOP instruction you load an unsigned iteration count into CX, then code the LOOP at the end of a series of instructions to be iterated. Each time LOOP is executed the CX register is decremented and a conditional branch to the top of the loop is performed. The five variants of the instruction (LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ) allow branching on three sets of conditions, since two pairs of variants are synonymous. Conditions for branching are:

LOOP	branches if CX non-zero after decrementing;
LOOPZ, LOOPE	branch if CX non-zero and ZF = 1;
LOOPNZ, LOOPNE	branch if CX non-zero and ZF = 0.

In every case, if the condition specified in the conditional loop is true, the signed displacement byte is sign extended to a word and added to the IP, which has been updated to point to the first byte of the next instruction. This limits the range of the conditional loop to 127 (decimal) bytes beyond and 126 bytes before the instruction (remember, the IP was incremented by 2 to point to the next instruction before the displacement was added).

MOV

Move Data

Format

Memory/Reg to or from Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
88	9+EA	Mem8 ← Reg8	MOV BYTESOMETHING,BL
8A	2	Reg8 ← Reg8	MOV BL,AL
89	9+EA	Mem16 ← Reg16	MOV WORDSOMETHING,BX
8B	2	Reg16 ← Reg16	MOV BX,AX
8A	8+EA	Reg8 ← Mem8	MOV BL,BYTESOMETHING
8B	8+EA	Reg16 ← Mem16	MOV BX,WORDSOMETHING

Direct-Addressed Memory to or from AX/AL

Opcode	AddrL	AddrH

Opcode	Clocks	Operation	Coding Example
A0	10	AL ← Mem8	MOV AL,BYTESOMETHING
A1	10	AX ← Mem16	MOV AX,WORDSOMETHING
A2	10	Mem8 ← AL	MOV BYTESOMETHING,AL
A3	10	Mem16 ← AX	MOV WORDSOMETHING,AX

Immed to Reg

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
B0+reg	4	Reg 8 ← Immed8	MOV CL,5
B8+reg	4	Reg16 ← Immed16	MOV SI,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
C6	4	Reg8 ← Immed8	MOV BL,32
C6	10+EA	Mem8 ← Immed8	MOV BYTESOMETHING,32
C7	4	Reg16 ← Immed16	MOV BX,1234H
C7	10+EA	Mem16 ← Immed16	MOV WORDSOMETHING,1234H

Memory/Reg to or from SReg

Opcode	ModRM*		

*—(Reg field = SReg)

Opcode	Clocks	Operation	Coding Example
8C	9+EA	Mem16 ← SReg	MOV WORDSOMETHING,DS
8C	2	Reg16 ← SReg	MOV AX,DS
8E	8+EA	SReg* ← Mem16	MOV DS,WORDSOMETHING
8E	2	SReg* ← Reg16	MOV DS,AX

*CS not allowed

Operation

LeftOpnd ← RightOpnd

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

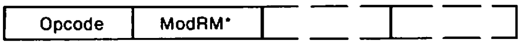
The right operand (source) is copied to the left operand (destination). The right operand is not modified. No flags are affected.

MUL

Unsigned Multiplication

Format

Memory/Reg with AL or AX



*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
F6	70-77	AX ← AL * Reg8	MUL BL
F6	(76-83) + EA	AX ← AL * Mem8	MUL BYTESOMETHING
F7	118-133	DX:AX ← AX * Reg16	MUL BX
F7	(124-139) + EA	DX:AX ← AX * Mem16	MUL WORDSOMETHING

Operation

```
if byte-operation then do;          /* byte operation, word result */
  AX ← AL * (Mem8 or Reg8);
  if AH = 0 then CY ← OF ← 0;
  else CY ← OF ← 1;
else if word-operation then do;     /* word-operation, dword result */
  DX:AX ← AX * (Mem16 or Reg16);
  if DX = 0 then CY ← OF ← 0;
  else CY ← OF ← 1;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

Description

There are two types of unsigned multiplication in the 8086/8088, distinguishable by the types of operands and the precision of the result:

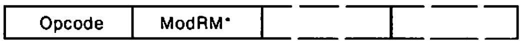
1. Multiply a byte memory or register operand by a byte in AL, producing a word result in AX (called 'byte-operation, word result' above).
2. Multiply a word memory or register operand by a word in AX, producing a dword result in DX:AX (called 'word-operation, dword result' above).

In both types of multiply the carry and overflow flags are used to signal whether the product has exceeded the precision of the operands which produced it. Thus, when multiplying two bytes, if the product is larger than can be expressed in a byte (i.e. prod > 256.) then the CY and OF flags will be set; otherwise, they will be cleared.

Negate an Integer

Format

Memory/Reg



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
F6	3	Reg8 ← 00H - Reg 8	NEG BL
F7	3	Reg16 ← 0000H - Reg16	NEG BX
F6	16+EA	Mem8 ← 00H - Mem8	NEG BYTESOMETHING
F7	16+EA	Mem16 ← 0000H - Mem16	NEG WORDSOMETHING

Operation

Operand ← 2's complement of Operand

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	1*

*except when operand is zero, then CF ← 0

Description

The two's complement of the register or memory operand replaces the old operand value.

NOP

No Operation

Format

Opcode

Opcode	Clocks	Operation	Coding Example
90	3	no operation	NOP

Operation

Perform no operation.

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

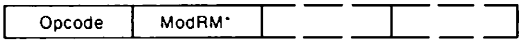
Description

NOP is a one-byte filler instruction which takes up space but affects none of the machine context except IP.

Form One's Complement

Format

Memory/Reg



*—(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
F6	3	Reg8 ← 0FFH - Reg8	NOT BL
F6	16 + EA	Mem8 ← 0FFH - Mem8	NOT BYTESOMETHING
F7	3	Reg16 ← 0FFFFH - Reg16	NOT BX
F7	16 + EA	Mem16 ← 0FFFFH - Mem16	NOT WORDSOMETHING

Operation

Operand ← one's complement of Operand

Flags

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

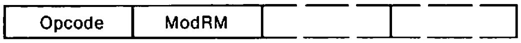
Description

The operand is inverted, that is, every 1 becomes a 0 and vice versa.

Logical Inclusive OR

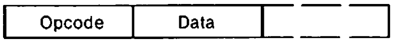
Format

Memory/Reg with Reg



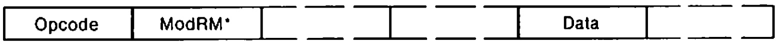
Opcode	Clocks	Operation	Coding Example
0A	3	Reg8 ← Reg8 OR Reg8	OR BL,CL
0A	9+EA	Reg8 ← Reg8 OR Mem8	OR BL,BYTESOMETHING
0B	3	Reg16 ← Reg16 OR Reg 16	OR BX,CX
0B	9+EA	Reg16 ← Reg16 OR Mem16	OR BX,WORDSOMETHING
08	16+EA	Mem8 ← Mem8 OR Reg8	OR BYTESOMETHING,BL
09	16+EA	Mem16 ← Mem16 OR Reg16	OR WORDSOMETHING,BX

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
0C	4	AL ← AL OR Immed8	OR AL,5
0D	4	AX ← AX OR Immed16	OR AX,400H

Immed to Memory/Reg



*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 OR Immed8	OR BL,32
80	17+EA	Mem8 ← Mem8 OR Immed8	OR BYTESOMETHING,32
81	4	Reg16 ← Reg16 OR Immed16	OR BX,1234H
81	17+EA	Mem16 ← Mem16 OR Immed16	OR WORDSOMETHING,1234H

Operation

LeftOpnd ← LeftOpnd or RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The inclusive OR of two operands replaces the left operand. The carry and overflow flags are cleared.

Output Byte, Word

Format

Fixed port

Opcode	Port

Opcode	Clocks	Operation	Coding Example
E6	10	Port8 ← AL	OUT BYTEPORTNUMBER,AL
E7	10	Port8 ← AX	OUT BYTEPORTNUMBER,AX

Variable port

Opcode

Opcode	Clocks	Operation	Coding Example
EE	8	Port16 (in DX) ← AL	OUT DX,AL
EF	8	Port16 (in DX) ← AX	OUT DX,AX

Operation

```
if fixed-port then
  portnumber in instruction;
  0 ≤ portnumber ≤ 0FFH;
else
  portnumber in DX;
  0 ≤ portnumber ≤ 0FFFFH;
end if;
if byte-output then ioport[portnumber] ← AL;
else ioport[portnumber] ← AX;
```

Flags

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

Description

OUT transfers a byte from AL or a word from AX to the specified output port. Use of the fixed port format allows access to ports 0 through FF, and encodes the port number in the instruction. To use the variable port format you load the DX register with a 16 bit port number and then code the mnemonic 'DX' in place of a constant port number. This format allows access to 64k ports.

POP

Pop a Word From the Stack

Format

Word Memory

Opcode	ModRM*		

*—(Reg field=000)

Opcode	Clocks	Operation	Coding Example
8F	17+EA	Mem16 ← (SP)++	POP WORDSOMETHING

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
58+reg	8	Reg16 ← (SP)++	POP BX

Segment Register

Opcode + SReg

Opcode	Clocks	Operation	Coding Example
07+(SReg*8)	8	SReg ← (SP)++	POP DS

Operation

Operand ← TOS;
SP ← SP + 2;

Flags

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

Description

The word on the top of the stack replaces the previous contents of the memory, register, or segment register operand. The stack pointer is incremented by 2 to point to the new top of stack.

If the destination operand is a segment register, the value POPed will be a paragraph number.

POP CS is NOT allowed.

Pop the TOS Into the Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9D	8	FLAGS ← (SP)+ +	POPF

Operation

Flags ← TOS;
SP ← SP + 2;

Flags

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

Description

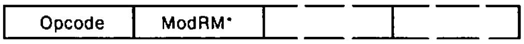
The TOS is copied to the Flags and the stack pointer is incremented by 2 to point to the new top of stack. Bit position to flag assignments are:

OF	← bit 11
DF	← bit 10
IF	← bit 9
TF	← bit 8
SF	← bit 7
ZF	← bit 6
AF	← bit 4
PF	← bit 2
CF	← bit 0

Push a Word Onto the Stack

Format

Memory/Reg



*—(Reg field=110)

Opcode	Clocks	Operation	Coding Example
FF	16+EA	—(SP) ← Mem16	PUSH WORDSOMETHING

Word Register

Opcode+reg

Opcode	Clocks	Operation	Coding Example
50+reg	11	—(SP) ← Reg16	PUSH BX

Segment Register

Opcode+SReg

Opcode	Clocks	Operation	Coding Example
06+(SReg*8)	10	—(SP) ← SReg	PUSH DS

Operation

SP ← SP - 2;
TOS ← Operand;

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

The stack pointer is decreased by 2 and the word operand is copied to the new top of stack.

PUSHF

Push the Flags to the Stack

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9C	10	—(SP) ← FLAGS	PUSHF

Operation

SP ← SP - 2;
TOS ← Flags;

Flags

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

Description

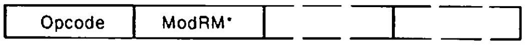
The stack pointer is decremented by 2 and the flags are copied to the new top of stack. Flag to bit position assignments are:

bit 11 ← OF
bit 10 ← DF
bit 9 ← IF
bit 8 ← TF
bit 7 ← SF
bit 6 ← ZF
bit 4 ← AF
bit 2 ← PF
bit 0 ← CF

Rotate Left Through Carry

Format

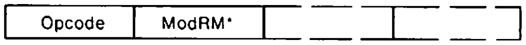
Memory or Reg by 1



*—(Reg field=010)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg 8 by 1	RCL BL,1
D0	15+EA	rotate Mem8 by 1	RCL BYTESOMETHING,1
D1	2	rotate Reg 16 by 1	RCL BX,1
D1	15+EA	rotate Mem16 by 1	RCL WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field=010)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	RCL BL,CL
D2	20+EA+4/bit	rotate Mem8 by CL	RCL BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	RCL BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	RCL WORDSOMETHING,CL

Operation

```
if variable-bit-rotate then count=CL or count=Immed8;
else count=1;
do until count=0
  tempcf ← CF;
  CF ← high-order-bit of operand;
  operand ← operand * 2 + tempcf;
  count ← count - 1;
```

```
end do;
if not variable-bit-rotate then do;
  if high-order-bit of operand <> CF then OF ← 1;
  else OF ← 0;
end if;
```

Flags

```
O D I T S Z A P C
X - - - - - - - X
```

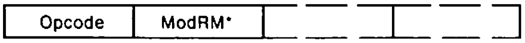
Description

The register or memory operand is rotated left through the CF according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the resulting carry flag. Only CF and OF are affected.

Rotate Right Through Carry

Format

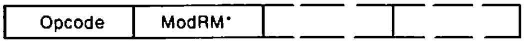
Memory or Reg by 1



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	RCR BL,1
D0	15+EA	rotate Mem8 by 1	RCR BYTESOMETHING,1
D1	2	rotate Reg16 by 1	RCR BX,1
D1	15+EA	rotate Mem16 by 1	RCR WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	RCR BL,CL
D2	20+EA+4/bit	rotate Mem8 by CL	RCR BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	RCR BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	RCR WORDSOMETHING,CL

Operation

```
if variable-bit-rotate then count=CL
else do;
  count=1;
  if high-order-bit of operand <> CF then OF ← 1;
  else OF ← 0;
end if;
do until count=0
  tempcf ← CF;
  CF ← low-order-bit of operand;
  operand ← operand / 2;
  high-order-bit of operand ← tempcf;
  count ← count - 1;
end do;
```

Flags

```
O D I T S Z A P C
X - - - - - - - X
```

Description

The register or memory operand is rotated right through the CF according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. If the shift count is 1, the overflow flag is set if the high bit of the un-rotated operand differs from the original carry flag. Only CF and OF are affected.

Repeat Prefix REP/REPZ/REPE/REPNE/REPNZ

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F3	2	repeat next instruction until CX=0	REP MOVSB
F3	2	repeat next instruction until CX=0 or ZF=1	REPE SCASB REPZ SCASB
F2	2	repeat next instruction until CX=0 or ZF=0	REPNE SCASB REPNZ SCASB

Operation

```
do while CX <> 0;
  /* acknowledge pending interrupts */
  /* perform string operation in subsequent byte */
  CX ← CX - 1;  /* does not affect flags */
  if string operation = SCAS or CMPS and
    ZF <> repeat condition then undo;
end do;
```

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

The REP prefix causes a succeeding string operation to be repeated until the count in CX goes to zero (REP causes CX to be decremented after each repetition of the string op). If the string operation is either SCAS or CMPS (or a variant of those such as SCASB...) then the ZF is compared to the repeat condition after the string op is performed, and the repeat is terminated if the ZF does not match the condition. For example, REPE SCASB will scan a string, comparing each byte to the AL register, as long as the ZF is 1, indicating 'EQUAL'.

REP, REPE, and REPZ are synonymous, as are REPNZ and REPNE.

Execution of the repeated string operation will not resume properly following an interrupt if more than one prefix is present preceding the string primitive. Execution will resume one byte before the primitive (presumably where the repeat resides), thus ignoring the additional prefixes.

RET

Return From Subroutine

Format

Opcode

Opcode	Clocks	Operation	Coding Example
C3	8	intra-segment return	RET
CB	18	inter-segment return	RET

Return and add constant to SP

Opcode	DataL	DataH

Opcode	Clocks	Operation	Coding Example
C2	12	intra-segment ret and add	RET 8
CA	17	inter-segment ret and add	RET 8

Operation

```
IP ← (SP) + + ;
SP ← SP + 2;
if intersegment then
  CS ← (SP) + + ;
  SP ← SP + 2;
if add immediate to SP then
  SP ← SP + immediate constant;
```

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

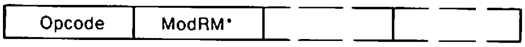
Description

RET transfers control through a back-link on the stack, reversing the effects of a CALL instruction. If the intra-segment RET is used, the back-link is assumed to be just the return-IP, while inter-segment RETs assume both IP and CS are on the stack. RETs may optionally add a constant to the stack pointer, effectively removing any arguments to the called routine which were pushed prior to the CALL.

Rotate Left

Format

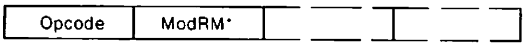
Memory or Reg by 1



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	ROL BL,1
D0	15 + EA	rotate Mem8 by 1	ROL BYTESOMETHING,1
D1	2	rotate Reg16 by 1	ROL BX,1
D1	15 + EA	rotate Mem16 by 1	ROL WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
D2	8 + 4/bit	rotate Reg8 by CL	ROL BL,CL
D2	20 + Ea + 4/bit	rotate Mem8 by CL	ROL BYTESOMETHING,CL
D3	8 + 4/bit	rotate Reg16 by CL	ROL BX,CL
D3	20 + EA + 4/bit	rotate Mem16 by CL	ROL WORDSOMETHING,CL

Operation

```
if variable-bit-rotate then count=CL
else count=1;
do until count=0
  CF ← high-order-bit of operand;
  operand ← operand * 2 + CF;
  count ← count - 1;
```

```
end do;
if not variable-bit-rotate then do;
  if high-order-bit of operand <> CF then OF ← 1;
  else OF ← 0;
end if;
```

Flags

```
O D I T S Z A P C
X - - - - - - - X
```

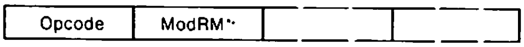
Description

The register or memory operand is rotated left according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. The high order bit of the operand is copied directly to the low order bit during the rotate, as well as to CF. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the resulting carry flag. (That is, if the high and low order bits of the result are not the same.) Only CF and OF are affected.

Rotate Right

Format

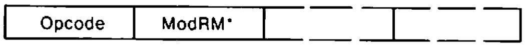
Memory or Reg by 1



*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	ROR BL,1
D0	15+EA	rotate Mem8 by 1	ROR BYTESOMETHING,1
D1	2	rotate Reg16 by 1	ROR BX,1
D1	15+EA	rotate Mem16 by 1	ROR WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	ROR BL,CL
D2	20+EA+4/bit	rotate Mem8 by CL	ROR BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	ROR BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	ROR WORDSOMETHING,CL

Operation

```
if variable-bit-rotate then count=CL
else count = 1;
do until count = 0
  tempcf ← CF;
  CF ← low-order-bit of operand;
  operand ← operand / 2;
  high-order-bit of operand ← CF;
  count ← count - 1;
end do;
if not variable-bit-rotate then do;
  if high-order-bit of operand <> CF then OF ←1;
  else OF ←0;
end if;
```

Flags

```
O D I T S Z A P C
X - - - - - - - X
```

Description

The register or memory operand is rotated right according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. The low bit of the operand is copied directly to the high bit during the rotate, as well as to the CF. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the un-rotated high bit. Only CF and OF are affected.

Store AH in Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9E	4	copy AH to low byte of flags word	SAHF

Operation

AH → SF:ZF:X:AF:X:PF:X:CF

/* 'X' indicates non-specified bit value */

Flags

O	D	I	T	S	Z	A	P	C
-	-	-	-	X	X	X	X	X

Description

The Sign, Zero, Auxiliary carry, Parity, and Carry Flags are loaded from AH in the following format:

AH bit7	goes to SF
AH bit6	goes to ZF
AH bit4	goes to AF
AH bit2	goes to PF
AH bit0	goes to CF

No other flags are altered.

SAL/SHL

Arithmetic/Logical Left Shift

Format

Memory or Reg by 1

Opcode	ModRM*		

*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SAL BL,1
D0	15 + EA	shift Mem8 by 1	SHL BYTESOMETHING,1
D1	2	shift Reg16 by 1	SHL BX,1
D1	15 + EA	shift Mem16 by 1	SAL WORDSOMETHING,1

Memory or Reg by count in CL

Opcode	ModRM*		

*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
D2	8 + 4/bit	shift Reg8 by CL	SHL BL,CL
D2	20 + EA + 4/bit	shift Mem8 by CL	SAL BYTESOMETHING,CL
D3	8 + 4/bit	shift Reg16 by CL	SAL BX,CL
D3	20 + EA + 4/bit	shift Mem16 by CL	SHL WORDSOMETHING,CL

Operation

```
if variable-bit-shift then count=CL
else count=1;
do until count=0
  CF ← high-order-bit of operand;
  operand ← operand * 2;
  count ← count - 1;
end do;
```

```
if not variable-bit-shift then do;
  if high-order-bit of operand <> CF then OF ← 1;
  else OF ← 0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Description

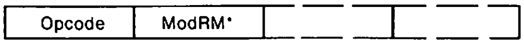
SHL (shift logical left) and SAL (shift arithmetic left) shift the operand left by COUNT bits, shifting in low-order zero bits.

SAR

Arithmetic Right Shift

Format

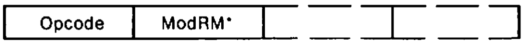
Memory or Reg by 1



*—(Reg field = 111)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SAR BL,1
D0	15 + EA	shift Mem8 by 1	SAR BYTESOMETHING,1
D1	2	shift Reg16 by 1	SAR BX,1
D1	15 + EA	shift Mem16 by 1	SAR WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field = 111)

Opcode	Clocks	Operation	Coding Example
D2	8 + 4/bit	shift Reg8 by CL	SAR BL,CL
D2	20 + EA + 4/bit	shift Mem8 by CL	SAR BYTESOMETHING,CL
D3	8 + 4/bit	shift Reg16 by CL	SAR BX,CL
D3	20 + EA + 4/bit	shift Mem16 by CL	SAR WORDSOMETHING,CL

Operation

```
if variable-bit-shift then count=CL
else count=1;
do until count=0
  CF ← low-order-bit of operand;
  operand ← operand / 2;   /* SIGNED DIVIDE */
  count ← count - 1;
end do;
```

```
if not variable-bit-shift then do;
  OF ← 0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Description

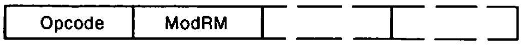
SAR (shift arithmetic right) shifts the operand right by COUNT bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

SBB

Integer Subtraction With Borrow

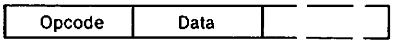
Format

Memory/Reg with Reg



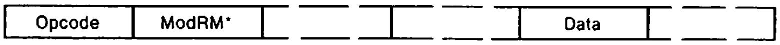
Opcode	Clocks	Operation	Coding Example
1A	3	Reg8 ← Reg8 - Reg8 - CF	SBB BL,CL
1A	9+EA	Reg8 ← Reg8 - Mem8 - CF	SBB BL,BYTESOMETHING
1B	3	Reg16 ← Reg16 - Reg16 - CF	SBB BX,CX
1B	9+EA	Reg16 ← Reg16 - Mem16 - CF	SBB BX,WORDSOMETHING
18	16+EA	Mem8 ← Mem8 - Reg8 - CF	SBB BYTESOMETHING,BL
19	16+EA	Mem16 ← Mem16 - Reg16 - CF	SBB WORDSOMETHING,BX

Immed from AX/AL



Opcode	Clocks	Operation	Coding Example
1C	4	AL ← AL - Immed8 - CF	SBB AL,5
1D	4	AX ← AX - Immed16 - CF	SBB AX,400H

Immed from Memory/Reg



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 - Immed8 - CF	SBB BL,32
80	17+EA	Mem8 ← Mem8 - Immed8 - CF	SBB BYTESOMETHING,32
81	4	Reg16 ← Reg16 - Immed16 - CF	SBB BX,1234H
81	17+EA	Mem16 ← Mem16 - Immed16 - CF	SBB WORDSOMETHING,1234H
83	4	Reg16 ← Reg16 - Immed8 - CF	SBB BX,32
83	17+EA	Mem16 ← Mem16 - Immed8 - CF (Immed8 is sign-extended before subtract)	SBB WORDSOMETHING,32

Operation

LeftOpnd ← LeftOpnd - RightOpnd - CF

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

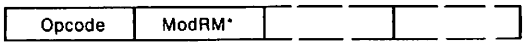
The result of subtracting the right operand, then the original value of the carry flag, from the left operand replaces the left operand.

SHR

Logical Right Shift

Format

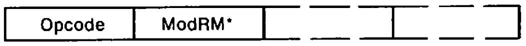
Memory or Reg by 1



*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SHR BL,1
D0	15+EA	shift Mem8 by 1	SHR BYTESOMETHING,1
D1	2	shift Reg16 by 1	SHR BX,1
D1	15+EA	shift Mem16 by 1	SHR WORDSOMETHING,1

Memory or Reg by count in CL



*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	shift Reg8 by CL	SHR BL,CL
D2	20+Ea+4/bit	shift Mem8 by CL	SHR BYTESOMETHING,CL
D3	8+4/bit	shift Reg16 by CL	SHR BX,CL
D3	20+EA+4/bit	shift Mem16 by CL	SHR WORDSOMETHING,CL

Operation

```
if variable-bit-shift then count=CL
else do;
  count=1;
  OF ← high-order-bit of operand;
end if;
do until count=0
  CF ← low-order-bit of operand;
  operand ← operand / 2;          /* UNSIGNED DIVIDE */
  count ← count - 1;
end do;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Description

SHR shifts the operand right by COUNT bits, shifting in high-order zero bits.

Set Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F9	2	set the carry flag	STC

Operation

CF ← 1

Flags

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	1

Description

STC sets the carry flag, CF. No other flags are affected.

Set Direction Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FD	2	set direction flag	STD

Operation

DF ← 1

Flags

```
O D I T S Z A P C
- 1 - - - - - - -
```

Description

STD sets the direction flag, DF. No other flags are affected.

STI

Set Interrupt Enable Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FB	2	set interrupt flag	STI

Operation

IF ← 1

Flags

```
O D I T S Z A P C
- - 1 - - - - - -
```

Description

STI sets the interrupt enable flag, IF. No other flags are affected.

String Operations

Format

Opcode

Opcode	Clocks	Operation	Coding Example
A6	22	flags ← (SI) - (DI)	CMPS BSTRING
A7	22	flags ← (SI) - (DI)	CMPS WSTRING
A4	18	(DI) ← (SI)	MOVS BSTRING1,BSTRING2
A5	18	(DI) ← (SI)	MOVS WSTRING1,WSTRING2
AE	15	flags ← (DI) - AX	SCAS BSTRING
AF	15	flags ← (DI) - AL	SCAS WSTRING
AC	12	AL ← (SI)	LODS BSTRING
AD	12	AX ← (SI)	LODS WSTRING
AA	11	(DI) ← AL	STOS BSTRING
AB	11	(DI) ← AX	STOS WSTRING

Operation

```
do until CX = 0;
   /* acknowledge any pending interrupts */
   perform string primitive once;
   CX    CX - 1;            /* does not affect flags * /
   if DF = 0 then add pointer adjustment to SI and/or DI
   else subtract pointer adjustment from SI and/or DI;
   if SCAS or CMPS, and repeat condition does not match ZF
   then undo;
end do;
```

Description

The string primitive operations are intended to be used primarily with the REP prefix. There are 7 primitives which, when so prefixed, perform the following operations:

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

CMPS
CMPSB
CMPSW
Compare the elements of two strings, one pointed to by ES:DI and the other by DS:SI.

Flags

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

MOVS MOVSB MOVSW	Move the string pointed to by DS:SI into memory pointed to by ES:DI.

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

SCAS SCASB SCASW	Scan a string pointed to by ES:DI, comparing each element to AX or AL according to the type of string, and setting the flags to the result of such a comparison. Used with the conditional repeat-prefix (REPE,...), this primitive can locate the next element matching AX/AL or next not-matching element.

Flags

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

LODS LODSB LODSW	Load each string element into AX/AL. This primitive would be used with the LOOP construct rather than the REP prefix, since some further processing on the data moved to AX/AL is almost surely necessary.

Flags

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

STOS STOSB STOSW	Store the AX or AL contents into the entire string.

Integer Subtraction

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
2A	3	Reg8 ← Reg8 - Reg8	SUB BL.CL
2A	9 + EA	Reg8 ← Reg8 - Mem8	SUB BL.BYTESOMETHING
2B	3	Reg16 ← Reg16 - Reg16	SUB BX.CX
2B	9 + EA	Reg16 ← Reg16 - Mem16	SUB BX.WORDSOMETHING
28	16 + EA	Mem8 ← Mem8 - Reg8	SUB BYTESOMETHING.BL
29	16 + EA	Mem16 ← Mem16 - Reg16	SUB WORDSOMETHING.BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
2C	4	AL ← AL - Immed8	SUB AL.5
2D	4	AX ← AX - Immed16	SUB AX.400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 - Immed8	SUB BL.32
80	17 + EA	Mem8 ← Mem8 - Immed8	SUB BYTESOMETHING.32
81	4	Reg16 ← Reg16 - Immed16	SUB BX.1234H
81	17 + EA	Mem16 ← Mem16 - Immed16	SUB WORDSOMETHING.1234H
83	4	Reg16 ← Reg16 - Immed8	SUB BX.32
83	17 + EA	Mem16 ← Mem16 - Immed8 (Immed8 is sign-extended before subtract)	SUB WORDSOMETHING.32

Operation

LeftOpnd ← LeftOpnd - RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

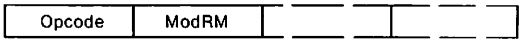
The result of subtracting the right operand from the left operand replaces the left operand.

TEST

Logical Compare

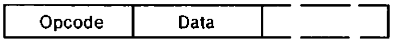
Format

Memory/Reg with Reg



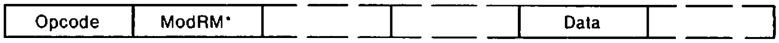
Opcode	Clocks	Operation	Coding Example
84	3	flags ← Reg8 AND Reg8	TEST BL,CL
84	9 + EA	flags ← Reg8 AND Mem8	TEST BL,BYTESOMETHING
85	3	flags ← Reg16 AND Reg16	TEST BX,CX
85	9 + EA	flags ← Reg16 AND Mem16	TEST BX,WORDSOMETHING

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
A8	4	flags ← AL AND Immed8	TEST AL,4
A9	4	flags ← AX AND Immed16	TEST AX,400H

Immed to Memory/Reg



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
F6	5	flags ← Reg8 AND Immed8	TEST BL,3FH
F6	11 + EA	flags ← Mem8 AND Immed8	TEST BYTESOMETHING,3FH
F7	5	flags ← Reg16 AND Immed16	TEST BX,3FFH
F7	11 + EA	flags ← Mem16 AND Immed16	TEST WORDSOMETHING,3FFH

Operation

flags ← LeftOpnd and RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The result of a bitwise logical AND of the two operands modifies the flags. Neither operand is modified.

Wait While TEST pin not Asserted

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9B	3+5n*	none	WAIT

*3+5n clocks where n is the number of times the TEST line is polled and found to be inactive.

Operation

None.

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

The WAIT instruction causes the processor to enter a wait state if the signal on a TEST pin is not asserted. The wait state may be interrupted by an enabled external interrupt. When this occurs the saved code location is that of the WAIT instruction, so that upon return from the interrupting task the wait state is re-entered. The wait state is cleared and execution resumed when the TEST signal is asserted. Execution resumes without allowing external interrupts until after the execution of the next instruction. The instruction allows the processor to synchronize itself with external hardware.

XCHG

Exchange Memory/Register With Register

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
86	4	Reg8 ←→ Reg8	XCHG BL,CL
86	17+EA	Mem8 ←→ Mem8	XCHG BYTESOMETHING,CL
87	4	Reg16 ←→ Reg16	XCHG BX,CX
87	17+EA	Mem16 ←→ Mem16	XCHG CX,WORDSOMETHING

Word Register with AX

Opcode + Reg

Opcode	Clocks	Operation	Coding Example
90+Reg	3	AX ←→ Reg16	XCHG AX,BX

Operation

```
temp ← left operand;
left operand ← right operand;
right operand ← temp;
```

Flags

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

Description

The two operands are exchanged. Segment registers are not legal operands. The order of the operands is immaterial. No flags are affected.

Table Look-up Translation

Format

Opcode

Opcode	Clocks	Operation	Coding Example
D7	11	replace AL with table entry	XLAT ASCII_TABLE (BYTE ARRAY)
D7	11		XLATB

Operation

AL ← table entry with effective address equal to BX + AL;

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

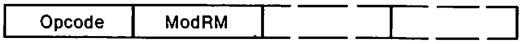
XLAT is intended for use as a table look-up instruction. You put the base address of the table in BX and a byte to be translated in AL. XLAT adds AL to the contents of BX and uses the result as an effective address. The byte at that EA is loaded into AL. BX is unchanged, and no flags are modified.

XOR

Logical Exclusive OR

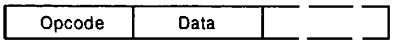
Format

Memory/Reg with Reg



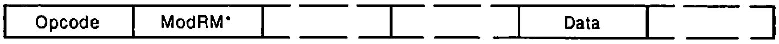
Opcode	Clocks	Operation	Coding Example
32	3	Reg8 ← Reg8 XOR Reg8	XOR BL,CL
32	9+EA	Reg8 ← Reg8 XOR Mem8	XOR BL,BYTESOMETHING
33	3	Reg16 ← Reg16 XOR Reg16	XOR BX,CX
33	9+EA	Reg16 ← Reg16 XOR Mem16	XOR BX,WORDSOMETHING
30	16+EA	Mem8 ← Mem8 XOR Reg8	XOR BYTESOMETHING,BL
31	16+EA	Mem16 ← Mem16 XOR Reg16	XOR WORDSOMETHING,BX

Immed to AX/AL



Opcode	Clocks	Operation	Coding Example
34	4	AL ← AL XOR Immed8	XOR AL,5
35	4	AX ← AX XOR Immed16	XOR AX,400H

Immed to Memory/Reg



*—(Reg field = 110)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 XOR Immed8	XOR BL,32
80	17+EA	Mem8 ← Mem8 XOR Immed8	XOR BYTESOMETHING,32
81	4	Reg16 ← Reg16 XOR Immed16	XOR BX,1234H
81	17+EA	Mem16 ← Mem16 XOR Immed16	XOR WORDSOMETHING,1234H

Operation

LeftOpnd ← LeftOpnd XOR RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The exclusive OR of two operands replaces the left operand. The carry and overflow flags are cleared.

The 8087 Instruction Set

This section provides a summary discussion of those elements of the 8087 Numeric Processor that are of specific interest to the 8087 programmer. The following programmer accessible features of the architecture are included: floating-point stack; status, control and tag words; exception pointers; and data types. An elementary description of 8087 operation is provided to give a working understanding of 8086/8087/8088 coprocessing, 8087 numeric processing, exception handlers, and 8087 emulators.

Those users who wish detailed information on the 8087 architecture, operation, and/or those who wish to write their own exception handlers are referred to *The 8086 Family User's Manual, Numerics Supplement,* Order No. 121586.

8087 Architectural Summary

The programmer accessible features of the 8087 Numeric Processor architecture consist of the eight floating-point stack elements; the seven words which constitute the 8087 environment (status word, control word, tag word, 2-word instruction address, and 2-word data address); and the seven data types accessible by the 8087.

Floating-Point Stack

The 8087 stack consists of eight elements divided into the fields shown in figure 6-1. The format of the fields corresponds with the temporary real data format used in all stack calculations and described under Data Types.

At a given point in time, the ST field in the status word identifies the current stack top element. This floating point stack element (rather than the status word field) is referred to in the rest of this chapter as ST. A load (push) operation, as in FLDLN2, decrements the stack pointer by 1 and loads a value (in this case $\log_e 2$) into the new stack top. An operation which pops the floating point stack increments the stack pointer by 1 (FADDP ST(i),ST adds the contents of the stack top to the stack element designated by (i), stores the result in ST(i) and increments the stack pointer by 1, making ST(1) the new stack top, ST(0).

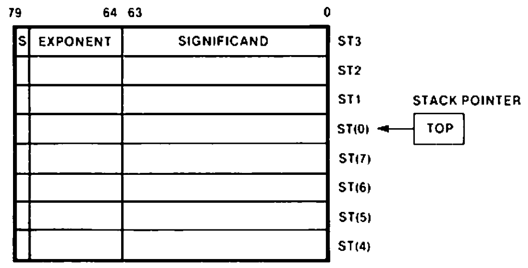


Figure 6-1. The 8087 Stack Fields 121623-8

Elements of the floating point stack can be addressed either implicitly or explicitly:

FST ST(3)	Stores the contents of the stack top into element 3.
FADD	Adds the contents of the stack top to the contents of ST(1), stores the result in ST(1) and pops the stack. The result is now in the new stack top.

Note that floating-point stack indices outside of the range 0-7 are flagged as "out of range."

Environment

The 8087 environment consists of the seven words shown in figure 6-2.

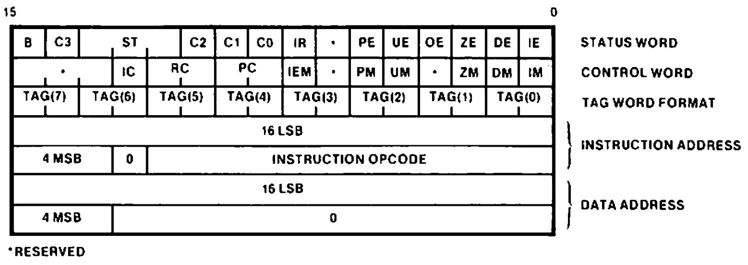


Figure 6-2. 8087 Environment 121623-9

Status Word

The status word reflects the overall condition of the 8087; it may be examined by storing it into memory with an 8087 instruction and then inspecting it with 8086/8088 CPU code. The status word is divided into the exception flag and status bit fields shown in figure 6-3. The busy field (bit 15) indicates whether the 8087 is executing an instruction (B=1) or is idle (B=0).

Several 8087 instructions (e.g., comparison instructions) result in modification of the condition code. The condition code is contained in bits 14 and 10-8 (C3-C0) of the status word. The condition code is used mainly for conditional branching. See the following instruction descriptions later in this chapter for condition code interpretations: FCOM, FCOMP, FCOMPP, FTST, FXAM and FPREM.

Bits 13-11 of the status word points to the 8087 stack element that is the current stack top (ST). Note that if ST=000B, a "push" operation which decrements ST, produces ST=111B; similarly, popping the stack with ST=111B yields ST=000B.

Bit 7 (IR) is the interrupt request field. The 8087 latches this bit to record a pending interrupt to the 8086/8088 CPU.

Bits 5-0 (PE, UE, OE, EE, DE, and IE) are set to indicate that the 8087 has detected an exception while executing an instruction.

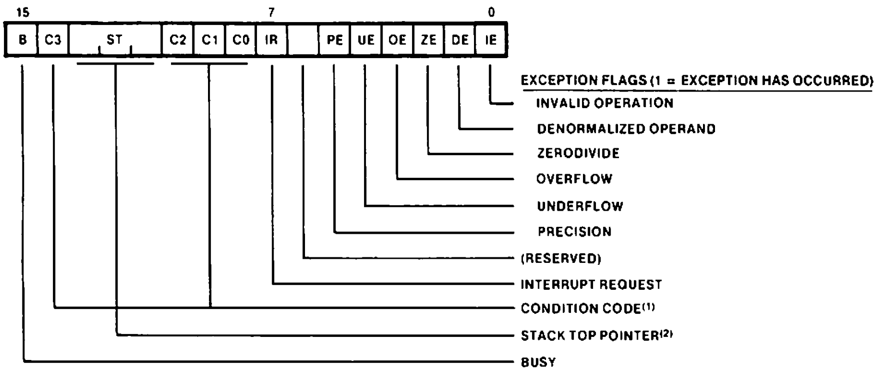


Figure 6-3. Status Word Format 121623-10

ST values

000 = element 0 is stack top
001 = element 1 is stack top
.
.
.
111 = element 7 is stack top

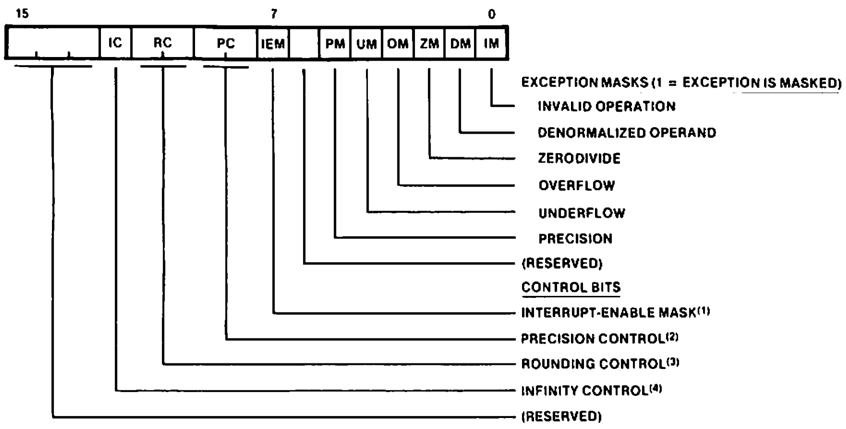
Control Word

The control word consists of the exception masks, an interrupt enable mask, and control bits as shown in figure 6-4. During the execution of most instructions, the 8087 checks for six classes of exception conditions:

1. Invalid operations—programming errors such as trying to load a floating point stack element that is not empty, popping an operand from an element that is empty, using operands that cause indeterminate results (0/0, square root of a negative number, trying to store an unnormalized number which will not denormalize, etc.).
2. Overflow—usually the exponent of the true result is too large for the destination real format.
3. Underflow—the true exponent is too small to be represented in the result format.
4. Zerodivide—division of a finite non-zero operand by zero.
5. Denormalized—an instruction attempts to operate on a denormalized number.
6. Precision—for instructions that perform exact arithmetic, this exception means that some precision has been lost in reporting the results of an operation.

When one of these six conditions occurs, the corresponding flag in the status word is set to 1. The 8087 checks the appropriate mask in the Control Word to determine if it should process the exception with a default handling procedure on chip (mask = 1) or invoke a user written exception handler (mask = 0).

In the first case, the exception is said to be MASKED (from user software).



(1) Interrupt-Enable Mask:
0 = Interrupts Enabled
1 = Interrupts Disabled (Masked)

(2) Precision Control:
00 = 24 bits
01 = (reserved)
10 = 53 bits
11 = 64 bits

(3) Rounding Control:
00 = Round to Nearest or Even
01 = Round Down (toward −∞)
10 = Round Up (toward +∞)
11 = Chop (Truncate Toward Zero)

(4) Infinity Control:
0 = Projective
1 = Affine

Figure 6-4. Control Word Format

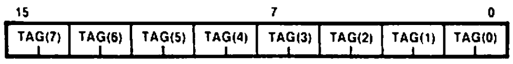
121623-11

The control bits have the following meanings:

PC:	Precision control—results are rounded to one of three precisions: Temporary Real (64 bits), Long Real (53 bits) or Short Real (24 bits).
RC:	Rounding Control—results are rounded in one of four directions: unbiased round to the nearest or even value, round toward +, round toward −, or round toward zero.
IC:	Infinity Control—there are two types of infinity arithmetic provided: affine and projective. The default means of closing a Number system is projective. See *The 8086 Family User's Manual, Numerics Supplement,* for a complete description.

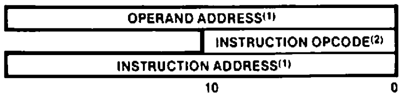
Tag Word

The tag word, as shown in figure 6-5, contains tags describing the contents of the corresponding stack elements.



Tag values:
00 = Valid (Normal or Unnormal)
01 = Zero (True)
10 = Special (Not-A-Number, ∞, or Denormal)
11 = Empty

Figure 6-5. Tag Word Format 121623-12



(1) 20-bit physical address
(2) 11 least significant bits of opcode: 5 most significant bits are always 8087 hook (11011B)

Figure 6-6. Exception Pointers Format 121623-13

Exception Pointers

The exception pointers shown in figure 6-6 are provided for user-written exception handlers. Whenever the 8087 executes an instruction, it saves the instruction address and the instruction opcode in the exception pointers. In addition, if the instruction references a memory operand, the address of the operand is retained also. An exception handler can be written to store these pointers in memory and obtain information concerning the instruction that caused the error.

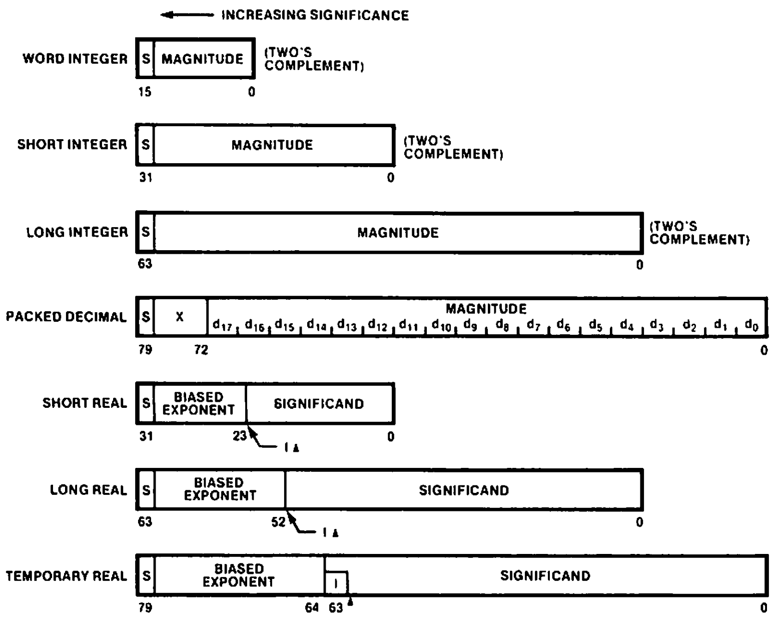
Data Types

The 8087 addresses seven different data types using all of the 8086 addressing modes. These data types and their valid ranges of value are shown in table 6-5.

Figure 6-7 describes how these formats are stored in memory (the sign is always located in the highest-addressed byte). In the figure, the most significant digits of all numbers (and field within numbers) are the leftmost digits.

Table 6-5. 8087 Data Types

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
WORD INTEGER	16	4-5	$-32768 \leq x \leq +32767$
SHORT INTEGER	32	9	$-2 \times 10^9 \leq x \leq 2 \times 10^9$
LONG INTEGER	64	18	$-9 \times 10^{18} \leq x \leq +9 \times 10^{18}$
PACKED DECIMAL	80	18	$-99...99 \leq x \leq +99...99$ (18 digits)
SHORT REAL	32	6-7	$0, 1.2 \times 10^{-38} \leq \lvert x \rvert \leq 3.4 \times 10^{38}$
LONG REAL	64	15-16	$0, 2.3 \times 10^{-308} \leq \lvert x \rvert \leq 1.7 \times 10^{308}$
TEMPORARY REAL	80	19-20	$0, 3.4 \times 10^{-4932} \leq \lvert x \rvert \leq 1.1 \times 10^{4932}$



NOTES:
S = Sign bit (0 = positive, 1 = negative)
d_n = Decimal digit (two per byte)
X = Bits have no significance: 8087 ignores when loading, zeros when storing.
▲ = Position of implicit binary point
I = Integer bit of significand; stored in temporary real, implicit in short and long real
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

Figure 6-7. Data Formats 121623-14

The three binary integer formats are identical except for length, which governs the range that can be accommodated in each format. The leftmost bit is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard two's complement notation (the binary integers are the only 8087 format to use two's complement). The quantity zero is represented with a positive sign (all bits 0). The 8087 word integer format is identical to the 16-bit signed integer data type of the 8086 and 8088.

Decimal integers are stored in packed decimal notation, with two decimal digits "packed" into each byte. Negative numbers are distinguished from positive ones only by the sign bit. All digits must be in the range 0H-9H.

The 8087 stores real numbers in a three-field binary format that resembles scientific notation. The number's significant digits are held in the SIGNIFICAND field, the EXPONENT field locates the binary point within the significant digits (determining the number's magnitude), and the SIGN field indicates whether the number is positive or negative. Negative numbers differ from positive numbers only in their sign bit.

The short and long real formats exist only in memory. If a number in one of these formats is loaded into the stack, it is automatically converted to temporary real.

Special values are included to increase flexibility though not within the domain of normal floating point arithmetic. These special values are listed here, but the reader is referred to *The 8086 Family User's Manual, Numerics Supplement,* for descriptions. The special values include:

- Signed zero
- $+\infty$ and $-\infty$ representations
- Indefinite values
- NAN values (Not-A-Number)
- Denormals
- Unnormals

8087 Operation

Coprocessing

The 8087 and host CPU act as coprocessors. They share the same instruction stream and sometimes perform parallel executions. The 8086/8088 has a set of ESCAPE instructions that, in memory addressing mode, cause the 8086/8088 to calculate the address and read the contents of that address. The 8086/8088 ignores the word it reads and executes subsequent instructions. The 8087, however, monitors the same instruction stream and when it detects an ESCAPE it begins processing. The 8087 latches the opcode and, if there was an address calculated, the 8087 captures both the address and the datum read by the 8086/8088. The 8087 decodes the instruction to determine how many more words it needs from memory. It increments the address and fetches data until all required data is read. The 8087 then releases the bus and begins calculating while the 8086/8088 continues executing the instruction stream.

The 8086/8088 WAIT instruction allows software to synchronize the 8086/8088 to the 8087 so that the host processor does not execute the next instruction until the 8087 is finished with its current (if any) instruction. To accomplish this, the programmer should explicitly code the FWAIT instruction immediately before an 8086/8088 instruction that accesses a memory operand read or written by a previous 8087 instruction.

If an 8087 and a processor other than its host CPU can both update a variable, access to that variable should be controlled so that one processor at a time has exclusive rights to it. This can be done by using an 8086/8088 XCHG instruction prefixed by LOCK. When the 8087 no longer needs the variable, the 8086/8088 clears it and again makes it available for use.

The 8087 interrupt requests are made to the 8086/8088 as the result of detecting an exception. Interrupts are enabled or disabled by the Interrupt Enable Mask (IEM) in the Control Word. When IEM is set to 1, interrupts are masked (disabled). The interrupt request remains set until it is explicitly cleared. This can be done by the FNCLEX, FNSAVE, or FINIT instructions.

Numeric Processing

The 8087 has four rounding modes, selectable by the RC field in the control word. The rounding modes and their corresponding RC fields are shown in table 6-6.

Table 6-6. Rounding Modes

RC Field	Rounding Mode	Rounding Action
00	Round to nearest	Closer to *b* of *a* or *c*; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward $-\infty$)	*a*
10	Round up (toward $+\infty$)	*c*
11	Chop (toward 0)	Smaller in magnitude of *a* or *c*

Note: $a < b < c$; *a* and *c* are representable, *b* is not.

Rounding occurs in arithmetic and store operations when the format of the destination cannot exactly represent the true result. This can happen when a precise temporary real number is stored in a shorter real format or in an integer format. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded. "Round to the nearest significant bit" is the default mode and is suitable for most applications. Other modes and applications are described in *The 8086 Family User's Manual, Numerics Supplement.*

The precision of results can be calculated to 64, 53, or 24 bits as selected by the PC field of the control word. The default setting is 64 bits. This setting is best suited for most applications.

The 8087's system of real numbers may be closed by either of two models of infinity. The IC field in the control word is set for either projective or affine closure. The default is projective, which is recommended for most computations. Both closure forms and their uses are described in *The 8086 Family User's Manual, Numerics Supplement.*

The 8087 can represent data and final results of calculations in the range $\pm 2.3\times10^{-308}$ to $\pm 1.7\times10^{308}$ (double precision). Compared to most computers, including large mainframes, the 8087 provides a very good approximation of the real number system. It is important to remember, however, that it is not an exact representation, and that arithmetic on real numbers is inherently approximate.

Conversely, and equally important, the 8087 does perform exact arithmetic on its integer subset of the reals. That is, an operation on two integers returns an exact integral result, provided that the true result is an integer and is in range.

The 8087 detects the six types of exceptions shown in table 6-7. The programmer has a choice of using the 8087 on-chip fault-handling capability by masking exceptions in the Control Word, or writing software exception handlers and unmasking exceptions in the control word. Table 6-3 shows the 8087 response to each situation.

If the exception is unmasked, its detection results in the generation of an interrupt. When an interrupt is generated, the interrupt procedure (exception handler) has available the exception flags, a pointer to the instruction causing the interrupt and a pointer to the datum if memory was addressed. Each of the exceptions shown in table 6-7 has a sticky flag associated with it, which means that once the flag is set, it remains until reset by software. Several instructions can be used to clear the flag: FCLEX clears exceptions; FRSTOR or FLDENV overwrite flags.

Those users who wish to write their own exception handlers should consult *The 8086 Family User's Manual, Numerics Supplement* since they will vary widely from one application to the next.

Table 6-7. Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN**, return it; if both are NANS, return NAN with larger absolute value; if neither is NAN, return *indefinite*.	Request interrupt.
Zerodivide	Return ∞ signed with "exclusive or" of operand signs.	Request interrupt.
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then re-evaluate for exceptions.	Request interrupt.
Overflow	Return properly signed ∞.	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt.

* On overflow, 24,576 decimal is *subtracted* from the true result's exponent; this forces the exponent back into range and permits a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is *added* to the true result's exponent.

** NAN is a member of a class of special values that exist in the real formats only. See the *The 8086 Family User's Manual, Numerics Supplement.*

8087 Emulators

Numeric processing capability is not restricted to 8087 users. Intel offers two 8086/8088 software products which provide 8087 functionality. E8087 emulates the full 8087 instruction set for assembly language programs. PE8087 furnishes numeric support for PL/M-86 software. Use of the 8087 Emulators necessitates modification of the instruction formats presented in this chapter.

ASM86, the Intel 8086/8087/8088 assembler, produces special object code for 8087 instructions. Floating point instructions are identified in such a way that they may be linked to the 8087 Emulators. Refer to the 8086/8087/8088 Assembler Operating Instructions for ISIS-II User's manual for a short description of this change and link procedure.

Organization of the 8087 Instruction Set

Data Transfer Instructions

These instructions are summarized in table 6-8. They move operands among stack elements or between the stack top and memory. Any of the seven data types can be converted to temporary real and loaded (pushed) onto the stack in a single operation; they can be stored in memory in the same manner. The data transfer instructions automatically update the 8087 tag word to reflect the stack contents following the instruction.

Table 6-8. Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

Arithmetic Instructions

The arithmetic instruction set for the 8087 provides a great many variations on the basic add, subtract, multiply and divide operations, and a number of other useful functions. Table 6-9 gives a summary of these instructions.

Table 6-9. Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

All mnemonics copyright Intel Corporation 1983

The stack element form is a generalization of the classical stack form; the programmer specifies the stack top as one operand and any stack element on the stack as the other operand. Coding the stack top as the destination provides a convenient way to make use of a constant held elsewhere in the stack. The converse coding (ST is the source operand) allows, for example, adding the top into a stack element used as an accumulator.

Often the operand in the stack top is needed for one operation but then is of no further use in the computation. The stack element and pop form can be used to pick up the stack top as the source operand, and then discard it by popping the floating point stack. Coding operands of ST(1),ST with a stack element pop mnemonic is equivalent to a classical stack operation: the top is popped and the result is left at the new top.

Programmers no longer need to spend valuable time eliminating square roots from algorithms because processors run too slowly. Other arithmetic instructions perform exact modulo division, round real numbers to integers, and scale values by powers of two.

The 8087's arithmetic instructions (addition, subtraction, multiplication, and division) allow the programmer to minimize memory references and to make optimum use of the 8087 floating-point stack.

Table 6-10 summarizes the available operation/operand forms that are provided for basic arithmetic. In addition to the four normal operations, two "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication.

- Operands may be located in stack elements or memory.
- Results may be deposited in a choice of stack elements.
- Operands may be a variety of 8087 data types: long real, short real, short integer or word integer, with automatic conversion to temporary real performed by the 8087.

Five instruction forms may be used across all six operations, as shown in table 6-10. The classical stack form may be used to make the 8087 operate like a classical stack machine. No operands are coded in this form, only the instruction mnemonic is coded. The 8087 picks the source operand from the stack top and the destination from the next stack element. It then performs the operation, pops the stack, and returns the result to the new stack top, effectively replacing the operands by the result.

Table 6-10. Basic Arithmetic Instructions and Operands

Instruction Form	Mnemonic Form	Operand Forms destination, source	ASM86 Example	
Classical stack	F*op*	{ST(1),ST}	FADD	
Stack element	F*op*	ST(i),ST or ST,ST(i)	FSUB	ST,ST(3)
Stack element and pop	F*op*P	ST(i),ST	FMULP	ST(2),ST
Real memory	F*op*	{ST,} short-real/long-real	FDIV	AZIMUTH
Integer memory	FI*op*	{ST,} word-integer/short-integer	FIDIV	N_PULSES

Notes: Braces { } surround *implicit* operands; these are not coded, and are shown here for information only.

op =	ADD	destination ← destination + source
	SUB	destination ← destination − source
	SUBR	destination ← source − destination
	MUL	destination ← destination · source
	DIV	destination ← destination ÷ source
	DIVR	destination ← source ÷ destination

All mnemonics copyright Intel Corporation 1983

The two memory forms increase the flexibility of the 8087's arithmetic instructions. They permit a real number or a binary integer in memory to be used directly as a source operand. This is a very useful facility in situations where operands are not used frequently enough to justify holding them in the floating point stack. Note that various forms of data allocation may be used to define these operands; they may be elements in arrays, structures or other data organizations, as well as simple scalars.

The six functional groups of instructions are discussed further in the next paragraphs.

Comparison Instructions

Each of these instructions (table 6-11) analyzes the top stack element, often in relationship to another operand, and reports the result in the status word condition code. The basic operations are compare, test (compare with zero), and examine (report tag, sign, and normalization). Special forms of the compare operation are provided to optimize algorithms by allowing direct comparisons with binary integers and real numbers in memory, as well as popping the stack after a comparison.

The FSTSW (store status word) instruction may be used following a comparison to transfer the condition code to memory for inspection. See individual descriptions of the instructions listed in table 6-11 for interpretations of the condition code bits.

Note that instructions other than those in the comparison group may update the condition code. To ensure that the status word is not altered inadvertently, it should be stored immediately after the compare operation.

Table 6-11. Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Transcendental Instructions

The instructions in this group are summarized in table 6-12. They perform the *core calculations* for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Prologue and epilogue software may be used to reduce arguments to the range accepted by the instructions and to adjust the result to correspond to the original arguments if necessary. The transcendentals operate on the top one or two stack elements, and they return their results to the stack.

Table 6-12. Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^X-1
FYL2X	$Y \cdot \log_2 X$
FYL2XP1	$Y \cdot \log_2(X+1)$

The transcendental instructions assume that their operands are *valid* and *in-range*. The instruction descriptions in this section provide the range of each operation. To be considered valid, an operand to a transcendental must be normalized; denormals, unnormals, infinities and NANs are considered invalid. Zero operands are accepted by some functions and are considered out-of-range by others. If a transcendental operand is invalid or out-of-range, the instruction will produce an undefined result without signaling an exception. It is the programmer's responsibility to ensure that operands are valid and in-range before executing a transcendental. FPREM may be used to bring an operand into range for periodic functions.

Constant Instructions

Each of these instructions (table 6-13) loads (pushes) a commonly-used constant onto the stack. The values have full temporary real precision (64 bits) and are accurate to approximately 19 decimal digits. Since a temporary real constant occupies 10 memory bytes, the constant instructions, which are only two bytes long, save storage and improve execution speed, in addition to simplifying programming.

Table 6-13. Constant Instructions

FLDZ	Load + 0.0
FLD1	Load + 1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Processor Control Instructions

When CPU interrupts are enabled, as will normally be the case when an application task is running, the "wait" forms of these instructions should be used. Most of the instructions shown in table 6-14 are used in system-level activities rather than in computations. These activities include: initialization, exception handling, and task switching.

Alternate mnemonics are shown for several of the processor control instructions in table 6-14. This mnemonic, distinguished by a second character of "N", instructs the assembler *not* to prefix the instruction with a CPU WAIT instruction (instead, a CPU NOP precedes the instruction). This "no-wait" form is intended for use in critical code regions where a WAIT instruction might precipitate an endless wait. Thus, when CPU interrupts are disabled, and the 8087 can potentially generate an interrupt, the "no-wait" form should be used.

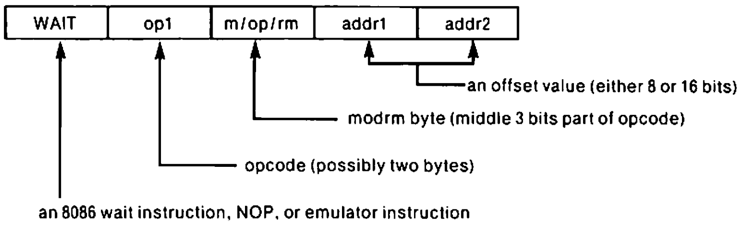
Except for FNSTENV and FNSAVE, all instructions which provide a no-wait mnemonic are self-synchronizing and can be executed back-to-back in any combination without intervening FWAITs. These instructions can be executed by one part of the 8087 while the other part is busy with a previously decoded instruction. To ensure that the processor control instruction executes after completion of any operation in progress, the "WAIT" form of that instruction should be used.

Table 6-14. Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Sample 8087 Instruction

Format


WAIT
op1
m/op/rm
addr1
addr2
an offset value (either 8 or 16 bits)
modrm byte (middle 3 bits part of opcode)
opcode (possibly two bytes)
an 8086 wait instruction, NOP, or emulator instruction

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
(the 8087 instruction coding)	(emulator instruction coding)	typical range	(machine operation)	MNEMONIC

Operation

(A description of the machine operation.)

Exceptions

I Z D O U P

(shows which exceptions could be set)

F2XM1

2^x-1

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F0	CD 19 F0	500 310-630	$ST \leftarrow 2^{ST}-1$	F2XM1

Operation

This instruction calculates the function $Y = 2^X-1$. X is taken from the top of the floating point stack and must be in the range $0 \leqslant X \leqslant 0.5$. The result Y replaces X at the stack top.

Exceptions

I	Z	D	O	U	P	*
				X	X	

*Operands not checked.

Description

This instruction is designed to produce a very accurate result even when x is close to zero. To obtain $Y = 2^X$, add 1 to the result delivered by F2XM1.

The following formulas show how values other than 2 may be raised to a power of X.

$$10^x = 2^{x \cdot \log_2 10}$$

$$e^x = 2^{x \cdot \log_2 e}$$

$$Y^x = 2^{x \cdot \log_2 y}$$

The 8087 has built-in instructions, described in this chapter, for loading the constants LOG_2 10 and LOG_2 e, and the FYL2X instruction may be used to calculate X * $\log_2 Y$.

Absolute Value

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E1	CD 19 E1	14 10-17	ST ← |ST|	FABS

Operation

The absolute value instruction changes the element in the top of the stack to its absolute value by making its sign positive.

Exceptions

I	Z	D	O	U	P
X					

FADD

Add Real

Format

Stack top + Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 C0+i	CD 18 C0+i	85 70-100	ST ← ST + ST(i)	FADD ST,ST(2)
9B DC C0+i	CD 1C C0+i	85 70-100	ST(i) ← ST + ST(i)	FADD ST(4),ST

Stack top + memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m0rm	CD 18 m0rm	105+EA (90-120)+EA	ST ← ST + mem-op (short-real)	FADD COUNT
9B DC m0rm	CD 1C m0rm	110+EA (95-125)+EA	ST ← ST + mem-op (long-real)	FADD MEAN

Operation

The add real instruction adds the source operand to the destination operand and places the result in the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

Add Real and Pop

Format

Stack top + Stack Element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE C1	CD 1E C1	90 75-105	ST(1) ← ST + ST(1) pop stack	FADD
9B DE C0 + i	CD 1E C0 + i	90 75-105	ST(i) ← ST + ST(i) pop stack	FADDP ST(2),ST

Operation

The add real and pop stack instruction adds the stack top to one of the stack elements, replacing the stack element with the sum, and then pops the floating point stack.

Exceptions

J	Z	D	O	U	P
X		X	X	X	X

FBLD

Packed Decimal (BCD) Load

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DF m4rm	CD IF m4rm	300 + EA (290-310) + EA	push stack ST ← mem-op	FBLD YTD_SALES

Operation

The BCD load instruction converts the memory operand from packed decimal to temporary real and pushes the result onto the stack. The sign of source is preserved, including the case when the value is negative zero.

Exceptions

I	Z	D	O	U	P
X					

Note

The packed decimal digits of the source are assumed to be in the range 0-9H. The instruction does not check for invalid digits (A-FH) and the result of attempting to load an invalid encoding is undefined.

Packed Decimal (BCD) Store and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DF m6rm	CD IF m6rm	530 + EA (520-540) + EA	mem-op ← ST pop stack	FBSTP FORECAST

Operation

The packed decimal store and pop stack instruction converts the contents of the stack top to a packed decimal integer, stores the result at the destination in memory, and pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X					

Note

FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then deleting least significant bits.

Users who are concerned about rounding may precede FBSTP with FRNDINT.

FCHS

Change Sign

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E0	CD 19 E0	15 10-17	ST ← −ST	FCHS

Operation

The change sign instruction complements the sign on the stack top element.

Exceptions

I	Z	D	O	U	P
X					

Clear Exceptions

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E2	CD 1B E2	5 2-8	clear 8087 exceptions	FCLEX
90 DB E2	CD 1B E2	5 2-8	clear 8087 exceptions (no wait)	FNCLEX

Operation

This instruction clears all exception flags, the interrupt request flag and the busy flag in the status word. As a consequence, the 8087's INT and BUSY lines go inactive. The FCLEX form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

FNCLEX is used in critical areas of code where a WAIT instruction might result in a deadlock. FCLEX is used to insure that the processor control instruction executes only after completion of any operation in progress in the NOP.

FCOM

Compare Real

Format

Compare Stack top and Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 D1	CD 18 D1	45 40-50	ST - ST(1)	FCOM
9B D8 D0+i	CD 18 D0+i	45 40-50	ST - ST(i)	FCOM ST(2)

Compare Stack top and memory operands

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m2rm	CD 18 m2rm	65+EA (60-70)+EA	ST - memop (short-real)	FCOM WAVELENGTH
9B DC m2rm	CD 1C m2rm	70+EA (65-75)+EA	ST - memop (long-real)	FCOM MEAN

Operation

The compare real instruction compares the stack top with the source operand. The source operand may be a stack element or short or long real memory operand. If no operand is coded. ST is compared with ST(1).

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                 ;STORE RESULT FROM FCOM
FWAIT                         ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1    ;MOVE STATUS BYTE TO AH
SAHF                          ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FCOMP

Compare Real and Pop

Format

Compare Stack top and Stack element and pop

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 D9	CD 18 D9	47 42-52	ST - ST(1) pop stack	FCOMP
9B D8 D8+i	CD 18 D8+i	47 42-52	ST - ST(i) pop stack	FCOMP ST(3)

Compare Stack top and memory operand and pop

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m3rm	CD 18 m3rm	68+EA (63-73)+EA	ST - mem-op pop stack (short-real)	FCOMP DENSITY
9B DC m3rm	CD 1C m3rm	72+EA (67-77)+EA	ST - mem-op pop stack (long-real)	FCOMP PERCENT

Operation

The compare real and pop stack instruction compares the stack top with the source operand and then pops the floating point stack. The source operand may be a stack element or short or long real memory operand. If no operand is coded, ST is compared with ST(1).

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                ;STORE RESULT FROM FCOM
FWAIT                        ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1   ;MOVE STATUS BYTE TO AH
SAHF                         ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FCOMPP

Compare Real and Pop Twice

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE D9	CD 1E D9	50 45-55	ST - ST(1) pop stack pop stack	FCOMPP

Operation

The compare real and pop stack twice instruction compares the stack top with ST(1) and pops the floating point stack twice, discarding both operands. No operands may be explicitly coded with this instruction.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87               ;STORE RESULT FROM FCOM
FWAIT                       ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1  ;MOVE STATUS BYTE TO AH
SAHF                        ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FDECSTP

Decrement Stack Pointer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F6	CD 19 F6	9	stack pointer ← 2	FDECSTP
		6-12	stack pointer - 1	

Operation

This instruction subtracts 1 from the stack top pointer in the status word. No tags or registers are altered, nor is any data transferred. Executing FDECSTP when the stack top pointer is 0, changes the pointer to 7.

Exceptions

I Z D O U P

Disable Interrupts

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E1	CD 1B E1	5 2-8	Set 8087 interrupt mask	FDISI
90 DB E1	CD 1B E1	5 2-8	Set 8087 interrupt mask (no wait)	FNDISI

Operation

The instruction sets the interrupt enable mask in the control word and prevents the NDP from issuing an interrupt request. The FDISI form of this instruction is preceded by an assembler-generated WAIT.

Exceptions

I Z D O U P

Description

The NO WAIT form of the instruction (FNDISI) is intended for use in critical code regions where a WAIT instruction might induce an endless wait.

Note

If WAIT is decoded with pending exceptions, the 8087 generates an interrupt— masked or not.

FDIV

Divide Real

Format

Stack top and Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 F0+i	CD 18 F0+i	198 193-203	ST ← ST/ST(i)	FDIV ST,ST(2)
9B DC F8+i	CD 1C F8+i	198 193-203	ST(i) ← ST(i)/ST	FDIV ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m6rm	CD 18 m6rm	220+EA (215-225)+EA	ST ← ST/mem-op (short-real)	FDIV DISTANCE
9B DC m6rm	CD 1C m6rm	225+EA (220-230)+EA	ST ← ST/mem-op (long-real)	FDIV GAMMA

Operation

The divide real instructions divide the destination by the source and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The divide real and pop stack instruction divides one of the stack elements by the stack top, replaces the stack element with the quotient, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

Divide Real and Pop

Format

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE F9	CD 1E F9	202 197-207	ST(1) ← ST(1)/ST pop stack	FDIV
9B DE F8+i	CD 1E F8+i	202 197-207	ST(i) ← ST(i)/ST pop stack	FDIVP ST(3),ST

Operation

The divide real instructions divide the destination by the source and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The divide real and pop stack instruction divides one of the stack elements by the stack top, replaces the stack element with the quotient, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FDIVR

Divide Real Reversed

Format

Stack top and Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 F8+i	CD 18 F8+i	199 194-204	ST ← ST(i)/ST	FDIVR ST,ST(2)
9B DC F0+i	CD 1C F0+i	199 194-204	ST(i) ← ST/ST(i)	FDIVR ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m7rm	CD 18 m7rm	221+EA (216-226)+EA	ST ← mem-op/ST (short-real)	FDIVR RATE
9B DC m7rm	CD 1C m7rm	226+EA (221-231)+EA	ST ← mem-op/ST (long-real)	FDIVR SPEED

Operation

The divide real reversed instructions divide the source operand by the destination and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse divide and pop stack instruction divides the stack top by one of the stack elements and returns the quotient to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

Divide Real Reversed and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE F1	CD 1E F1	203 198-208	ST(1) ← ST/ST(1) pop stack	FDIVR
9B DE F0 + i	CD 1E F0 + i	203 198-208	ST(i) ← ST/ST(i)	FDIVRP ST(4),ST

Operation

The divide real reversed instructions divide the source operand by the destination and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse divide and pop stack instruction divides the stack top by one of the stack elements and returns the quotient to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FENI
FNENI
Enable Interrupts

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E0	CD 1B E0	5 2-8	clear 8087 interrupt mask	FENI
90 DB E0	CD 1B E0	5 2-8	clear 8087 interrupt mask (no wait)	FNENI

Operation

This instruction clears the interrupt enable mask in the control word, allowing the 8087 to generate interrupt requests. The FENI form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

The NO WAIT form of the instruction (FNENI), is intended for use in critical code regions where a WAIT instruction might induce an endless wait.

The WAIT form of this instruction (FENI), should be used in all non-critical code regions. This form insures that the processor control instruction executes after completion of any operation in progress in the NEU.

Free Register

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD C0 + i	CD 1D C0 + i	11 9-16	TAG(i) masked empty	FFREE ST(1)

Operation

This instruction changes the destination stack element's tag to empty. The contents of this stack element are unaffected.

Exceptions

I Z D O U P

FIADD

Integer Add

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m0rm	CD 1A m0rm	125 + EA (108-143) + EA	ST ← ST + mem-op (short integer)	FIADD DISTANCE
9B DE m0rm	CD 1E m0rm	120 + EA (102-137) + EA	ST ← ST + mem-op (word integer)	FIADD PULSE

Operation

This instruction adds the integer memory source to the top of the stack and returns the sum to the destination at the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

Integer Compare

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m2rm	CD 1A m2rm	85 + EA (78-91) + EA	ST – mem-op (short integer)	FICOM PASSES
9B DE m2rm	CD 1E m2rm	80 + EA (72-86) + EA	ST – mem-op (word integer)	FICOM CENTS

Operation

The integer compare instructions convert the memory operand (a word or short binary integer) to temporary real and compare it with the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                ;STORE RESULT FROM FICOM
FWAIT                        ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1   ;MOVE STATUS BYTE TO AH
SAHF                         ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

Integer Compare and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m3rm	CD 1A m3rm	87 + EA (80-93) + EA	ST - mem-op pop stack (short integer)	FICOMP LIMIT
9B DE m3rm	CD 1E m3rm	82 + EA (74-88) + EA	ST - mem-op pop stack (word integer)	FICOMP SAMPLE

Operation

The integer compare instructions convert the memory operand (a word or short binary integer) to temporary real and compare it with the top of the stack. FICOMP additionally discards the value in ST by popping the floating point stack.

Exceptions

I	Z	D	O	U	P
X	X				

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                 ;STORE RESULT FROM FICOMP
FWAIT                         ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1    ;MOVE STATUS BYTE TO AH
SAHF                          ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

Integer Divide

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m6rm	CD 1A m6rm	236 + EA (230-243) + EA	ST ← ST/mem-op (short integer)	FIDIV SURVEY
9B DE m6rm	CD 1E m6rm	230 + EA (224-238) + EA	ST ← ST/mem-op (word integer)	FIDIV ANGLE

Operation

The integer divide instruction divides the top of the stack by the integer memory operand and returns the quotient to the top of the stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FIDIVR

Integer Divide Reversed

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m7rm	CD 1A m7rm	237 + EA (231-245) + EA	ST ← mem-op/ST (short integer)	FIDIVR COORD
9B DE m7rm	CD 1E m7rm	230 + EA (225-239) + EA	ST ← mem-op/ST (word integer)	FIDIVR FREQUENCY

Operation

The reversed integer divide instruction divides the integer memory operand by the top of the stack and returns the quotient to the stack top.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

Integer Load

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m0rm	CD 1B m0rm	56 + EA (52-60) + EA	push stack ST ← mem-op (short integer)	FILD STANDOFF
9B DF m0rm	CD 1F m0rm	50 + EA (46-54) + EA	push stack ST ← mem-op (word integer)	FILD SEQUENCE
9B DF m5rm	CD 1F m5rm	64 + EA (60-68) + EA	push stack ST ← mem-op (long integer)	FILD RESPONSE

Operation

The integer load instruction converts the integer memory operand from its binary integer format (word, short, or long) to temporary real and pushes the result onto the stack. The new stack top is tagged zero if all bits in the source were zero, and is tagged valid otherwise.

Exceptions

I	Z	D	O	U	P
X					

FIMUL

Integer Multiply

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m1rm	CD 1A m1rm	136 + EA (130-144) + EA	ST ← ST • mem-op (short integer)	FIMUL BEARING
9B DE m1rm	CD 1E m1rm	130 + EA (124-138) + EA	ST ← ST • mem-op (word integer)	FIMUL POSITION

Operation

The integer multiply instruction multiplies the integer memory operand and the top of the stack and returns the product to the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

Increment Stack Pointer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F7	CD 19 F7	9 6-12	stack pointer ← stack pointer + 1	FINCSTP

Operation

The stack pointer increment instruction adds 1 to the stack top pointer in the status word. It does not alter tags or register contents, nor does it transfer data. It is not equivalent to popping the stack since it does not set the tag of the previous stack to empty. Incrementing a stack pointer of 7 changes it to 0.

Exceptions

I Z D O U P

FINIT
FNINIT
Initialize Processor

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E3	CD 1B E3	5 2-8	initialize 8087	FINIT
90 DB E3	CD 1B E3	5 2-8	initialize 8087 (no wait)	FNINIT

Operation

The initialize processor instruction performs the functional equivalent of a hardware RESET, except that it does not affect the instruction fetch synchronization of the 8087 and its CPU. FINIT/FNINIT sets the control word to 03FFH, empties all floating point stack elements, and clears exception flags and busy interrupts. The FINIT form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Note

The system should call the INIT87 procedure in lieu of executing FINIT/FNINIT when the processor is first initialized, for compatability with the 8087 emulator.

Integer Store

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m2rm	CD 1B m2rm	88 + EA (82-92) + EA	mem-op ← ST (short integer)	FIST COUNT
9B DF m2rm	CD 1F m2rm	86 + EA (80-90) + EA	mem-op ← ST (word integer)	FIST FACTOR

Operation

The integer store instruction rounds the contents of the stack top to an integer (according to the RC field of the control word) and transfers the result to the memory destination. The destination may define a word or short integer variable. Negative zero is stored in the same encoding as positive zero: 0000...00.

Exceptions

I	Z	D	O	U	P
X					X

FISTP

Integer Store and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m3rm	CD 1B m3rm	90+EA (84-94)+EA	mem-op ← ST pop stack (short integer)	FISTP CORRECTED
9B DF m3rm	CD 1F m3rm	88+EA (82-92)+EA	mem-op ← ST pop stack (word integer)	FISTP ALPHA
9B DF m7rm	CD 1F m7rm	100+EA (94-105)+EA	mem-op ← ST pop stack (long integer)	FISTP READINGS

Operation

The integer store and pop stack instruction rounds the contents of the stack top to an integer (according to the RC field of the control word) and transfers the result to the memory destination. The floating point stack is popped following the transfer. The destination may be any of the binary integer data types.

Exceptions

I	Z	D	O	U	P
X					X

Integer Subtract

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m4rm	CD 1A m4rm	125 + EA (108-143) + EA	ST ← ST - mem-op (short integer)	FISUB BASE
9B DE m4rm	CD 1E m4rm	120 + EA (102-137) + EA	ST ← ST - mem-op (word integer)	FISUB SIZE

Operation

This instruction subtracts the integer memory operand from the top of the stack and returns the difference to the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

FISUBR

Integer Subtract Reversed

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m5rm	CD 1A m5rm	125 + EA (109-144) + EA	ST ← mem-op - ST (short integer)	FISUBR FLOOR
9B DE m5rm	CD 1E m5rm	120 + EA (103-139) + EA	ST ← mem-op - ST (word integer)	FISUBR BALANCE

Operation

The integer subtract reversed instruction subtracts the stack top from the integer memory source and returns the difference to the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X		X

Load Real

Format

Stack element to Stack top

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 C0 + i	CD 19 C0 + i	20 17-22	$T_1 \leftarrow$ ST(i) push stack ST $\leftarrow T_1$	FLD ST(2)

Memory operand to Stack top

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m0rm	CD 19 m0rm	43 + EA (38-56) + EA	push stack ST $\leftarrow$ mem-op (short real)	FLD READING
9B DD m0rm	CD 1D m0rm	46 + EA (40-60) + EA	push stack ST $\leftarrow$ mem-op (long real)	FLD TEMPERATURE
9B DB m5rm	CD 1B m5rm	57 + EA (53-65) + EA	push stack ST $\leftarrow$ mem-op (temp real)	FLD SAVEREADING

Operation

The load real instruction pushes the source operand onto the top of the floating point stack. This is done by decrementing the stack pointer by one and then copying the contents of the source to the new stack top. The source may be a stack element on the stack (ST(i)), or any of the real data types in memory. Short and long real source operands are converted to temporary real automatically. Executing FLD ST(0) duplicates the old stack top in the new stack top.

Exceptions

I	Z	D	O	U	P
X		X			

FLDCW

Load Control Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m5rm	CD 19 m5rm	10 + EA (7-14) + EA	processor control word ← mem-op	FLDCW CONTROL

Operation

This instruction replaces the current processor control word with the word defined by the source operand.

Exceptions

I Z D O U P

Description

This instruction is typically used to establish, or change, the 8087's mode of operation.

Note

If an exception bit in the status word is set, loading a new control word that unmasks that exception and clears the interrupt enable mask will generate an immediate request before the next instruction is executed. When changing modes, the recommended procedure is to first clear any exceptions and then load the new control word.

Load Environment

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m4rm	CD 19 m4rm	40 + EA (35-45) + EA	8087 environment ← mem-op	FLDENV ENV STORE

Operation

The load environment instruction reloads the 8087 environment from the memory area defined by the source operand. This data should have been written by a previous FSTENV/FNSTENV instruction.

Exceptions

I Z D O U P

Description

CPU instructions may immediately follow FLDENV, but no subsequent NDP instruction should be executed without an intervening FWAIT or assembler-generated WAIT.

Note

Loading an environment image that contains an unmasked exception causes an immediate interrupt request from 8087 (assuming IEM = 0 in the environment image).

FLDLG2

Load $\text{Log}_{10}2$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EC	CD 19 EC	21 18-24	push stack ST ← $\log_{10}2$	FLDLG2

Operation

The load log base 10 of 2 instruction pushes the value $\log_{10}2$ onto the top of the floating point stack. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

Load $\text{Log}_e 2$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 ED	CD 19 ED	20 17-23	push stack ST ← $\log_e 2$	FLDLN2

Operation

The load log base e of 2 instruction pushes the value $\log_e 2$ onto the top of the floating point stack. This constant has temporary real precision of 64 bits with an accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDL2E

Load Log_2e

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks ·Typical Range	Operation	Coding Example
9B D9 EA	CD 19 EA	18 15-21	push stack ST ← $\log_2 e$	FLDL2E

Operation

The load log base 2 of e instruction pushes the value $\log_2 e$ onto the top of the floating point stack. This value has full temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

Load $\text{Log}_2 10$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E9	CD 19 E9	19 16-22	push stack ST ← $\log_2 10$	FLDL2T

Operation

The load log base 2 of 10 instruction pushes the constant $\log_2 10$ onto the stack. This constant has temporary real precision of 64 bits with accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDPI

Load π

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EB	CD 19 EB	19 16-22	push stack ST ← π	FLDPI

Operation

This instruction pushes π onto the top of the stack. The π value has full temporary real precision of 64 bits with an accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

Load +0.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EE	CD 19 EE	14 11-17	push stack ST ← 0.0	FLDZ

Operation

The load zero instruction pushes the value +0.0 onto the top of the floating point stack. The constant has temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

FLD1

Load +1.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E8	CD 19 E8	18 15-21	push stack ST ← 1.0	FLD1

Operation

This instruction pushes the constant +1.0 onto the top of the floating point stack. This constant has full temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

Multiply Real

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 C8 + i	CD 18 C8 + i	138* 130-145*	ST ← ST • ST(i)	FMUL ST,ST(3)
9B DC C8 + i	CD 1C C8 + i	138* 130-145*	ST(i) ← ST(i) • ST	FMUL ST(2),ST

*Clocks are 97 / 90-105 when one or both operands are short.

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 mlrm	CD 18 mlrm	118 + EA (110-125) + EA	ST ← ST • mem-op (short real)	FMUL SPEED
9B DC mlrm	CD 1C mlrm	161 + EA* (154-168) + EA*	ST ← ST • mem-op (long real)	FMUL HEIGHT

*Clocks are 120 + EA / (112-126) + EA when one or both operands are short.

Operation

The multiply real instruction multiplies the destination operand by the source and returns the product to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FMULP

Multiply Real and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE C8 + i	CD 1E C8+i	142* 134-148*	ST(i) ← ST(i) • ST pop stack	FMULP ST(2),ST

*Clocks are 100 94-108 when one or both operands are short.

Operation

The multiply real instruction multiplies the destination operand by the source and returns the product to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The multiply real and pop stack instruction multiplies one of the stack elements by the stack top, replaces the stack element with the product, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

No operation

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 D0	CD 19 D0	13 10-16	ST ← ST	FNOP

Operation

This operation stores the stack top to the stack top and thus effectively performs no operation.

Exceptions

I Z D O U P

FPATAN

Partial Arctangent

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F3	CD 19 F3	650 250-800	$T_1 \leftarrow$ arctan (ST(1)/ST) pop stack ST $\leftarrow T_1$	FPATAN

Operation

The partial arctangent instruction computes the function Θ = ARCTAN (Y/X). X is taken from the top stack element and Y from ST(1). Y and X must observe the inequality $0 < Y < X < +\infty$. The instruction pops the floating point stack and returns Θ to the new stack top, overwriting the Y operand.

Exceptions

I	Z	D	O	U	P	*
				X	X	

*operands not checked

Description

This instruction assumes that the operands are valid and in-range. To be considered valid, an operand must be normalized. If an operand is either invalid or out-or-range, the instruction will produce an undefined result without signalling an exception.

Partial Remainder

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F8	CD 19 F8	125 15-190	ST ← REPEAT (ST ST(1))	FPREM

Operation

This instruction performs modulo division on the stack top by ST(1). FPREM produces an *EXACT* result; the precision exception does not occur. The sign of the remainder is the same as the sign of the original dividend.

Exceptions

I	Z	D	O	U	P
X		X		X	

Description

FPREM operates by performing successive subtractions. It can reduce a magnitude difference of up to 2^{64} in one execution. If FPREM produces a remainder that is less than the modulus (ST(1)), the function is complete and bit C2 of the status word condition code is cleared. If the function is incomplete, C2 is set to 1; the result in ST is then called the partial remainder.

Software can be used to inspect C2 by storing the status word following execution of FPREM and re-executing the instruction (using the partial remainder in ST as the dividend), until C2 is cleared. An alternate possibility is comparing ST to ST(1) to determine when the function is complete. If ST > ST(1), FPREM must be executed again. If ST = ST(1), the remainder is 0 and execution is complete. If ST < ST(1), execution is complete and the remainder is ST.

Note

A context switch between the instructions in the remainder loop can be forced by a higher priority interrupting routine which needs the 8087.

One important use of FPREM is to reduce arguments (operands) of periodic transcendental functions to the range permitted by these instructions. For example, the FPTAN (tangent) instruction requires its argument to be less than $\pi/4$. Using $\pi/4$ as a modulus, FPREM will reduce an argument so that it is in the range of FPTAN. Because FPREM produces an exact result, the argument reduction does NOT introduce roundoff error into the calculations even if several iterations are required to bring the argument into range. The rounding of π produces a rounded period rather than a rounded argument.

FPREM also provides the least-significant three bits of the quotient generated by FPREM (in C_3, C_1, C_0). This is also important for transcendental argument reduction since it locates the original angle in the correct one of eight $\pi/4$ segments of the unit circle.

FPTAN

Partial Tangent

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example

9B D9 F2	CD 19 F2	450
		30-540
			Y/X ← TAN (ST)
			ST ← Y
			push stack
			ST ← X
				FPTAN

Operation

The partial tangent instruction computes the function Y/X = TAN(Θ). Θ is taken from the top stack element. The value of Θ must be within the range $0 <= \Theta < \pi/4$. The result of the operation is a ratio; y replaces Θ in the stack and X is pushed, becoming the new stack top. Θ is measured in radians.

Exceptions

I	Z	D	O	U	P	*
X					X	

*operands not checked

Description

The ratio result of FPTAN is designed to optimize the calculation of the other trigonometric functions.

This instruction assumes that the operand is valid and in-range; to be considered valid, an operand must be normalized. If the operand is invalid or out-of-range, the instruction will produce an undefined result without signalling an exception.

Round to Integer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FC	CD 19 FC	45 16-50	ST ← nearest integer (ST)	FRNDINT

Operation

This instruction rounds the top stack element to an integer.

Exceptions

I	Z	D	O	U	P
X					X

Description

Assume that ST contains the 8087 real number encoding of the decimal value 155.625. FRNDINT will change the value to 155 if the RC field of the control word is set to down or chop; or to 156 if it is set to up or nearest.

FRSTOR

Restore Saved State

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m4rm	CD 1D m4rm	202 + EA (197-207) + EA	8087 state ← mem-op	FRSTOR STATE SAVE

Operation

The restore state instruction reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction.

Exceptions

I Z D O U P

Note

CPU instructions may immediately follow FRSTOR, but no NDP instruction should be executed without an intervening FWAIT or an assembler-generated WAIT.

The 8087 resets to its new state at the conclusion of the FRSTOR. The 8087 will, for example, generate an immediate interrupt request if indicated by the exception and mask bits in the memory image.

FSAVE FNSAVE

Save State

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m6rm	CD 1D m6rm	202 + EA (197-207) + EA	mem-op ← 8087 state	FSAVE STATE SAVE
90 DD m6rm	CD 1D m6rm	202 + EA (197-207) + EA	mem-op ← 8087 state (no wait)	FNSAVE STATE

Operation

The save state instruction writes the full 8087 state—environment plus register stack—to the memory location specified in the destination operand, and initializes the NDP. The FSAVE form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

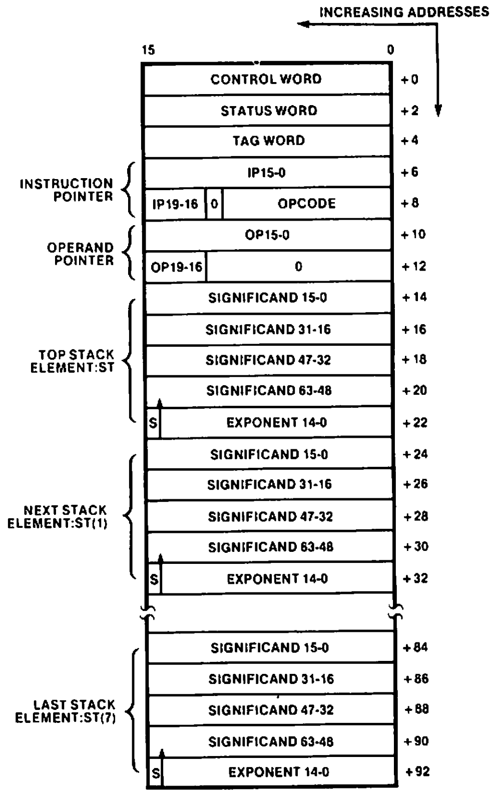
Description

Figure 6-8 shows the 94-byte save area layout. Typically, FSAVE/FNSAVE will be coded to save this image on the CPU stack.

If an instruction is executing in the 8087 when FNSAVE is decoded, the CPU queues the save and delays its execution until the running instruction completes normally, or encounters an unmasked exception. The save image, therefore, reflects the state of the 8087 following completion of any running instruction. After writing the state image to memory, FSAVE/FNSAVE initializes the 8087 as if FINIT/FNINT had been executed.

FSAVE/FNSAVE is useful whenever a program wants to save the current state of the NDP and initialize it for a new routine. Three examples are:

1. An operating system needs to perform a context switch (suspend the task that has been running and give control to a new task);
2. An interrupt handler needs to use the 8087;
3. An application task wants to pass a "clean" 8087 stack to a sub-routine.



NOTES:
S = Sign
Bit 0 of each field is rightmost, least significant bit of corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

Figure 6-8. FSAVE/FRSTOR Memory Layout 121623-15

Note

FSAVE/FNSAVE, like FSTENV/FNSTENV, must be protected from any other 8087 instruction that might execute while the save is in progress. When FSAVE is coded, this can be insured by placing an explicit FWAIT in front of a subsequent no-wait mnemonic, if there is one. When FSAVE is executed with CPU interupts disabled, an FWAIT should be executed before CPU interrupts are enabled or any subsequent 8087 instruction is executed. Because the FNSAVE initializes the NDP, there is no danger of the FWAIT causing an endless wait. Other CPU instructions may be executed between the FNSAVE and the FWAIT; this will reduce interrupt latency if the FNSAVE is queued in the 8087.

Scale

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FD	CD 19 FD	35 32-38	ST ← ST • $2^{ST(1)}$	FSCALE

Operation

This instruction interprets the value contained in ST(1) as an integer, and adds this value to the exponent of the number in ST. ST(1) must be in the range $-2^{15} \leqslant ST(1) < +2^{15}$ and ST(1) must be an integer.

Exceptions

I	Z	D	O	U	P
X			X	X	

Description

FSCALE is particularly useful for scaling the elements of a vector because it provides rapid multiplication or division by integral powers of 2.

Note

FSCALE assumes the scale factor in ST(1) is an integral value in the range $-2^{15} \leqslant x < 2^{15}$. If the value is not an integer, but is in-range and is greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude, i.e., it chops the value toward 0. If the value is out of range, or $0 < | x | < 1$, the instruction will produce an undefined result and will not signal an exception. The recommended practice is to load the scale factor from a word integer to ensure correct operation.

FSQRT

Square Root

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FA	CD 19 FA	183 180-186	ST ← $\sqrt{ST}$	FSQRT

Operation

This instruction replaces the contents of the top of the stack with its square root. ST must be in the range $-0 \leqslant ST \leqslant +\infty$.

Exceptions

I	Z	D	O	U	P
X		X			X

Store Real

Format

Stack top to Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD D0 + i	CD 1D D0 + i	18 15-22	ST(i) ← ST	FST ST(4)

Stack top to memory operand

WAIT	op1	m/op/rm	[addr1]	[addr2]

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m2rm	CD 19 m2rm	87 + EA (84-90) + EA	mem-op ← ST (short-real)	FST MEAN
9B DD m2rm	CD 1D m2rm	100 + EA (96-104) + EA	mem-op ← ST (long-real)	FST READING

Operation

The store real instruction transfers the top of the stack to the destination, which may be another stack element or a short or long real memory operand. If the destination is short or long real, the significand is rounded to the width of the destination according to the RC field of the control word and the exponent is converted to the width and bias of the destination format.

Exceptions

I	Z	D	O	U	P
X			X	X	X

Note

If the stack top is tagged special (it contains ∞, a NAN, or a denormal), the stack top significand is not rounded. In this case, the least significant bits of the stack top are deleted to fit the destination. The exponent is treated in the same way. This preserves the value's identification as ∞, or a NAN (exponent of all ones), or a denormal (exponent all zeros) so that it can be properly loaded and tagged later in the program, if desired.

FSTCW
FNSTCW

Store Control Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m7rm	CD 19 m7rm	15 + EA (12-18) + EA	mem-op ← processor control word	FSTCW CONTROL
90 D9 m7rm	CD 19 m7rm	15 + EA (12-18) + EA	mem-op ← processor control word (no wait)	FNSTSW CONTROL

Operation

The store control word instructions write the current processor control word to the memory location defined by the destination. The FSTCW form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

When application tasks are running, the WAIT form of this instruction should be used. The NO WAIT form is provided for use in critical code regions where a WAIT instruction might induce an endless wait.

Store Environment

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m6rm	CD 19 m6rm	45 + EA (40-50) + EA	mem-op ← 8087 environment	FSTENV ENVIRON
90 D9 m6rm	CD 19 m6rm	45 + EA (40-50) + EA	mem-op ← 8087 environment (no wait)	FNSTENV ENVIRON

Operation

This instruction writes the 8087 basic status (control word, status word, and tag word) and exception pointers to the memory location defined by the destination operand. The FSTENV form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

FSTENV/FNSTENV is often used by exception handlers because it provides access to the exception pointers which identify the offending instruction and operand.

FSTENV/FNSTENV typically saves the environment on the CPU stack. After the environment is saved, FSTENV/FNSTENV sets all exception masks in the processor; it does not affect the interrupt enable mask. Figure 6-9 shows the format of the environment data in memory. If FNSTENV is decoded while another instruction is executing concurrently in the NEU, the 8087 does not store the environment until the other instruction has completed. The data saved by this instruction, therefore, reflects the state of the 8087 AFTER any previously decoded instruction has been executed.

Note

FSTENV/FNSTENV must be allowed to complete before any other 8087 instruction is decoded. When FSTENV is coded, an assembler-generated WAIT should precede any subsequent 8087 instruction. When using FNSTENV, with CPU interrupts disabled, an explicit FWAIT should be executed before enabling CPU interrupts.

There is no risk of the FWAIT causing an endless wait. FNSTENV masks all exceptions so that interrupt requests from the 8087 are prevented.

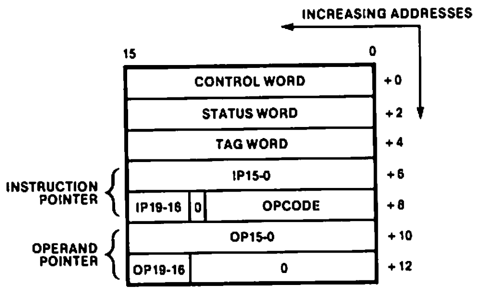


Figure 6-9. FSTENV and FLDENV Memory Layouts 121623-16

Store Real and Pop

Format

Stack top to Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD D8 + i	CD 1D D8 + i	20 17-24	ST(i) ← ST pop stack	FSTP ST(2)

Stack top to memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m3rm	CD 19 m3rm	89 + EA (86-92) + EA	mem-op ← ST pop stack (short-real)	FSTP TOTAL
9B DD m3rm	CD 1D m3rm	102 + EA (98-106) + EA	mem-op ← ST pop stack (long-real)	FSTP AVERAGE
9B DB m7rm	CD 1B m7rm	55 + EA (52-58) + EA	mem-op ← ST pop stack (temp-real)	FSTP TEMP_STORE

Operation

The store real and pop stack instruction transfers the top of the stack to the destination and then pops the stack. The destination may be another stack element, or memory operand (short-real, long-real, or temporary-real). If the destination is short or long real memory, the significand is rounded to the width of the destination according to the RC field of the control word and the exponent is converted to the width and bias of the destination format.

This instruction allows storing temporary real numbers into memory. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

Exceptions

I	Z	D	O	U	P
X			X	X	X

FSTSW
FNSTSW

Store Status Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m7rm	CD 1D m7rm	15 + EA (12-18) + EA	mem-op ← 8087 status word	FSTSW SAVE_STAT
90 DD m7rm	CD 1D m7rm	15 + EA (12-18) + EA	mem-op ← 8087 status word (no wait)	FNSTSW SAVE STAT

Operation

The store status word instructions write the current value of the 8087 status word to the destination operand in memory. The FSTSW form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

The three primary uses of this instruction are:

1. To implement conditional branching following a comparison or FPREM instruction (WAIT form).
2. To poll the 8087 to determine if it is busy (NO-WAIT form).
3. To invoke exception handlers in environments that do not use interrupts (WAIT form).

Note

If the WAIT form is used with an outstanding unmasked exception, deadlock will result.

Subtract Real

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E0 + i	CD 18 E0 + i	85 70-100	ST ← ST − ST(i)	FSUB ST,ST(2)
9B DC E8 + i	CD 1C E8 + i	85 70-100	ST(i) ← ST(i) − ST	FSUB ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m4rm	CD 18 m4rm	105 + EA (90-120) + EA	ST ← ST − mem-op (short-real)	FSUB VALUE
9B DC m4rm	CD 1C m4rm	110 + EA (95-125) + EA	ST ← ST − mem-op (long-real)	FSUB BASE

Operation

The subtract real instruction subtracts the source operand from the destination and returns the difference to the destination. The source operand may be either the stack top, a stack element or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FSUBP

Subtract Real and Pop

Format

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E8 + i	CD D8 E8 + i	90 75-105	ST(1) ← ST(1) - ST pop stack	FSUB
9B DE E8+i	CD 1E E8+i	90 75-105	ST(i) ← ST(i) - ST pop stack	FSUBP ST(2),ST

Operation

The subtract real instruction subtracts the source operand from the destination and returns the difference to the destination. The source operand may be either the stack top, a stack element or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The subtract real and pop stack instruction subtracts the stack top from one of the stack elements, replacing the stack element with the difference and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

Subtract Real Reversed

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E8 + i	CD D8 E8 + i	87 70-100	ST ← ST(i) - ST	FSUBR ST,ST(i)
9B DC E0 + i	CD 1C E0 + i	87 70-100	ST(i) ← ST - ST(i)	FSUBR ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m5rm	CD 18 m5rm	105 + EA (90-120) + EA	ST ← mem-op - ST (short-real)	FSUBR INDEX
9B DC m5rm	CD 1C m5rm	110 + EA (95-125) + EA	ST ← mem-op - ST (long-real)	FSUBR VECTOR

Operation

The reverse subtract instruction subtracts the destination from the source and returns the difference to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

FSUBRP

Subtract Real Reversed and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE E1	CD 1E E1	90 75-105	ST(1) ← ST - ST(1) pop stack	FSUBR
9B DE E0 + i	CD 1E E0 + i	90 75-105	ST(i) ← ST - ST(i) pop stack	FSUBRP ST(2),ST

Operation

The reverse subtract instruction subtracts the destination from the source and returns the difference to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse subtract and pop stack instruction subtracts one of the stack elements from the stack top and returns the difference to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

Test Stack Top Against +0.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E4	CD 19 E4	42 38-48	ST ← ST - 0.0	FTST

Operation

The test instruction compares the element in the top of the floating point stack with zero and posts the result to the condition code.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Condition Code Test Results

C3	C0	Result
0	0	ST is positive
0	1	ST is negative
1	0	ST is zero (+ or −)
1	1	ST is not comparable (i.e., it is a NAN or projective ∞)

FWAIT

(CPU) Wait while 8087 is busy

Format

WAIT

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B	90	3+5n 3+5n	8086 wait instruction	FWAIT

Operation

This instruction is an alternate mnemonic for the CPU WAIT instruction. FWAIT must be used instead of WAIT for 8087 emulator compatability is desired.

Exceptions

I Z D O U P

Description

The FWAIT mnemonic should be coded whenever the programmer wants to synchronize the CPU to the NDP. This means that further instruction decoding will be suspended until the NDP has completed the current instruction. This is useful if the CPU wants to inspect a value stored by the NDP (i.e., FIST should be followed by FWAIT to ensure that the value has been stored before attempting to examine it).

Note

Programmers should not code WAIT to synchronize the CPU and 8087. The routines that alter an object program for 8087 emulation change any FWAITs to NOPs but do not change any explicitly coded WAITs. The program will wait forever if a WAIT is encountered in emulated execution since there is no 8087 to drive the CPU's test pin active.

Examine Stack Top

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E5	CD 19 E5	17 12-23	set condition code	FXAM

Operation

The examine instruction reports the content of the top of the floating point stack as positive/negative and NAN/unnormal/denormal/normal/zero, or empty. The condition codes which can be generated are shown in table 6-15.

Exceptions

I Z D O U P

Description

Table 6-15 lists and interprets all of the condition code values that FXAM generates. Although four different encodings may be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all encodings. Bits C2 an C1 should be ignored when examining for empty.

Table 6-15. FXAM Condition Code Settings

Condition Code				Interpretation
C3	C2	C1	C0	
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	+ ∞
0	1	1	0	− Normal
0	1	1	1	− ∞
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

FXCH

Exchange Registers

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 C8	CD 19 C8	12 10-15	$T_1 \leftarrow ST(1)$ $ST(1) \leftarrow ST$ $ST \leftarrow T_1$	FXCH
9B D9 C8+i	CD 19 C8+i	12 10-15	$T_1 \leftarrow ST(i)$ $ST(i) \leftarrow ST$ $ST \leftarrow T_1$	FXCH ST(3)

Operation

The exchange instruction swaps the contents of a stack element and the stack top. If the stack element is not explicitly coded, ST(1) is used.

Exceptions

I	Z	D	O	U	P
X					

Description

Many 8087 instructions operate only on the stack top; FXCH provides an easy way to use these instructions on lower stack elements. For example, the following sequence takes the square root of the third element from the top.

```
FXCH    ST(3)
FSQRT
FXCH    ST(3)
```

Extract Exponent and Significand

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F4	CD 19 F4	50 27-55	$T_1 \leftarrow$ exponent (ST) $T_2 \leftarrow$ significand (ST) ST $\leftarrow T_1$ push stack ST $\leftarrow T_2$	FXTRACT

Operation

The extract instruction "decomposes" the number in the stack top into two numbers that represent the actual value of the operand's exponent and significand fields. The "exponent" replaces the original operand on the stack and the "significand" is pushed onto the stack.

Exceptions

I	Z	D	O	U	P
X					

Description

FXTRACT is useful in conjunction with FBSTP for converting numbers in 8087 temporary real format to decimal representations (e.g., for printing or displaying). It can also be useful for debugging, since it allows the exponent and significand parts of a real number to be examined separately.

Note

Following execution of FXTRACT, ST (the new stack top), contains the value of the original significand expressed as a real number. The sign of this number is the same as the operand's; its exponent is 0 true (16,383 or 3FFFH biased), and its significand is identical to the original operand's. ST(1) contains the value of the original operand's true (unbiased) exponent expressed as a real number. If the original operand is zero, FXTRACT produces zeros in ST and ST(1) and BOTH are signed as the original operand.

Example

Assume ST contains a number whose true exponent is +4 (i.e., its exponent field contains 4003H). After executing FXTRACT, ST(1) will contain the real number +4.0; its sign will be positive, its exponent field will contains 4001+) (+2 true) and its significand field will contain 1Δ00 . . . 00B. In other words, the value in ST(1) will be $1.0x2^2 = 4$.

If ST contains an operand whose true exponent is −7 (i.e., its exponent field contains 3FF8H), then FXTRACT will return an "exponent" of −7.0. After the instruction executes, ST(1)'s sign and exponent fields will contains C001H (negative sign, true exponent of 2) and its significand will be 1Δ1100 . . . 00B. The value in ST(1) will be $-1.11x2^2 = -7.0$.

In both cases, following FXTRACT, ST's sign and significand fields will be the same as the original operand's and its exponent field will contain 3FFFH, (0 true).

$Y * \text{Log}_2 X$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F1	CD 19 F1	950 900-1100	$ST_1 \leftarrow ST(1) \cdot \log_2(ST)$ pop stack $ST \leftarrow T_1$	FYL2X

Operation

This instruction calculates the function:

$$Z = Y \cdot \log_2 X$$

X is taken from the stack top and Y from ST(1). The operands must be in the ranges $0 < X < \infty$ and $-\infty < Y < +\infty$. The instruction pops the stack and returns Z at the (new) stack top replacing the Y operand.

Exceptions

I	Z	D	O	U	P	*
					X	

*operands not checked

Note

This function optimizes the calculation of log to any base other than two since a multiplication is always required:

$$\log_n X = \frac{1}{\log_2 n} \cdot \log_2 X$$

FYL2XP1

Y * Log_2 (X + 1)

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F9	CD 19 F9	850 700-1000	$T_1 \leftarrow ST + 1$ $T_2 \leftarrow ST(1) * \log_2 T_1$ pop stack $ST \leftarrow T_2$	FYL2XP1

Operation

This instruction calculates the function $Z = Y*LOG_2 (X + 1)$. X is taken from the stack top and must be in the range $0 < | X | < (1- \sqrt{2}/2)$. Y is taken from ST(1) and must be in the range $-\infty\Delta < Y < \infty$. FYL2XP1 pops the floating point stack and returns Z at the new stack top, replacing Y.

Exceptions

I	Z	D	O	U	P	*
					X	

*operands not checked

Note

This instruction provides improved accuracy over FYL2X when computing the log of a number very close to 1. For example, when calculating 1 + E where E << 1, being able to input E rather than 1 + E to the function allows more significant digits to be retained.

Assembling

Chapter 7

Assembling an MS-Assembler Source File

Assembling an MS-Assembler Source File

Assembling a program with MS-Assembler requires two types of commands: a command to start MS-Assembler, and answers to command prompts. In addition, four switches control alternate MS-Assembler features. Usually, you will type all the commands to MS-Assembler on the computer keyboard. As an option, answers to the command prompts and any switches may be contained in a response (batch) file. Two command characters are provided to assist you while entering assembler commands. These command characters are described in Section 7.2, "Command Characters."

7.1 How to Start MS-Assembler

MS-Assembler may be started in two ways. By the first method, you type the commands in response to individual prompts. By the second method, you type all commands on the line used to start MS-Assembler.

7.1.1 Method 1: Prompts

Type:

```
MASM
```

MS-Assembler is loaded into memory. Then, the MS-Assembler displays a series of four prompts. You answer the prompts as commands to the MS-Assembler to perform specific tasks.

At the end of each line, you may specify one or more switches, each of which must be preceded by a forward slash (/).

The command prompts are summarized here and described in more detail in Section 7.3, "MS-Assembler Command Prompts."

Summary of Command Prompts

PROMPT	RESPONSES
Source filename [.ASM]:	Enter List .ASM file to be assembled. (There is no default; a filename response is required.)
Object filename [source.OBJ]	Enter List filename for relocatable object code. (The default is source-filename.OBJ)
Source listing [NUL.LST]:	Enter List filename for listing. (The default is no listing file.)
Cross reference [NUL.CRF]:	Enter List filename for cross-reference file (used with MS-CREF to create a cross-reference listing). (The default is no cross-reference file.)

7.1.2 Method 2: Command Line

Type:

```
MASM <source>,<object>,<listing>,<cross-ref>
  [/switch . . . ]
```

where the entries following MASM are responses to the command prompts:

source is the source filename

object is the name of the file to receive the relocatable output

listing is the name of the file to receive the listing

cross-ref is the name of the file to receive the cross-reference output

/switch are optional switches, which may be placed following any of the response entries (just before any of the commas or after the <cross-ref>, as shown).

To select the default for a field, simply enter a second comma without space in between (see the example below).

Once you have entered the command line, MS-Assembler is loaded into memory. Then MS-Assembler immediately begins assembly.

Example:

```
MASM FUN, ,FUN/D/X,FUN
```

This example causes MS-Assembler to be loaded and the source file FUN.ASM to be assembled. MS-Assembler then outputs the relocatable object code to a file named FUN.OBJ (default caused by two commas in a row), creates a listing file named FUN.LST for both assembly passes but with false conditionals suppressed, and creates a cross-reference file named FUN.CRF. If listing file switches are given but no filename, the switches are ignored.

7.2 MS-Assembler Command Characters

MS-Assembler provides two command characters.

Semicolon

Use a single semicolon (;), followed immediately by a carriage return, at any time after responding to the first prompt (from Source filename: on) to select default responses to the remaining prompts. This feature saves time and eliminates the need to enter a series of carriage returns.

NOTE

Once the semicolon has been entered, you can no longer respond to any of the prompts for that assembly. Therefore, do not use the semicolon to skip over some prompts. For this, use the ENTER key.

Example:

```
Source filename [.ASM]: FUN
Object filename [FUN.OBJ]: ;
```

The remaining prompts will not appear, and MS-Assembler will use the default values (including no listing file and no cross-reference file).

To achieve the same result, you could type:

```
Source filename [.ASM]: FUN ;
```

This response produces the same files as the previous example.

CONTROL-C

Use CONTROL C at any time to abort the assembly. If you enter an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press CONTROL C to exit MS-Assembler. You can then restart MS-Assembler. If the error has been typed but not entered, you may delete the erroneous characters for that line only.

7.3 MS-Assembler Command Prompts

You give commands to MS-Assembler by entering responses to four text prompts. When you have answered the last prompt, MS-Assembler begins assembly automatically. When assembly is finished, MS-Assembler exits to the operating system. MS-Assembler has finished successfully when the operating system prompt appears. If the assembly is unsuccessful, MS-Assembler displays the appropriate error message.

MS-Assembler prompts you for the names of source, object, listing, and cross-reference files.

All command prompts accept a file specification as a response. You may type:

- A filename only
- A device designation only
- A filename and an extension
- A device designation and filename, or
- A device designation, filename, and extension.

Do not type only a filename extension.

The following is a discussion of the command prompts that are displayed when you start MS-Assembler with Method 1:

Source filename [.ASM]:

Type the filename of your source program. MS-Assembler assumes by default that the filename extension is .ASM, as shown in square brackets in the prompt text. If your source program has any other filename extension, do not enter the extension. Otherwise, also omit the extension.

Object filename [source.OBJ]:

Type the filename you want to receive the generated object code. If you simply press the ENTER key when this prompt appears, the object file will be given the same name as the source file, but with the filename extension .OBJ. Do not enter an extension.

Source listing [NUL.LST]:

Type the name of the file you want to receive the source listing. If you simply press the ENTER key when this prompt appears, MS-Assembler does not produce this listing file. If you type a filename only, the listing is created and placed in a file with the name you type plus the filename extension .LST.

The source listing file will contain a list of all the statements in your source program and will show the code and offsets generated for each statement. The listing will also show any error messages generated during the session.

Cross reference [NUL.CRF]:

Type the name of the file you want to receive the cross-reference file. If you press only the ENTER key, MS-Assembler does not produce this cross-reference file. If you type a filename only, the cross-reference file is created and given the name you type plus the filename extension .CRF.

The cross-reference file is used as the source file for the Cross-Reference Utility (MS-CREF). MS-CREF converts this cross-reference file into a cross-reference listing, which you can use to aid you during program debugging.

The cross-reference file contains a series of control symbols that identify records in the file. MS-CREF uses these control symbols to create a listing that shows all occurrences of every symbol in your program. The occurrence that defines the symbol is also identified.

7.4 MS-Assembler Command Switches

The three MS-Assembler switches control assembler functions. Switches must be typed at the end of a prompt response, regardless of which method is used to start MS-Assembler. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is typed at the end of one response, each switch must be preceded by a forward slash (/). Do not specify only a switch as a response to a command prompt.

Switch	Function
/D	Produces a source listing on both assembler passes. The listings will, when compared, show where in the program phase errors occur and can possibly give you a clue to why the errors occur.
/O	Outputs the listing file in octal radix. The generated code and the offsets shown on the listing will all be given in octal. The actual code in the object file will be the same as when the /O switch is not given. The /O switch affects only the listing file.
/X	Suppresses the listing of false conditionals. If your program contains conditional blocks, the listing file will show the source statements, but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not generate code, use the /X switch to suppress the blocks that evaluate false from your listing.

The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives.

If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, the second restores listing of false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When MS-Assembler encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing.

Of course, the /X switch has no effect if no listing is created. See additional discussion under the .TFCOND directive in Section 4.2.4, "Listing Directives."

The following chart illustrates the various effects of the conditional listing directives in combination with the /X switch.

Pseudo-op	*No /X*	*/X*
(none) ON	OFF	
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

Summary of Command Switches

SWITCH	ACTION
/D	Produce a listing on both assembler passes.
/O	Show generated object code and offsets in octal radix on listing.
/X	Suppress the listing of false conditionals. Also used with the .TFCOND directive.

7.5 Formats Of Listings and Symbol Tables

The source listing produced by MS-Assembler (created when you specify a filename in response to the Source listing: prompt) is divided into two parts.

The first part of the listing shows:

- The line number for each line of the source file, if a cross-reference file is also being created.
- The offset of each source line that generates code.
- The code generated by each source line.
- A plus sign (+), if the code came from a macro, or a letter C, if the code came from an INCLUDE file.
- The source statement line.

The second part of the listing shows:

- Macros—name and length in bytes
- Structures and records—name, width and fields
- Segments and groups—name, size, align, combine, and class
- Symbols—name, type, value, and attributes
- The number of warning errors and severe errors

7.5.1 Program Listing

The program portion of the listing is essentially your source program file with the line numbers, offsets, generated code, and (where applicable) a plus sign to indicate that the source statements are part of a macro block, or a letter C to indicate that the source statements are from a file input by the INCLUDE directive.

If any errors occur during assembly, the error message is printed directly below the statement where the error occurred.

Part of a listing file follows this discussion, with notes explaining what the various entries represent.

The comments have been moved down one line because of format restrictions. If you print your listing on 132-column paper, use the page directive here so that the comments shown will easily fit on the same line as the rest of the statement.

Explanatory notes are spliced into the listing at points of special interest.

Summary of Listing Symbols

R	= Linker resolves entry to left of R
E	= External
----	= Segment name, group name, or segment variable used in MOV AX,<——>, DD <——>, JMP <——>, and so on.
=	= Statement has an EQU or = directive
nn:	= Statement contains a segment override
nn/	= REPxx or LOCK prefix instruction. Example:

```
003C    F3/ A5        REP    MOVSW
                             ;move DS:SI to ES:DI
                             ;until CX=0
```

[xx]	= DUP expression;xx is the value in parentheses following DUP; for example: DUP(?) places ?? where xx is shown here
+	= Line comes from a macro expansion
C	= Line comes from file named in INCLUDE directive statement

MS-Assembler 1-Dec-81 PAGE 1-3

EXTX PASCAL entry for initializing programs

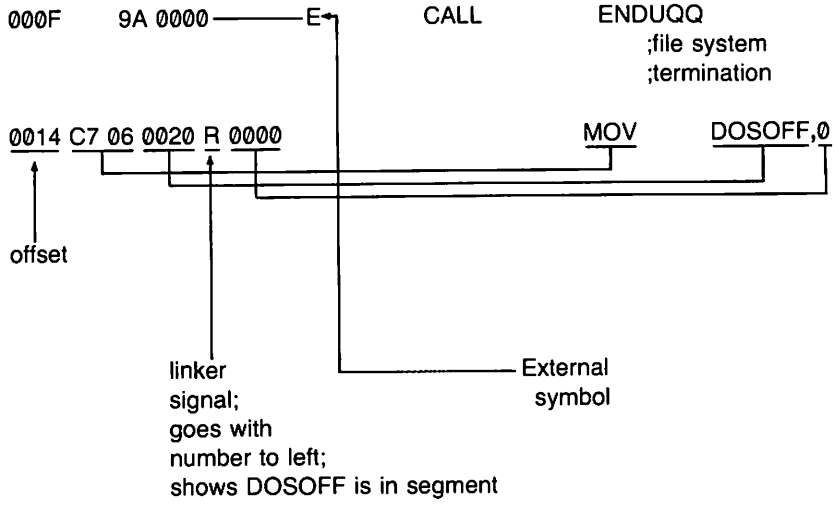
```
                                                     ;
0000                STACK            SEGMENT     WORD STACK       'STACK'
= 0000              HEAPbeg          EQU         THIS BYTE
└──────Indicates EQU or = directive
                                             ;Base of heap before init
0000          14 [                   DB          20 DUP (?)←┐
                 ?? ← Shows value in parentheses ────────────┘
                 ]
                    Indicates DUP expression
= 0014              SKTOP            EQU         THIS BYTE
0014                STACK            ENDS

0000                MAINSTARTUP      SEGMENT                      'MEMORY'
                    DGROUP           GROUP       DATA,STACK<CONST,HEAP,
                                                 MEMORY
                                     ASSUME      CS:MAINSTARTUP,DS:
                                                 DGROUP,ES:DGROUP,SS:
                                                 DGROUP

                                     PUBLIC BEGXQQ ;Main entry

                                                  ;
0000                BEGXQQ           PROC        FAR
0000    B8 ----     R                MOV         AX,DGROUP
                                                 ;Get data segment value
0003    8E D8                        MOV         DS,AX ;Set DS seg

0005    8C 06 0022 R                 MOV         CESXQQ,ES

Generated  Name                   Action         Expression          Comment
Offset
                                                                        .
                                                                        .
                                                                        .

000C    26: 8B 1E 0002               MOV         BX,ES:2   ;Highest
        ┬                                           ┬        ;paragraph
        └──────────────────── Segment override ─────┘
```

```
MS-Assembler 1-Dec-81 PAGE 1-4

ENTX PASCAL entry for initializing programs

0011    2B D8                SUB     BX,AX       ;Get # paras for DS
0013    81 FB 1000           CMP     BX,4096     ;More than 64K?
0017    7E 03                JLE     SMLSTK      ;No, use what we have
0019    BB 1000              MOV     BX,4096     ;Can only address 64k

001C                      SMLSTK:      REPT          4
                                 SHL     BX,1
                                     ;Convert para to offset
                                 ENDM

001C    D1 E3                    SHL     BX,1
                                     ;Convert para to offset
001E    D1 E3                    SHL     BX,1
                                     ;Convert para to offset
0020    D1 E3                    SHL     BX,1
                                     ;Convert para to offset
0022    D1 E3                    SHL     BX,1
                                     ;Convert para to offset
                                   V
macro      these lines          macro                       number of
block      from macro           directive                   repetitions

0024    8B E3           MOV      SP,BX
                        ;Set stack to top of memory
                        --
                        --
                                 --
0069    EA 0000 ------- R        JMP     FAR PTR STARTmain
                        signal to linker       segment variable

        linker resolves: indicates segment name, group name, or segment variable used in MOV
        AX, <-----> ; DD <-----> ; JMP <-----> ,etc. (See other examples in this listing.)

006E         BEGXQQ               ENDP
                                  --
                                  --
                                  --
007E         MAIN_STARTUP         ENDS

0000         ENTXCM               SEGMENT WORD 'CODE'
                   ASSUME         CS:ENTXCM
                   PUBLIC
                   ENDXQQ,DOSXQQ
```

MS-Assembler 1-Dec-81 PAGE 1-5

ENTX PASCAL entry for initializing programs

```
0000            STARTmain       PROC        FAR
                                ;This code remains
0000    9A 0000———E             CALL        ENTGQQ
                                ;call main program
                ;
0005            ENDXQQ          LABEL       FAR
                                ;termination entry point
0005    9A 0000———E             CALL        ENDOQQ
                                ;user system termination
000A    9A 0000——— E            CALL        ENDYQQ
                                ;close all open files
000F    9A 0000———E             CALL        ENDUQQ
                                                ;file system
                                                ;termination
```



```
0014 C7 06 0020 R 0000                      MOV        DOSOFF,0
```

offset
linker signal; goes with number to left; shows DOSOFF is in segment
External symbol

```
00 2E 0020 R                    JMP         DWORD PTR DOSOFF
                                            ;return to DOS
001E            STARTmain       ENDP

                                -.
                                -.
                                -.
0037            ENTXCM          ENDS

                        END     BEGXQQ
```

7.5.2 Differences Between Pass 1 And Pass 2 Listings

If you specify the /D switch when you run MS-Assembler to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that assembled without reporting any errors. When the source file was reassembled using the /D switch, an error was produced on pass 1, but not on pass 2 (which is when errors are usually reported).

Example:

During Pass 1 a jump with a forward reference produces:

```
0017  7E 00                       JLE   SMLSTK  ;No, use what we have
E r r o r  ---                    9:Symbol not defined
0019  BB 1000
                                  MOV  BX,4096   ;Can only address 64k
001C    SMLSTK: REPT              4
```

During Pass 2 this same instruction is fixed up and does not return an error.

```
0017  7E 03                       JLE   SMLSTK  ;No, use what we have
0019  BB 1000                     MOV  BX,4096   ;Can only address 64k
001C    SMLSTK: REPT              4
```

Notice that the JLE instruction's code now contains 03 instead of 00; this is a jump of 3 bytes.

The same amount of code was produced during both passes, so there was no phase error. The only difference in this case is one of content instead of size.

7.5.3 Symbol Table Format

The symbol table portion of a listing separates all "symbols" into their respective categories, showing appropriate descriptive data. This data gives you an idea of how your program is using various symbolic values, and is useful when you debug.

Also, you can use a cross-reference listing, produced by MS-CREF, to help you locate uses of the various "symbols" in your program.

On the next page is a complete symbol table listing. Following the complete listing, sections from different symbol tables are shown with explanatory notes.

For all sections of symbol tables, this rule applies: if there are no symbolic values in your program for a particular category, the heading for the category will be omitted from the symbol table listing. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

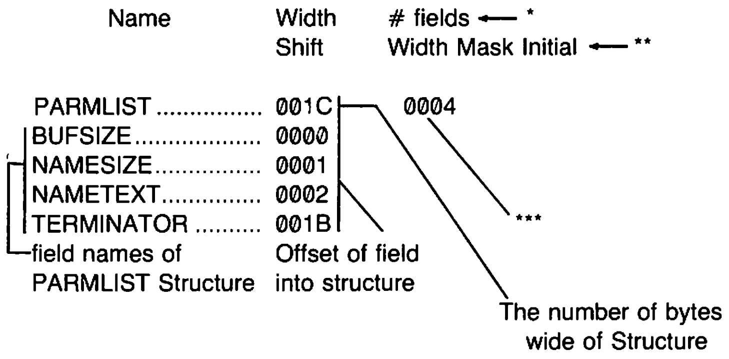
```
MS-Assembler MACRO
Assembler     date     PAGE     Symbols-1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXMP1M.ASM)

Macros:

              Name           Length

BIOSCALL ....................  0002
DISPLAY .....................  0005
DOSCALL .....................  0002
KEYBOARD ....................  0003
LOCATE ......................  0003
SCROLL ......................  0004

Structures and records:

              Name           Width      # fields
                             Shift      Width      Mask       Initial

PARMLIST ....................  001C       0004
  BUFSIZE ...................  0000
  NAMESIZE ..................  0001
  NAMETEXT ..................  0002
  TERMINATOR ................  001B

Segments and groups:

              Name           Size       align      combine    class

CSEG ........................  0044       PARA       PUBLIC     'CODE'
STACK .......................  0200       PARA       STACK      'STACK'
WORKAREA ....................  0031       PARA       PUBLIC     'DATA'
```

```
Symbols:

        Name            Type        Value       Attr

CLS........................ N PROC      0036        CSEG        Length =000E
MAXCHAR............ Number      0019
MESSG.................. L BYTE      001C        WORKAREA
PARMS.................. L 001C      0000        WORKAREA
RECEIVR.............. L FAR       0000                    External
START.................. F PROC      0000        CSEG        Length =0036

Warning   Severe
Errors    Errors
0         0

Macros:

        Name            Length<———————number of 32-byte blocks
                                      macro occupies in memory
BIOSCALL............ 0002
DISPLAY............... 0005
DOSCALL.............. 0002
KEYBOARD........... 0003
LOCATE................ 0003
SCROLL................ 0004
   ↑
   |
names of macros
```

This section of the symbol table tells you the names of your macros and how big they are in 32-byte block units. In this listing, the macro DISPLAY is 5 blocks long or (5 X 32 bytes =) 160 bytes long.

Structures and records:

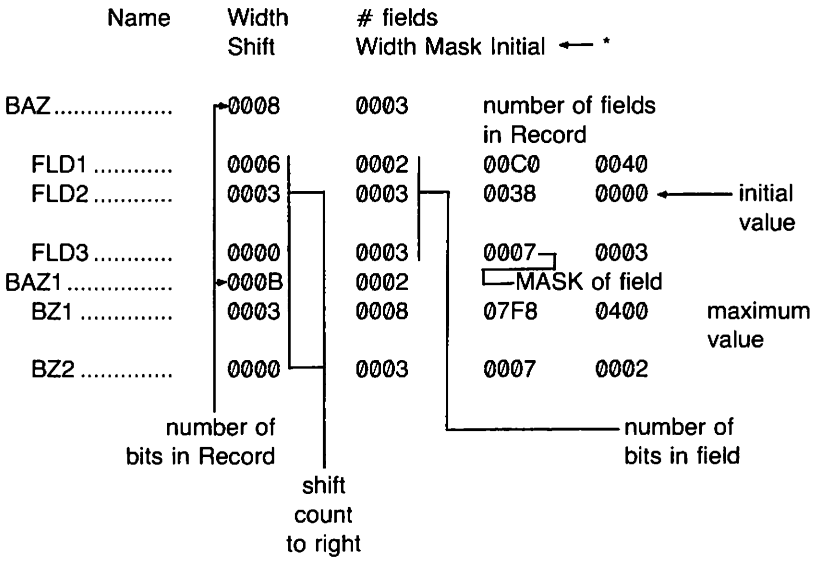
Example for Structures



```
        Name                Width    # fields  <-- *
                            Shift    Width Mask Initial  <-- **

    PARMLIST ............. 001C       0004
    BUFSIZE .............. 0000
    NAMESIZE ............. 0001
    NAMETEXT ............. 0002
    TERMINATOR ........... 001B
```

***
field names of PARMLIST Structure
Offset of field into structure
The number of bytes wide of Structure

Example for Records



```
        Name          Width    # fields
                      Shift    Width Mask Initial  <-- *

BAZ .................. 0008    0003
  FLD1 ............... 0006    0002     00C0    0040
  FLD2 ............... 0003    0003     0038    0000
  FLD3 ............... 0000    0003     0007    0003
BAZ1 ................. 000B    0002
  BZ1 ................ 0003    0008     07F8    0400
  BZ2 ................ 0000    0003     0007    0002
```

*	This line applies to Structure Names (begin in column 1).
**	This line for fields of Records (indented).
***	Number of fields in Structure.

This section lists your Structures and/or Records and their fields. The upper line of column headings applies to Structure names, Record names, and field names of Structures. The lower line of column headings applies to field names of Records.

For Structures

Width (upper line) shows the number of bytes your Structure occupies in memory.

# fields shows how many fields comprise your Structure.

For Records:

Width (upper line) shows the number of bits the Record occupies.

# fields shows how many fields comprise your Record.

For Fields of Structures:

Shift shows the number of bytes the fields are offset into the Structure.

The other columns are not used for fields of Structures.

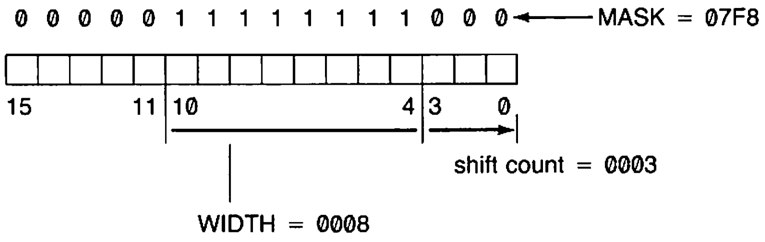
For Fields of Records:

Shift is the shift count to the right.

Width (lower line) shows the number of bits this field occupies.

Mask shows the maximum value of the record, expressed in hexadecimal, if one field is masked and ANDed (the field is set to all 1's and all other fields are set to all 0's).

Using field BZ1 of the Record BAZ1 above to illustrate:



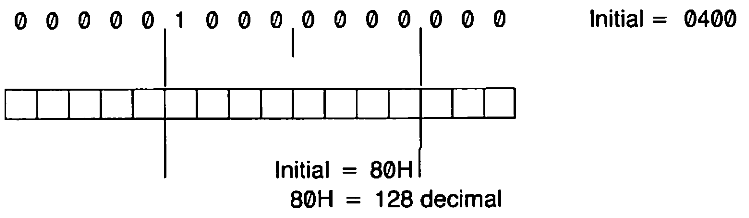
Initial shows the value specified as the initial value for the field, if any.

When naming the field, you specified:
fieldname:# = value

Fieldname is the name of the field

is the width of the field in bits

Value is the initial value you want this field to hold. The symbol table shows this value as if it is placed in the field and all other fields are masked (equal 0). Using the example and diagram from above:



Segments and groups:

```
              Name                 Size     align          combine class
                                                      ┌────called Private
                                                      /   in MS-LINK manual
AAAXQQ.......................... 0000     WORD     NONE        'CODE' ←────── segment
DGROUP.......................... GROUP ←──────────────────────────────── group
  DATA.......................... 0024     WORD     PUBLIC      'DATA'     │
  STACK......................... 0014     WORD     STACK       'STACK'    │ segments
  CONST......................... 0000     WORD     PUBLIC      'CONST'    │   of
  HEAP.......................... 0000     WORD     PUBLIC      'MEMORY'   │   DGROUP
  MEMORY........................ 0000     WORD     PUBLIC      'MEMORY'   │
ENTXCM.......................... 0037     WORD     NONE        'CODE'
MAIN_STARTUP.................... 007E     PARA     NONE        'MEMORY'
                                 — — — — — — — — — — — — — — —
                                 length          statement line entries
                                   of
                                 segment
```

For Groups:

The name of the group appears under the Name column, beginning in column 1 with the applicable Segment names indented 2 spaces. The word Group appears under the Size column.

For Segments:

The segment names may appear in column 1 (as here) if you do not declare them part of a group. If you declare a group, the segment names appear indented under their group name.

For all Segments, whether a part of a group or not:

Size is the number of bytes the Segment occupies.

Align is the type of boundary where the segment begins:
PAGE = page - address is xxx00H (low byte = 0); begins on a 256-byte boundary

PARA	= paragraph - address is xxxx0H (low nibble = 0); default
WORD	= word - address is xxxxeH (e = even number; low bit of low byte = 0) bit map - |x|x|x|x|x|x|x|0|
BYTE	= byte - address is xxxxxH (anywhere)

Combine describes how the LINK utility will combine the various segments. (See the description of the LINK utility in the *MS-DOS Commands Reference Manual.*)

Class is the class name under which MS-LINK will combine segments in memory. (See the description of the LINK utility in the *MS-DOS Commands Reference Manual.*)

Symbols:

Name	Type	Value	Attr	
FOO	Number	0005		all formed by EQU or = directive
FOO1	Text	1.234		
FOO2	Number	0008		
FOO3	Alias	FOO		
FOO4	Text	5[BP][DI]		
FOO5	Opcode			

Symbols:

```
        Name            Type        Value    Attr

BEGHQQ.................L WORD      0012     DATA        Global
BEGOQQ.................L FAR       0000           External
BEGXQQ.................F PROC      0000     MAIN_STARTUP Global Length=006E
CESXQQ.................L WORD      0022     DATA        Global     ┌─────
CLNEQQ.................L WORD      0002     DATA        Global     └──length
CRCXQQ.................L WORD      001C     DATA        Global          of PROC
CRDXQQ.................L WORD      001E     DATA        Global
CSXEQQ.................L WORD      0000     DATA        Global
CURHQQ.................L WORD      0014     DATA        Global
DOSOFF.................L WORD      0020     DATA
DOSXQQ.................F PROC      001E     ENTXCM      Global Length  =0019
ENDHQQ.................L WORD      0016     DATA        Global
ENDOQQ.................L FAR       0000                 External
ENDUQQ.................L FAR       0000                 External
ENDXQQ.................L FAR       0005     ENTXCM      Global
ENDYQQ.................L FAR       0000                 External
ENTGQQ.................L FAR       0000                 External
FREXQQ.................F PROC      006E     MAIN_STARTUP Global Length=0010
HDRFQQ.................L WORD      0006     DATA        Global
HDRVQQ.................L WORD      0008     DATA        Global
HEAPBEG...............BYTE         0000     STACK<────┬──EQU statements
HEAPLOW...............BYTE         0000     HEAP<─────┘  showing segment
INIUQQ.................L FAR       0000                 External │
PNUXQQ.................L WORD      0004     DATA        Global   │
RECEQQ.................L WORD      0010     DATA        Global   │
REFEQQ.................L WORD      000C     DATA        Global   │
REPEQQ.................L WORD      000E     DATA        Global   │
RESEQQ.................L WORD      000A     DATA        Global   │
SKTOP..................BYTE        0014     STACK<───────────────┘
SMLSTK.................L NEAR      001C     MAIN_STARTUP
STARTMAIN..............F PROC      0000     ENTXCM Length=001E
STKBQQ.................L WORD      0018     DATA        Global
STKHQQ.................L WORD      001A     DATA        Global
┌──────────────────────
└──If MS-Assembler knows this length as one of the type lengths (BYTE, WORD, DWORD, QWORD,
      TBYTE), it shows that type name here.
```

This section lists all other symbolic values in your program that do not fit under the other categories.

Type shows the symbol's type:

```
L = Label
F = Far
N = Near
PROC = Procedure
Number |
Alias  |———————— all defined by EQU or = directive
Text   |
Opcode |
```

These entries may be combined to form the various types shown in the example.

For all procedures, the length of the procedure is given after its attribute (segment).

You may also see an entry under Type like:

```
L 0031
```

This entry results from code such as the following:

```
BAZ LABEL FOO
```

where FOO is a STRUC that is 31 bytes long.

BAZ will be shown in the symbol table with the L 0031 entry. Basically, Number (and some other similar entries) indicates that the symbol was defined by an EQU or = directive.

Value usually shows the numeric value the symbol represents. When the symbol was defined by an EQU or = directive, the Value column shows some text.

Attr shows the segment of the symbol, if known. Otherwise, the Attr column is blank. Following the segment name, the table will show either External, Global, or a blank (which means not declared with either the EXTRN or PUBLIC directive). The last entry applies to PROC types only. This is a length = entry, which is the length of the procedure.

If Type is *Number, Opcode, Alias,* or *Text,* the Symbols section of the listing will be structured differently. Whenever you see one of these four entries under Type, the symbol was created by an EQU directive or an = directive. All information that follows one of these entries is considered its "value," even if the "value" is simple text.

Each of the four types shows a value as follows:

Number shows a constant numeric value.

Opcode shows a blank. The symbol is an alias for an instruction mnemonic.

Sample directive statement: FOO EQU ADD

Alias shows a symbol name which the named symbol equals.

Sample directive statement: FOO EQU BAX

Text shows the "text" the symbol represents. "Text" is any other operand to an EQU directive that does not fit one of the other three categories above.

Sample directive statements:

```
GOO EQU 'WOW'
BAZ EQU DS:8[BX]
ZOO EQU 1.234
```

Appendices

Appendix A / Reserved Words

DUAL FUNCTION KEYWORD/SYMBOLS

AND	NOT	OR	SHL	SHR	XOR

SYMBOLS

AAA	ENTER	FLDENV	FXCH	JNP	PUSH
AAD	ES	FLDL2E	FXTRACT	JNS	PUSHA
AAM	ESC	FLDL2T	FYL2X	JNZ	PUSHF
AAS	F2XM1	FLDLG2	FYL2XP1	JO	RCL
ADC	FABS	FLDLN2	HLT	JP	RCR
ADD	FADD	FLDPI	IDIV	JPE	REP
AH	FADDP	FLDZ	IMUL	JPO	REPE
AL	FBLD	FMUL	IN	JS	REPNE
AX	FBSTP	FMULP	INC	JZ	REPNZ
BH	FCHS	FNCLEX	INS	LAHF	REPZ
BL	FCLEX	FNDISI	INSB	LDS	RET
BOUND	FCOM	FNENI	INSW	LEA	ROL
BP	FCOMP	FNINIT	INT	LEAVE	ROR
BX	FCOMPP	FNOP	INTO	LES	SAHF
CALL	FDECSTP	FNSAVE	IRET	LOCK	SAL
CBW	FDISI	FNSTCW	JA	LODS	SAR
CH	FDIV	FNSTENV	JAE	LODSB	SBB
CL	FDIVP	FNSTSW	JB	LODSW	SCAS
CLC	FDIVR	FPATAN	JBE	LOOP	SCASB
CLD	FDIVRP	FPREM	JC	LOOPE	SCASW
CLI	FENI	FPTAN	JCXZ	LOOPNE	SI
CMC	FFREE	FRNDINT	JE	LOOPNZ	SP
CMP	FIADD	FRSTOR	JG	LOOPZ	SS
CMPS	FICOM	FSAVE	JGE	MOV	ST
CMPSB	FICOMP	FSCALE	JL	MOVS	STC
CMPSW	FIDIV	FSQRT	JLE	MOVSB	STD
CS	FIDIVR	FST	JMP	MOVSW	STI
CWD	FILD	FSTCW	JNA	MUL	STOS
CX	FIMUL	FSTENV	JNAE	NEG	STOSB
DAA	FINCSTP	FSTP	JNB	NIL	STOSW
DAS	FINIT	FSTSW	JNBE	NOP	SUB
DEC	FIST	FSUB	JNC	OUT	TEST
DH	FISTP	FSUBP	JNE	OUTS	WAIT
DI	FISUB	FSUBR	JNG	OUTSB	XCHG
DIV	FISUBR	FSUBRP	JNGE	OUTSW	XLAT
DL	FLD	FTST	JNL	POP	XLATB
DS	FLD1	FWAIT	JNLE	POPA	??SEG
DX	FLDCW	FXAM	JNO	POPF	

NON-CONFLICTING KEYWORDS

DA	INCLUDE	NOERRORPRINT	NOPR	PAGEWIDTH	SB
DATE	LI	NOGE	NOPRINT	PAGING	STACK
DEBUG	LIST	NOGEN	NOSB	PI	SYMBOLS
EJ	M1	NOLI	NOSYMBOLS	PL	TITLE
EJECT	MACRO	NOLIST	NOTY	PR	TT
EP	MEMORY	NOMACRO	NOTYPE	PRINT	TY
ERRORPRINT	MOD186	NOMR	NOXR	PW	TYPE
GEN	MR	NOOBJECT	NOXREF	RESTORE	WF
GENONLY	NODB	NOOJ	OBJECT	RS	WORKFILES
GO	NODEBUG	NOPAGING	OJ	SA	XR
IC	NOEP	NOPI	PAGELENGTH	SAVE	XREF

HANDS-OFF KEYWORDS

ABS	DWORD	GT	NE	PTR	SEG
ASSUME	END	HIGH	NEAR	PUBLIC	SEGFIX
AT	ENDM	INPAGE	NOSEGFIX	PURGE	SEGMENT
BYTE	ENDP	LABEL	NOTHING	QWORD	SHORT
CODEMACRO	ENDS	LE	OFFSET	RECORD	SIZE
COMMON	EQ	LENGTH	ONLY186	RELB	STRUC
DB	EQU	LOW	ORG	RELW	TBYTE
DD	EVEN	LT	PAGE	RFIX	THIS
DQ	EXTRN	MASK	PARA	RFIXM	TYPE
DT	FAR	MOD	PREFX	RNFIX	WIDTH
DUP	GE	MODRM	PROC	RNFIXM	WORD
DW	GROUP	NAME	PROCLEN	RWFIX	?

Appendix B / Flag Operations

FLAG REGISTERS

Flags are used to distinguish or denote certain results of data manipulation. The 8086 provides the four basic mathematical operations (+, -, *, /) in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard two's complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer instructions in Chapter 6).

Adjustment operations are provided to allow arithmetic to be performed directly on unpacked decimal digits or on packed decimal representations, and the auxiliary flag (AF) facilitates these adjustments.

Flags also aid in interpreting certain operations which could destroy one of their operands. For example, a compare is actually a subtract operation; a zero result indicates that the operands are equal. Since it is unacceptable for the compare to destroy either of the operands, the processor includes several work registers reserved for its own use in such operations. The programmer cannot access these registers. They are used for internal data transfers and for holding temporary values in destructive operations, whose results are reflected in the flags.

Your program can test the setting of five of these flags (carry, sign, zero, overflow, and parity) using one of the conditional jump instructions. This allows you to alter the flow of program execution based on the outcome of a previous operation. the auxiliary carry flag is reserved for the use of the ASCII and decimal adjust instructions, as will be explained later in this section.

It is important for you to know which flags are set by a particular instruction. Assume, for example, that your program is to test the parity of an input byte and then execute one instruction sequence if parity is even, a different instruction sequence if parity is odd. Coding a JPE (jump if parity is even) or JPO (jump if parity is odd) instruction immediately following the IN (input) instruction would produce false results, since the IN instruction does not affect the condition flags. The jump conditionally executed by your program would reflect the outcome of some previous operation unrelated to the IN instructions.

For the operation to work correctly, you must include some instruction that alters the parity flag after the IN instruction, but before the jump instruction. For example, you can add zero to the input byte in the accumulator. This sets the parity flag without altering.the data in the accumulator.

In other cases, you will want to set a flag though there may be a number of intervening instructions before you test it. In these cases, you must check the operation of the intervening instructions to be sure that they do not affect the desired flag.

The flags set by each instruction are detailed in the individual instructions in Chapter 6 of this manual.

Details of Flag Usage. Six flag registers are set or cleared by most arithmetic operations to reflect certain properties of the result of the operation. They follow these rules below, where "set" means set to 1 and "clear" means cler to 0. Further discussion of each of these flags follows the concise description.

CF	is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.
AF	is set if the operation resulted in a carry out of (from addition) or borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.
ZF	is set if the result of the operation is zero; otherwise ZF is cleared.
SF	is set if the high-order bit of the result is set; otherwise SF is cleared.
PF	is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).
OF	is set if the signed operation resulted in an overflow, i.e., the operation resulted in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa; otherwise OF is cleared.

Carry Flag. As its name implies, the carry flag is commonly used to indicate whether an addition causes a "carry" into the next higher order digit. (However, the increment and decrement instructions (INC, DEC) do not affect CF.) The carry flag is also used as a "borrow" flag in subtractions.

The logical AND, OR, and XOR instructions also affect CF. These instructions set or reset particular bits of their destination (register or memory). See the descriptions of the logic instruction in Chapter 6.

The rotate and shift instructions move the contents of the operand (registers or memory) one or more positions to the left or right. They treat the carry flag as though it were an extra bit of the operand. The original value in CF is only preserved by RCL and RCR. Otherwise it is simply replaced with the next bit rotated out of the source, i.e., the high-order bit if an RCL is used, the low-order bit if RCR.

Example:

Addition of two one-byte numbers can produce a carry out of the high-order bit:

Bit Number:	**7654**	**3210**
AEH -	1010	1110B
+ 74H -	0111	0100B
122H	0010	0010B - 22H ;carry flag - 1

An addition that causes a carry out of the high-order bit of the destination sets the flag to 1; an addition that does not cause a carry resets the flag to zero.

Sign Flag. The high-order bit of the result of operations on registers or memory can be interpreted as a sign. Instructions that affect the sign flag set the flag equal to this high-order bit. A zero indicates a positive value; a one indicates a negative value. This value is duplicated in the sign flag so that conditional jump instructions can test for positive and negative values. The high order bit for byte value is bit 7; for word values it is bit 15.

Zero Flag. Certain instructions set the zero flag to one. This indicates that the last operation to affect ZF resulted in all zeros in the destination (register or memory). If that result was other than zero, then ZF is reset to 0. A result that has a carry and a zero result sets both flags, as shown below:

```
  10100111
 +01011001
 ---------    -----------
  00000000    Carry Flag = 1
              Zero Flag = 1
              meaning yes, zero
```

Parity Flag. Parity is determined by counting the number of one bits set in the low order 8 bits of the destination of the last operation to affect PF. Instructions that affect the parity flag set the flag to one for even parity and reset the flag to zero to indicate odd parity.

Auxiliary Carry Flag. The auxiliary carry flag indicates a carry out of bit 3 of the accumulator. You cannot test this flag directly in your program; it is present to enable the Decimal Adjust instructions to perform their function.

The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and XOR instructions.

Appendix C / Instructions Hexadecimal Order

00	00000000	MOD REG R/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01	00000001	MOD REG R/M	ADD	EA,REG	WORD ADD (REG) TO EA
02	00000010	MOD REG R/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03	00000011	MOD REG R/M	ADD	REG,EA	WORD ADD (EA) TO REG
04	00000100		ADD	AL,DATA8	BYTE ADD DATA TO REG AL
05	00000101		ADD	AX,DATA16	WORD ADD DATA TO REG AX
06	00000110		PUSH	ES	PUSH (ES) ON STACK
07	00000111		POP	ES	POP STACK TO REG ES
08	00001000	MOD REG R/M	OR	EA,REG	BYTE OR (REG) TO EA
09	00001001	MOD REG R/M	OR	EA,REG	WORD OR (REG) TO EA
0A	00001010	MOD REG R/M	OR	REG,EA	BYTE OR (EA) TO REG
0B	00001011	MOD REG R/M	OR	REG,EA	WORD OR (EA) TO REG
0C	00001100		OR	AL,DATA8	BYTE OR DATA TO REG AL
0D	00001101		OR	AX,DATA16	WORD OR DATA TO REG AX
0E	00001110		PUSH	CS	PUSH (CS) ON STACK
0F	00001111		(not used)		
10	00010000	MOD REG R/M	ADC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11	00010001	MOD REG R/M	ADC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12	00010010	MOD REG R/M	ADC	REA,EA	BYTE ADD (EA) W/ CARRY TO REG
13	00010011	MOD REG R/M	ADC	REG,EA	WORD ADD (EA) W/ CARRY TO REG
14	00010100		ADC	AL,DATA8	BYTE ADD DATA W/CARRY TO REG AL
15	00010101		ADC	AX,DATA16	WORD ADD DATA W/ CARRY TO REG AX
16	00010110		PUSH	SS	PUSH (SS) ON STACK
17	00010111		POP	SS	POP STACK TO REG SS
18	00011000	MOD REG R/M	SBB	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19	00011001	MOD REG R/M	SBB	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A	00011010	MOD REG R/M	SBB	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B	00011011	MOD REG R/M	SBB	REG,EA	WORD SUB (EA) W/ BORROW FROM REG
1C	00011100		SBB	AL,DATA8	BYTE SUB DATA W/ BORROW FROM REG AL
1D	00011101		SBB	AX,DATA16	WORD SUB DATA W/ BORROW FROM REG AX
1E	00011110		PUSH	DS	PUSH (DS) ON STACK
1F	00011111		POP	DS	POP STACK TO REG DS
20	00100000	MOD REG R/M	AND	EA,REG	BYTE AND (REG) TO EA
21	00100001	MOD REG R/M	AND	EA,REG	WORD AND (REG) TO EA
22	00100010	MOD REG R/M	AND	REG,EA	BYTE AND (EA) TO REG
23	00100011	MOD REG R/M	AND	REG,EA	WORD AND (EA) TO REG
24	00100100		AND	AL,DATA8	BYTE AND DATA TO REG AL
25	00100101		AND	AX,DATA16	WORD AND DATA TO REG AX
26	00100110		ES:		SEGMENT OVERIDE W/ SEGMENT REG ES
27	00100111		DAA		DECIMAL ADJUST FOR ADD
28	00101000	MOD REG R/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29	00101001	MOD REG R/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A	00101010	MOD REG R/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B	00101011	MOD REG R/M	SUB	REG,EA	WORD SUBTRACT (EA) FROM REG
2C	00101100		SUB	AL,DATA8	BYTE SUBTRACT DATA FROM REG AL
2D	00101101		SUB	AX,DATA16	WORD SUBTRACT DATA FROM REG AX
2E	00101110		CS:		SEGMENT OVERIDE W/ SEGMENT REG CS
2F	00101111		DAS		DECIMAL ADJUST FOR SUBTRACT
30	00110000	MOD REG R/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31	00110001	MOD REG R/M	XOR	EA,REG	WORD XOR (REG) TO EA
32	00110010	MOD REG R/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33	00110011	MOD REG R/M	XOR	REG,EA	WORD XOR (EA) TO REG
34	00110100		XOR	AL,DATA8	BYTE XOR DATA TO REG AL
35	00110101		XOR	AX,DATA16	WORD XOR DATA TO REG AX
36	00110110		SS:		SEGMENT OVERIDE W/ SEGMENT REG SS
37	00110111		AAA		ASCII ADJUST FOR ADD
38	00111000	MOD REG R/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39	00111001	MOD REG R/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A	00111010	MOD REG R/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B	00111011	MOD REG R/M	CMP	REG,EA	WORD COMPARE (REG) WITH (EA)
3C	00111100		CMP	AL,DATA8	BYTE COMPARE DATA WITH (AL)
3D	00111101		CMP	AX,DATA16	WORD COMPARE DATA WITH (AX)
3E	00111110		DS:		SEGMENT OVERIDE W/ SEGMENT REG DS
3F	00111111		AAS		ASCII ADJUST FOR SUBTRACT
40	01000000		INC	AX	INCREMENT (AX)
41	01000001		INC	CX	INCREMENT (CX)

42	01000010			INC	DX	INCREMENT (DX)
43	01000011			INC	DX	INCREMENT (BX)
44	01000100			INC	SP	INCREMENT (SP)
45	01000101			INC	BP	INCREMENT (BP)
46	01000110			INC	SI	INCREMENT (SI)
47	01000111			INC	DI	INCREMENT (DI)
48	01001000			DEC	AX	DECREMENT (AX)
49	01001001			DEC	CX	DECREMENT (CX)
4A	01001010			DEC	DX	DECREMENT (DX)
4B	01001011			DEC	BX	DECREMENT (BX)
4C	01001100			DEC	SP	DECREMENT (SP)
4D	01001101			DEC	BP	DECREMENT (BP)
4E	01001110			DEC	SI	DECREMENT (SI)
4F	01001111			DEC	DI	DECREMENT (DI)
50	01010000			PUSH	AX	PUSH (AX) ON STACK
51	01010001			PUSH	CX	PUSH (CX) ON STACK
52	01010010			PUSH	DX	PUSH (DX) ON STACK
53	01010011			PUSH	BX	PUSH (BX) ON STACK
54	01010100			PUSH	SP	PUSH (SP) ON STACK
55	01010101			PUSH	BP	PUSH (BP) ON STACK
56	01010110			PUSH	SI	PUSH (SI) ON STACK
57	01010111			PUSH	DI	PUSH (DI) ON STACK
58	01011000			POP	AX	POP STACK TO REG AX
59	01011001			POP	CX	POP STACK TO REG CX
5A	01011010			POP	DX	POP STACK TO REG DX
5B	01011011			POP	BX	POP STACK TO REG BX
5C	01011100			POP	SP	POP STACK TO REG SP
5D	01011101			POP	BP	POP STACK TO REG BP
5E	01011110			POP	SI	POP STACK TO REG SI
5F	01011111			POP	DI	POP STACK TO REG DI
63	01100011			(not used)		
64	01100100			(not used)		
65	01100101			(not used)		
66	01100110			(not used)		
67	01100111			(not used)		
70	01110000			JO	DISP8	JUMP ON OVERFLOW
71	01110001			JNO	DISP8	JUMP ON NOT OVERFLOW
72	01110010			JC/JB/JNAE	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73	01110011			JNC/JNB/JAE	DISP8	JUMP ON NOT BELOW/ABOVE OR EQUAL
74	01110100			JE/JZ	DISP8	JUMP ON EQUAL/ZERO
75	01110101			JNE/JNZ	DISP8	JUMP ON NOT EQUAL/NOT ZERO
76	01110110			JBE/JNA	DISP8	JUMP ON BELOW OR EQUAL/NOT ABOVE
77	01110111			JNBE/JA	DISP8	JUMP ON NOT BELOW OR EQUAL/ABOVE
78	01111000			JS	DISP8	JUMP ON SIGN
79	01111001			JNS	DISP8	JUMP ON NOT SIGN
7A	01111010			JP/JPE	DISP8	JUMP ON PARITY/PARITY EVEN
7B	01111011			JNP/JPO	DISP8	JUMP ON NOT PARITY/PARITY ODD
7C	01111100			JL/JNGE	DISP8	JUMP ON LESS/NOT GREATER OR EQUAL
7D	01111101			JNL/JGE	DISP8	JUMP ON NOT LESS/GREATER OR EQUAL
7E	01111110			JLE/JNG	DISP8	JUMP ON LESS OR EQUAL/NOT GREATER
7F	01111111			JNLE/JG	DISP8	JUMP ON NOT LESS OR EQUAL/GREATER
80	10000000	MOD 000	R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
80	10000000	MOD 001	R/M	OR	EA,DATA8	BYTE OR DATA TO EA
80	10000000	MOD 010	R/M	ADC	EA,DATA8	BYTE ADD DATA W/CARRY TO EA
80	10000000	MOD 011	R/M	SBB	EA,DATA8	BYTE SUB DATA W/BORROW FROM EA
80	10000000	MOD 100	R/M	AND	EA,DATA8	BYTE AND DATA TO EA
80	10000000	MOD 101	R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
80	10000000	MOD 110	R/M	XOR	EA,DATA8	BYTE XOR DATA TO EA
80	10000000	MOD 111	R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
81	10000001	MOD 000	R/M	ADD	EA,DATA16	WORD ADD DATA TO EA
81	10000001	MOD 001	R/M	OR	EA,DATA16	WORD OR DATA TO EA
81	10000001	MOD 010	R/M	ADC	EA,DATA16	WORD ADD DATA W/CARRY TO EA

81	10000001	MOD	011	R/M	SBB	EA,DATA16	WORD SUB DATA W/ BORROW FROM EA
81	10000001	MOD	100	R/M	AND	EA,DATA16	WORD AND DATA TO EA
81	10000001	MOD	101	R/M	SUB	EA,DATA16	WORD SUBTRACT DATA FROM EA
81	10000001	MOD	110	R/M	XOR	EA,DATA16	WORD XOR DATA TO EA
81	10000001	MOD	111	R/M	CMP	EA,DATA16	WORD COMPARE DATA WITH (EA)
82	10000010	MOD	000	R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
82	10000010	MOD	001	R/M	(not used)		
82	10000010	MOD	010	R/M	ADC	EA,DATA8	BYTE ADD DATA W/ CARRY TO EA
82	10000010	MOD	011	R/M	SBB	EA,DATA8	BYTE SUB DATA W/ BORROW FROM EA
82	10000010	MOD	100	R/M	(not used)		
82	10000010	MOD	101	R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
82	10000010	MOD	110	R/M	(not used)		
82	10000010	MOD	111	R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
83	10000011	MOD	000	R/M	ADD	EA,DATA8	WORD ADD DATA TO EA
83	10000011	MOD	001	R/M	(not used)		
83	10000011	MOD	010	R/M	ADC	EA,DATA8	WORD ADD DATA W/ CARRY TO EA
83	10000011	MOD	011	R/M	SBB	EA,DATA8	WORD SUB DATA W/ BORROW FROM EA
83	10000011	MOD	100	R/M	(not used)		
83	10000011	MOD	101	R/M	SUB	EA,DATA8	WORD SUBTRACT DATA FROM EA
83	10000011	MOD	110	R/M	(not used)		
83	10000011	MOD	111	R/M	CMP	EA,DATA8	WORD COMPARE DATA WITH (EA)
84	10000100	MOD	REG	R/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85	10000101	MOD	REG	R/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86	10000110	MOD	REG	R/M	XCHG	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87	10000111	MOD	REG	R/M	XCHG	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88	10001000	MOD	REG	R/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89	10001001	MOD	REG	R/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A	10001010	MOD	REG	R/M	MOV	REG,EA	BYTE MOVE (EA) TO REG
8B	10001011	MOD	REG	R/M	MOV	REG,EA	WORD MOVE (EA) TO REG
8C	10001100	MOD	0SR	R/M	MOV	EA,SR	WORD MOVE (SEGMENT REG SR) TO EA
8C	10001100	MOD	1--	R/M	(not used)		
8D	10001101	MOD	REG	R/M	LEA	REG,EA	LOAD EFFECTIVE ADDRESS OF EA TO REG
8E	10001110	MOD	0SR	R/M	MOV	SR,EA	WORD MOVE (EA) TO SEGMENT REG SR
8E	10001110	MOD	--	R/M	(not used)		
8F	10001111	MOD	000	R/M	POP	EA	POP STACK TO EA
8F	10001111	MOD	001	R/M	(not used)		
8F	10001111	MOD	010	R/M	(not used)		
8F	10001111	MOD	011	R/M	(not used)		
8F	10001111	MOD	100	R/M	(not used)		
8F	10001111	MOD	101	R/M	(not used)		
8F	10001111	MOD	110	R/M	(not used)		
8F	10001111	MOD	111	R/M	(not used)		
90	10010000				XCHG	AX,AX	EXCHANGE (AX) WITH (AX), (NOP)
91	10010001				XCHG	AX,CX	EXCHANGE (AX) WITH (CX)
92	10010010				XCHG	AX,DX	EXCHANGE (AX) WITH (DX)
93	10010011				XCHG	AX,BX	EXCHANGE (AX) WITH (BX)
94	10010100				XCHG	AX,SP	EXCHANGE (AX) WITH (SP)
95	10010101				XCHG	AX,BP	EXCHANGE (AX) WITH (BP)
96	10010110				XCHG	AX,SI	EXCHANGE (AX) WITH (SI)
97	10010111				XCHG	AX,DI	EXCHANGE (AX) WITH (DI)
98	10011000				CBW		BYTE CONVERT (AL) TO WORD (AX)
99	10011001				CWD		WORD CONVERT (AX) TO DOUBLE WORD
9A	10011010				CALL	DISP16,SEG16	DIRECT INTER SEGMENT CALL
9B	10011011				WAIT		WAIT FOR TEST SIGNAL
9C	10011100				PUSHF		PUSH FLAGS ON STACK
9D	10011101				POPF		POP STACK TO FLAGS
9E	10011110				SAHF		STORE (AH) INTO FLAGS
9F	10011111				LAHF		LOAD REG AH WITH FLAGS
A0	10100000				MOV	AL,ADDR16	BYTE MOVE (ADDR) TO REG AL
A1	10100001				MOV	AX,ADDR16	WORD MOVE (ADDR) TO REG AX
A2	10100010				MOV	ADDR16,AL	BYTE MOVE (AL) TO ADDR
A3	10100011				MOV	ADDR16,AX	WORD MOVE (AX) TO ADDR
A4	10100100				MOVS	DST8,SRC8	BYTE MOVE, STRING OP
A5	10100101				MOVS	DST16,SRC16	WORD MOVE, STRING OP
A6	10100110				CMPS	SIPTR,DIPTR	COMPARE BYTE, STRING OP
A7	10100111				CMPS	SIPTR,DIPTR	COMPARE WORD, STRING OP
A8	10101000				TEST	AL,DATA8	BYTE TEST (AL) WITH DATA
A9	10101001				TEST	AX,DATA16	WORD TEST (AX) WITH DATA
AA	10101010				STOS	DST8	BYTE STORE, STRING OP
AB	10101011				STOS	DST16	WORD STORE, STRING OP
AC	10101100				LODS	SRC8	BYTE LOAD, STRING OP
AD	10101101				LODS	SRC16	WORD LOAD, STRING OP
AE	10101110				SCAS	DIPTR8	BYTE SCAN, STRING OP

AF 10101111				SCAS	DIPTR16	WORD SCAN, STRING OP
B0 10110000				MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001				MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010				MOV	DL,DATA8	BYTE MOVE DATA TO REG DL
B3 10110011				MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100				MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101				MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110				MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111				MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000				MOV	AX,DATA16	WORD MOVE DATA TO REG AX
B9 10111001				MOV	CX,DATA16	WORD MOVE DATA TO REG CX
BA 10111010				MOV	DX,DATA16	WORD MOVE DATA TO REG DX
BB 10111011				MOV	BX,DATA16	WORD MOVE DATA TO REG BX
BC 10111100				MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD 10111101				MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE 10111110				MOV	SI,DATA16	WORD MOVE DATA TO REG SI
BF 10111111				MOV	DI,DATA16	WORD MOVE DATA TO REG DI

C2 11000010				RET	DATA16	INTRA SEGMENT RETURN, ADD DATA TO REG S
C3 11000011				RET		INTRA SEGMENT RETURN
C4 11000100	MOD	REG	R/M	LES	REG,EA	WORD LOAD REG AND SEGMENT REG ES
C5 11000101	MOD	REG	R/M	LDS	REG,EA	WORD LOAD REG AND SEGMENT REG DS
C6 11000110	MOD	000	R/M	MOV	EA,DATA8	BYTE MOVE DATA TO EA
C6 11000110	MOD	001	R/M	(not used)		
C6 11000110	MOD	010	R/M	(not used)		
C6 11000110	MOD	011	R/M	(not used)		
C6 11000110	MOD	100	R/M	(not used)		
C6 11000110	MOD	101	R/M	(not used)		
C6 11000110	MOD	110	R/M	(not used)		
C6 11000110	MOD	111	R/M	(not used)		
C7 11000111	MOD	000	R/M	MOV	EA,DATA16	WORD MOVE DATA TO EA
C7 11000111	MOD	001	R/M	(not used)		
C7 11000111	MOD	010	R/M	(not used)		
C7 11000111	MOD	011	R/M	(not used)		
C7 11000111	MOD	100	R/M	(not used)		
C7 11000111	MOD	101	R/M	(not used)		
C7 11000111	MOD	110	R/M	(not used)		
C7 11000111	MOD	111	R/M	(not used)		

CA 11001010				RET	DATA16	INTER SEGMENT RETURN, ADD DATA TO REG SP
CB 11001011				RET		INTER SEGMENT RETURN
CC 11001100				INT	3	TYPE 3 INTERRUPT
CD 11001101				INT	TYPE	TYPED INTERRUPT
CE 11001110				INTO		INTERRUPT ON OVERFLOW
CF 11001111				IRET		RETURN FROM INTERRUPT
D0 11010000	MOD	000	R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD	001	R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD	010	R/M	RCL	EA,1	BYTE ROTATE EA LEFT THRU CARRY 1 BIT
D0 11010000	MOD	011	R/M	RCR	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD	100	R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD	101	R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD	110	R/M	(not used)		
D0 11010000	MOD	111	R/M	SAR	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD	000	R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT

D1	11010001	MOD	001	R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1	11010001	MOD	010	R/M	RCL	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1	11010001	MOD	011	R/M	RCR	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1	11010001	MOD	100	R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1	11010001	MOD	101	R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1	11010001	MOD	110	R/M	(not used)		
D1	11010001	MOD	111	R/M	SAR	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT
D2	11010010	MOD	000	R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2	11010010	MOD	001	R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2	11010010	MOD	010	R/M	RCL	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2	11010010	MOD	011	R/M	RCR	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2	11010010	MOD	100	R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2	11010010	MOD	101	R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2	11010010	MOD	110	R/M	(not used)		
D2	11010010	MOD	111	R/M	SAR	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS
D3	11010011	MOD	000	R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3	11010011	MOD	001	R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS
D3	11010011	MOD	010	R/M	RCL	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3	11010011	MOD	011	R/M	RCR	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3	11010011	MOD	100	R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3	11010011	MOD	101	R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3	11010011	MOD	110	R/M	(not used)		
D3	11010011	MOD	111	R/M	SAR	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4	11010100	00001010			AAM		ASCII ADJUST FOR MULTIPLY
D5	11010101	00001010			AAD		ASCII ADJUST FOR DIVIDE
D6	11010110				(not used)		
D7	11010111				XLAT	TABLE	TRANSLATE USING (BX)
D8	11011---	MOD	---	R/M	ESC	EA	ESCAPE TO EXTERNAL DEVICE
D8	11011000	MOD	000	R/M	FADD	Short-real	ADD 4-BYTE EA TO ST
D8	11011000	MOD	001	R/M	FMUL	Short-real	MULTIPLY ST BY 4-BYTE EA
D8	11011000	MOD	010	R/M	FCOM	Short-real	COMPARE 4-BYTE EA WITH ST
D8	11011000	MOD	011	R/M	FCOMP	Short-real	COMPARE 4-BYTE EA WITH ST AND POP
D8	11011000	MOD	100	R/M	FSUB	Short-real	SUBTRACT 4-BYTE EA FROM ST
D8	11011000	MOD	101	R/M	FSUBR	Short-real	SUBTRACT ST FROM 4-BYTE EA
D8	11011000	MOD	110	R/M	FDIV	Short-real	DIVIDE ST BY 4-BYTE EA
D8	11011000	MOD	111	R/M	FDIVR	Short-real	DIVIDE 4-BYTE EA BY ST
D8	11011000	1 1	000	(i)	FADD	ST, ST(i)	ADD ELEMENT TO ST
D8	11011000	1 1	001	(i)	FMUL	ST, ST(i)	MULTIPLY ST BY ELEMENT
D8	11011000	1 1	010	(i)	FCOM	ST(i)	COMPARE ST(i) WITH ST
D8	11011000	1 1	011	(i)	FCOMP	ST(i)	COMPARE ST(i) WITH ST AND POP
D8	11011000	1 1	100	(i)	FSUB	ST, ST(i)	SUBTRACT ELEMENT FROM ST
D8	11011000	1 1	101	(i)	FSUBR	ST, ST(i)	SUBTRACT ST FROM STACK ELEMENT
D8	11011000	1 1	110	(i)	FDIV	ST, ST(i)	DIVIDE ST BY ELEMENT
D8	11011000	1 1	111	(i)	FDIVR	ST, ST(i)	DIVIDE ST(i) BY ST
D9	11011001	MOD	000	R/M	FLD	Short-real	PUSH 4-BYTE EA TO ST
D9	11011001	MOD	001	R/M	(not used)		
D9	11011001	MOD	010	R/M	FST	Short-real	STORE 4-BYTE REAL TO EA
D9	11011001	MOD	011	R/M	FSTP	Short-real	STORE 4-BYTE REAL TO EA AND POP
D9	11011001	MOD	100	R/M	FLDENV	14 BYTES	LOAD 8087 ENVIRONMENT FROM EA
D9	11011001	MOD	101	R/M	FLDCW	2-BYTES	LOAD CONTROL WORD FROM EA
D9	11011001	MOD	110	R/M	FSTENV	14-BYTES	STORE 8087 ENVIRONMENT INTO EA
D9	11011001	MOD	111	R/M	FSTCW	2-BYTES	STORE CONTROL WORD INTO EA
D9	11011001	1 1	000	(i)	FLD	ST(i)	PUSH ST(i) ONTO ST
D9	11011001	1 1	001	(i)	FXCH	ST(i)	EXCHANGE ST AND ST(i)
D9	11011001	1 1	010	000	FNOP		STORE ST IN ST
D9	11011001	1 1	010	001	(not used)		
D9	11011001	1 1	010	01-	(not used)		
D9	11011001	1 1	010	1--	(not used)		
D9	11011001	1 1	011	(i)	*(1)		
D9	11011001	1 1	100	000	FCHS		CHANGE SIGN OF ST
D9	11011001	1 1	100	001	FABS		TAKE ABSOLUTE VALUE OF ST
D9	11011001	1 1	100	01-	(not used)		
D9	11011001	1 1	100	100	FTST		TEST ST AGAINST 0.0
D9	11011001	1 1	100	101	FXAM		EXAMINE ST AND REPORT CONDITION CODE
D9	11011001	1 1	100	11-	(not used)		
D9	11011001	1 1	101	000	FLD1		PUSH +1.0 TO ST
D9	11011001	1 1	101	001	FLDL2T		PUSH log$_2$10 TO ST
D9	11011001	1 1	101	010	FLDL2E		PUSH log$_2$e TO ST
D9	11011001	1 1	101	011	FLDPI		PUSH Pi TO ST
D9	11011001	1 1	101	100	FLDLG2		PUSH log$_{10}$2 TO ST
D9	11011001	1 1	101	101	FLDLN2		PUSH log$_e$2 TO ST
D9	11011001	1 1	101	110	FLDZ		PUSH ZERO TO ST

D9 11011001	1 1	101	111	(not used)		
D9 11011001	1 1	110	000	F2XM1		CALCULATE $2^X - 1$
D9 11011001	1 1	110	001	FYL2X		CALCULATE FUNCTION $Y \cdot \log_2 X$
D9 11011001	1 1	110	010	FPTAN		CALCULATE TAN OF Θ AS A RATIO
D9 11011001	1 1	110	011	FPATAN		CALCULATE ARCTAN OF Θ
D9 11011001	1 1	110	100	FXTRACT		EXTRACT EXPONENT AND SIGNIFICAND FROM ST VALUE
D9 11011001	1 1	110	101	(not used)		
D9 11011001	1 1	110	110	FDECSTP		DECREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1	110	111	FINCSTP		INCREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1	111	000	FPREM		MODULO DIVISION OF ST BY ST(1)
D9 11011001	1 1	110	001	FYL2XP1		CALCULATE VALUE OF $Y \cdot \log_2 (X+1)$
D9 11011001	1 1	111	010	FSQRT		CALCULATE SQUARE ROOT OF ST
D9 11011001	1 1	111	011	(not used)		
D9 11011001	1 1	111	100	FRNDINT		ROUND ST TO INTEGER
D9 11011001	1 1	111	101	FSCALE		ADD ST(1) TO EXPONENT OF ST
D9 11011001	1 1	111	11-	(not used)		
DA11011010	MOD	000	R/M	FIADD	Short-integer	ADD 4-BYTE INTEGER EA TO ST
DA11011010	MOD	001	R/M	FIMUL	Short-integer	MULTIPLY ST BY 4-BYTE INTEGER EA
DA11011010	MOD	010	R/M	FICOM	Short-integer	CONVERT 4-BYTE INTEGER EA, AND COMPARE WITH ST
DA11011010	MOD	011	R/M	FICOMP	Short-integer	CONVERT 4-BYTE INTEGER EA, COMPARE WITH ST, POP
DA11011010	MOD	100	R/M	FISUB	Short-integer	SUBTRACT 4-BYTE INTEGER EA FROM ST
DA11011010	MOD	101	R/M	FISUBR	Short-integer	SUBTRACT ST FROM 4-BYTE INTEGER EA
DA11011010	MOD	110	R/M	FIDIV	Short-integer	DIVIDE ST BY 4-BYTE INTEGER EA
DA11011010	MOD	111	R/M	FIDIVR	Short-integer	DIVIDE 4-BYTE INTEGER EA BY ST
DA11011010	1 1	--	---	(not used)		
DB11011011	MOD	000	R/M	FILD	Short-integer	PUSH 4-BYTE INTEGER EA ONTO ST
DB11011011	MOD	001	R/M	(not used)		
DB11011011	MOD	010	R/M	FIST	Short integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA
DB11011011	MOD	011	R/M	FISTP	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA, POP
DB11011011	MOD	100	R/M	(not used)		
DB11011011	MOD	101	R/M	FLD	Temp-real	PUSH 10-BYTE EA ONTO ST
DB11011011	MOD	110	R/M	Reserved		
DB11011011	MOD	111	R/M	FSTP	Temp-real	STORE ST INTO 10-BYTE EA, POP
DB11011011	1 1	0--	---	Reserved		
DB11011011	1 1	100	000	FENI		ENABLE INTERRUPT
DB11011011	1 1	100	001	FDISI		DISABLE INTERRUPTS
DB11011011	1 1	100	010	FCLEX		CLEAR EXCEPTIONS
DB11011011	1 1	100	011	FINIT		INITIALIZE PROCESSOR
DB11011011	1 1	100	1--	Reserved		
DB11011011	1 1	101	---	Reserved		
DB11011011	1 1	11-	---	Reserved		
DC11011100	MOD	000	R/M	FADD	Long-real	ADD 8-BYTE EA TO ST
DC11011100	MOD	001	R/M	FMUL	Long-real	MULTIPLY ST BY 8-BYTE EA
DC11011100	MOD	010	R/M	FCOM	Long-real	COMPARE ST WITH 8-BYTE EA
DC11011100	MOD	011	R/M	FCOMP	Long-real	COMPARE ST WITH 8-BYTE EA, POP STACK
DC11011100	MOD	100	R/M	FSUB	Long-real	SUBTRACT 8-BYTE EA FROM ST
DC11011100	MOD	101	R/M	FSUBR	Long-real	SUBTRACT ST FROM 8-BYTE EA
DC11011100	MOD	110	R/M	FDIV	Long-real	DIVIDE ST BY 8-BYTE EA
DC11011100	MOD	111	R/M	FDIVR	Long-real	DIVIDE 8-BYTE EA BY ST
DC11011100	1 1	000	(i)	FADD	ST(i), ST	ADD ST TO ELEMENT
DC11011100	1 1	001	(i)	FMUL	ST(i), ST	MULTIPLY ELEMENT BY ST
DC11011100	1 1	010	(i)	*(2)		
DC11011100	1 1	011	(i)	*(3)		
DC11011100	1 1	100	(i)	FSUBR	ST(i), ST	SUBTRACT ST FROM ELEMENT
DC11011100	1 1	101	(i)	FSUB	ST(i), ST	SUBTRACT ELEMENT FROM ST
DC11011100	1 1	110	(i)	FDIVR	ST(i), ST	DIVIDE ST(i) BY ST
DC11011100	1 1	111	(i)	FDIV	ST(i), ST	DIVIDE ST BY ST(i)
DD11011101	MOD	000	R/M	FLD	Long-real	PUSH 8-BYTE EA ONTO ST
DD11011101	MOD	001	R/M	Reserved		
DD11011101	MOD	010	R/M	FST	Long-real	STORE ST INTO 8-BYTE EA
DD11011101	MOD	011	R/M	FSTP	Long-real	STORE ST INTO 8-BYTE EA, POP
DD11011101	MOD	100	R/M	FRSTOR	94-BYTES	RESTORE 8087 STATE FROM EA
DD11011101	MOD	101	R/M	Reserved		
DD11011101	MOD	110	R/M	FSAVE	94-BYES	SAVE 8087 STATE TO EA
DD11011101	MOD	111	R/M	FSTSW	2-BYTES	STORE 8087 STATUS WORD TO 2-BYTE EA
DD11011101	1 1	000	(i)	FFREE	ST(i)	SET STACK TAG TO "EMPTY"
DD11011101	1 1	001	(i)	*(4)		
DD11011101	1 1	010	(i)	FST	ST(i)	STORE ST INTO ST(i)
DD11011101	1 1	011	(i)	FSTP	ST(i)	STORE ST INTO ST(i), POP
DD11011101	1 1	1--	---	Reserved		
DE 11011110	MOD	000	R/M	FIADD	Word-integer	ADD 2-BYTE INTEGER EA TO ST
DE 11011110	MOD	001	R/M	FIMUL	Word-integer	MULTIPLY ST BY 2-BYTE INTEGER EA

DE 11011110	MOD	010	R/M	FICOM	Word-integer	COMPARE 2-BYTE EA INTEGER WITH ST
DE 11011110	MOD	011	R/M	FICOMP	Word-integer	COMPARE 2-BYTE INTEGER EA WITH ST. POP
DE 11011110	MOD	100	R/M	FISUB	Word-integer	SUBTRACT 2-BYTE INTEGER EA FROM ST
DE 11011110	MOD	101	R/M	FISUBR	Word-integer	SUBTRACT ST FROM 2-BYTE INTEGER EA
DE 11011110	MOD	110	R/M	FIDIV	Word-integer	DIVIDE ST BY 2-BYTE INTEGER EA
DE 11011110	MOD	111	R/M	FIDIVR	Word-integer	DIVIDE 2-BYTE INTEGER EA BY ST
DE 11011110	1 1	000	(i)	FADDP	ST(i). ST	ADD ST TO ELEMENT. POP
DE 11011110	1 1	001	(i)	FMULP	ST(i). ST	MULTIPLY ST BY ELEMENT. POP
DE 11011110	1 1	010	---	*(5)		
DE 11011110	1 1	011	000	Reserved		
DE11011110	1 1		011	001	FCOMPP	COMPARE ST WITH ST(1), POP TWICE
DE 11011110	1 1	011	01-	Reserved		
DE 11011110	1 1	011	1--	Reserved		
DE 11011110	1 1	100	(i)	FSUBRP	ST(i). ST	SUBTRACT ST FROM ELEMENT. POP
DE 11011110	1 1	101	(i)	FSUBP	ST(i). ST	SUBTRACT ST(i) FROM ST. POP
DE 11011110	1 1	110	(i)	FDIVRP	ST(i). ST	DIVIDE STACK ELEMENT BY ST. POP
DE 11011110	1 1	111	(i)	FDIVP	ST(i). ST	DIVIDE ST BY STACK ELEMENT. POP
DF 11011111	MOD	000	R/M	FILD	Word-integer	CONVERT 2-BYTE EA AND PUSH ONTO STACK
DF 11011111	MOD	001	R/M	Reserved		
DF 11011111	MOD	010	R/M	FIST	Word-integer	ROUND ST AND STORE IN 2-BYTE INTEGER EA
DF 11011111	MOD	011	R/M	FISTP	Word-integer	ROUND ST. STORE IN 2-BYTE INTEGER EA. POP
DF 11011111	MOD	100	R/M	FBLD	Packed decimal	LOAD BCD TO ST
DF 11011111	MOD	101	R/M	FILD	Long-integer	CONVERT 8-BYTE INTEGER EA AND PUSH ONTO STACK
DF 11011111	MOD	110	R/M	FBSTP	Packed decimal	CONVERT ST. STORE IN 10-BYTE BCD EA. POP
DF 11011111	MOD	111	R/M	FISTP	Long-integer	ROUND ST. STORE IN 8-BYTE INTEGER EA. POP
DF 11011111	1 1	000	(i)	*(6)		
DF 11011111	1 1	001	(i)	*(7)		
DF 11011111	1 1	010	(i)	*(8)		
DF 11011111	1 1	011	(i)	*(9)		
DF 11011111	1 1	---	---	Reserved		
E0 11100000				LOOPNZ/LOOPNE	DISP8	LOOP (CX) TIMES WHILE NOT ZERO/NOT EQUAL
E1 11100001				LOOPZ/LOOPE	DISP8	LOOP (CX) TIMES WHILE ZERO/EQUAL
E2 11100010				LOOP	DISP8	LOOP (CX) TIMES
E3 11100011				JCXZ	DISP8	JUMP ON (CX)-0
E4 11100100				IN	AL.PORT	BYTE INPUT FROM PORT TO REG AL
E5 11100101				IN	AX.PORT	WORD INPUT FROM PORT TO REG AX
E6 11100110				OUT	PORT.AL	BYTE OUTPUT (AL) TO PORT
E7 11100111				OUT	PORT.AX	WORD OUTPUT (AX) TO PORT
E8 11101000				CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001				JMP	DISP16	DIRECT INTRA SEGMENT JUMP
EA 11101010				JMP	DISP16.SEG16	DIRECT INTER SEGMENT JUMP
EB 11101010				JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC 11101010				IN	AL.DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED 11101010				IN	AX.DX	WORD INPUT FROM PORT (DX) TO REG AX
EE 11101010				OUT	DX.AL	BYTE OUTPUT (AL) TO PORT (DX)
EF 11101010				OUT	DX.AX	WORD OUTPUT (AX) TO PORT (DX)
F0 11110000				LOCK		BUS LOCK PREFIX
F1 11110001				(not used)		
F2 11110010				REPNZ/REPNE		REPEAT WHILE (CX)≠0 AND (ZF)=0
F3 11110011				REPZ/REPE/REP		REPEAT WHILE (CX)≠0 AND (ZF)=1
F4 11110100				HLT		HALT
F5 11110101				CMC		COMPLEMENT CARRY FLAG
F6 11110110	MOD	000	R/M	TEST	EA.DATA8	BYTE TEST (EA) WITH DATA
F6 11110110	MOD	001	R/M	(not used)		
F6 11110110	MOD	010	R/M	NOT	EA	BYTE INVERT EA
F6 11110110	MOD	011	R/M	NEG	EA	BYTE NEGATE EA
F6 11110110	MOD	100	R/M	MUL	EA	BYTE MULTIPLY BY (EA). UNSIGNED
F6 11110110	MOD	101	R/M	IMUL	EA	BYTE MULTIPLY BY (EA). SIGNED
F6 11110110	MOD	110	R/M	DIV	EA	BYTE DIVIDE BY (EA). UNSIGNED
F6 11110110	MOD	111	R/M	IDIV	EA	BYTE DIVIDE BY (EA). SIGNED
F7 11110111	MOD	000	R/M	TEST	EA.DATA16	WORD TEST (EA) WITH DATA
F7 11110111	MOD	001	R/M	(not used)		
F7 11110111	MOD	010	R/M	NOT	EA	WORD INVERT EA
F7 11110111	MOD	011	R/M	NEG	EA	WORD NEGATE EA
F7 11110111	MOD	100	R/M	MUL	EA	WORD MULTIPLY BY (EA). UNSIGNED
F7 11110111	MOD	101	R/M	IMUL	EA	WORD MULTIPLY BY (EA). SIGNED
F7 11110111	MOD	110	R/M	DIV	EA	WORD DIVIDE BY (EA). UNSIGNED
F7 11110111	MOD	111	R/M	IDIV	EA	WORD DIVIDE BY (EA). SIGNED
F8 11111000				CLC		CLEAR CARRY FLAG
F9 11111001				STC		SET CARRY FLAG
FA 11111010				CLI		CLEAR INTERRUPT FLAG
FB 11111011				STI		SET INTERRUPT FLAG

FC 11111100				CLD		CLEAR DIRECTION FLAG
FD 11111101				STD		SET DIRECTION FLAG
FE 11111110	MOD	000	R/M	INC	EA	BYTE INCREMENT EA
FE 11111110	MOD	001	R/M	DEC	EA	BYTE DECREMENT EA
FE 11111110	MOD	010	R/M	(not used)		
FE 11111110	MOD	011	R/M	(not used)		
FE 11111110	MOD	100	R/M	(not used)		
FE 11111110	MOD	101	R/M	(not used)		
FE 11111110	MOD	110	R/M	(not used)		
FE 11111110	MOD	111	R/M	(not used)		
FF 11111111	MOD	000	R/M	INC	EA	WORD INCREMENT EA
FF 11111111	MOD	001	R/M	DEC	EA	WORD DECREMENT EA
FF 11111111	MOD	010	R/M	CALL	EA	INDIRECT INTRA SEGMENT CALL
FF 11111111	MOD	011	R/M	CALL	EA	INDIRECT INTER SEGMENT CALL
FF 11111111	MOD	100	R/M	JMP	EA	INDIRECT INTRA SEGMENT JUMP
FF 11111111	MOD	101	R/M	JMP	EA	INDIRECT INTER SEGMENT JUMP
FF 11111111	MOD	110	R/M	PUSH	EA	PUSH (EA) ON STACK
FF 11111111	MOD	111	R/M	(not used)		

REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE:

16-BIT (W=1)		8-BIT (W=0)		SEGMENT REG	
000	AX	000	AL	00	ES
001	CX	001	CL	01	CS
010	DX	010	DL	10	SS
011	BX	011	BL	11	DS
100	SP	100	AH		
101	BP	101	CH		
110	SI	110	DH		
111	DI	111	BH		

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)

00	000	(BX) + (SI)	DS
00	001	(BX) + (DI)	DS
00	010	(BP) + (SI)	SS
00	011	(BP) + (DI)	SS
00	100	(SI)	DS
00	101	(DI)	DS
00	110	DISP16 (DIRECT ADDRESS)	DS
00	111	(BX)	DS
01	000	(BX) + (SI) + DISP8	DS
01	001	(BX) + (DI) + DISP8	DS
01	010	(BP) + (SI) + DISP8	SS
01	011	(BP) + (DI) + DISP8	SS
01	100	(SI) + DISP8	DS
01	101	(DI) + DISP8	DS
01	110	(BP) + DISP8	SS
01	111	(BX) + DISP8	DS
10	000	(BX) + (SI) + DISP16	DS
10	001	(BX) + (DI) + DISP16	DS
10	010	(BP) + (SI) + DISP16	SS
10	011	(BP) + (DI) + DISP16	SS
10	100	(SI) + DISP16	DS
10	101	(DI) + DISP16	DS
10	110	(BP) + DISP16	SS
10	111	(BX) + DISP16	DS
11	000	REG AX / AL	
11	001	REG CX / CL	
11	010	REG DX / DL	
11	011	REG BX / BL	
11	100	REG SP / AH	
11	101	REG BP / CH	
11	110	REG SI / DH	
11	111	REG DI / BH	

FLAGS REGISTER CONTAINS:

X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

*The marked encodings are NOT generated by the language translators. If however, the 8087 encounters one of these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)

(2) FCOM ST(i)

(3) FCOMP ST(i)

(4) FXCH ST(i)

(5) FCOMP ST(i)

(6) FFREE ST(i) and pop stack

(7) FXCH ST(i)

(8) FSTP ST(i)

(9) FSTP ST(i)

86/88/ INSTRUCTION SET MATRIX

Hi \ Lo	0	1	2	3	4	5	6	7
0	ADD b f r/m	ADD w f r/m	ADD b t r/m	ADD w t r/m	ADD b ia	ADD w ia	PUSH ES	POP ES
1	ADC b f r/m	ADC w f r/m	ADC b t r/m	ADC w t r/m	ADC b i	ADC w i	PUSH SS	POP SS
2	AND b f r/m	AND w f r/m	AND b t r/m	AND w t r/m	AND b i	AND w i	SEG ES	DAA
3	XOR b f r/m	XOR w f r/m	XOR b t r/m	XOR w t r/m	XOR b i	XOR w i	SEG SS	AAA
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5	USH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6								
7	JO	JNO	JB/ JNAE	JNB/ JAE	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JA
8	Immed b r/m	Immed w r/m	Immed b r/m	Immed is r/m	TEST b r/m	TEST w r/m	XCHG b r/m	XCHG w r/m
9	NOP	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI
A	MOV m → AL	MOV m → AX	MOV AL → m	MOV AX → m	MOVS b	MOVS w	CMPS b	CMPS w
B	MOV i → AL	MOV i → CL	MOV i → DL	MOV i → BL	MOV i → AH	MOV i → CH	MOV i → DH	MOV i → BH
C			RET (i+SP)	RET	LES	LDS	MOV b i r/m	MOV w i r/m
D	Shift b	Shift w	Shift b v	Shift w v	AAM	AAD		XLAT
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	IN b	IN w	OUT b	OUT w
F	LOCK		REP	REP z	HLT	CMC	Grp 1 b r/m	Grp 1 w r/m

Hi \ Lo	8	9	A	B	C	D	E	F
0	OR b f r/m	OR w f r/m	OR b t r/m	OR w t r/m	OR b i	OR w i	PUSH CS	
1	SBB b f r/m	SBB w f r/m	SBB b t r/m	SBB w t r/m	SBB b i	SBB w i	PUSH DS	POP DS
2	SUB b f r/m	SUB w f r/m	SUB b t r/m	SUB w t r/m	SUB b i	SUB w i	SEG CS	DAS
3	CMP b f r/m	CMP w f r/m	CMP b t r/m	CMP w t r/m	CMP b i	CMP w i	SEG DS	AAS
4	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6								
7	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	MOV b f r/m	MOV w f r/m	MOV b t r/m	MOV w t r/m	MOV sr f r/m	LEA	MOV sr t r/m	POP r/m
9	CBW	CWD	CALL l d	WAIT	PUSHF	POPF	SAHF	LAHF
A	TEST b i	TEST w i	STOS b	STOS w	LODS b	LODS w	SCAS b	SCAS w
B	MOV i → AX	MOV i → CX	MOV i → DX	MOV i → BX	MOV i → SP	MOV i → BP	MOV i → SI	MOV i → DI
C			RET l,(i+SP)	RET l	INT Type 3	INT (Any)	INTO	IRET
D	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	CALL d	JMP d	JMP l d	JMP si d	IN v b	IN v w	OUT v b	OUT v w
F	CLC	STC	CLI	STI	CLD	STD	Grp 2 b r/m	Grp 2 w r/m

where

mod r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR		SAR
Grp 1	TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
2	INC	DEC	CALL id	CALL l id	JMP id	JMP l id	PUSH	—

b	=	byte operation
d	=	direct
f	=	from CPU reg
i	=	immediate
ia	=	immed. to accum
ib	=	immediate byte
id	=	indirect
is	=	immed. byte sign ext
iw	=	immediate word
l	=	long ie. intersegment
m	=	memory
r	=	register
r/m	=	EA is second byte
si	=	short intrasegment
sr	=	segment register
t	=	to CPU reg
v	=	variable
w	=	word operation
z	=	zero

Appendix D / Instructions in Alphabetical Order

MS-Assembler supports both the 8086 and 8087 mnemonics. The mnemonics are listed alphabetically with their full names. The 8086 instructions are also listed in groups based on the type of arguments the instruction takes.

D.1 8086 Instruction Mnemonics, Alphabetical

Mnemonic	Full Name
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC	Add with carry
ADD	Add
AND	AND
CALL	CALL
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare
CMPS	Compare byte or word (of string)
CMPSB	Compare byte string
CMPSW	Compare word string
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement
DIV	Divide
ESC	Escape
HLT	Halt
IDIV	Integer divide
IMUL	Integer multiply
IN	Input byte or word
INC	Increment
INT	Interrupt
INTO	Interrupt on overflow
IRET	Interrupt return
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below

Mnemonic	Full Name
JBE	Jump on below or equal
JC	Jump on carry
JCXZ	Jump on CX zero
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero

Mnemonic	Full Name
LOOPZ	LOOP while zero
MOV	Move
MOVS	Move byte or word (of string)
MOVBS	Move byte (string)
MOVSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
NOT	NOT
OR	OR
OUT	Output byte or word
POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SCAS	Scan byte or word (of string)
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Subtract
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

D.2 8087 Instruction Mnemonics, Alphabetical

Mnemonic	Full Name
F2XM1	Calculate 2X-1
FABS	Take absolute value of top of stack
FADD	Add real
FADDP	Add real and pop stack
FBLD	Load packed decimal onto top of stack
FBSTP	Store packed decimal and pop stack
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOMP	Compare real and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIVP	Divide real and Pop stack
FDIVR	Reversed real divide
FDIVRP	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FIADD	Add integer
FICOM	Integer compare
FICOMP	Integer compare and pop stack
FIDIV	Integer divide
FIDIVR	Reversed integer divide
FILD	Load integer onto top of stack
FIMUL	Integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST	Store integer
FISTP	Store integer and pop stack
FISUB	Integer subtract
FISUBR	Reversed integer subtract

Mnemonic	Full Name
FLD	Load real onto top of stack
FLD1	Load +1.0 onto top of stack
FLDCW	Load control word
FLDENV	Load 8087 environment
FLDL2E	Load log 2 e onto top of stack
FLDL2T	Load log 2 10 onto top of stack
FLDLG2	Load log 10 2 onto top of stack
FLDLN2	Load log e 2 onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
FMULP	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor, with no WAIT
FNOP	No operation
FNSAVE	Save 8087 state with no WAIT
FNSTCW	Store control word without WAIT
FNSTENV	Store 8087 environment with no WAIT
FNSTSW	Store 8087 status word with on WAIT
FPATAN	Partial arctangent function
FPREM	Partial remainder
FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR	Restore state
FSAVE	Save 8087 state after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FSTCW	Store control word with WAIT
FSTENV	Store 8087 environment after WAIT
FSTP	Store real and pop stack
FSTSW	Store 8087 status word after WAIT
FSUB	Subtract real
FSUBP	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBRP	Reversed real subtract and pop stack

FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack element and stack top
FXTRACT	Extract exponent and significand from number in top of stack
FYL2X	Calculate Y:log 2 X
FYL2PI	Calculate Y:log 2 (x + 1)

Appendix E / Instructions by Argument Type

E.1 8086 Instruction Mnemonics by Argument Type

In this section, the instructions are grouped according to the type of argument(s) they take. In each group the instructions are listed alphabetically in the first column. The formats of the instructions with the valid argument types are shown in the second column. If a format shows OP, that format is legal for all the instructions shown in that group. If a format is specific to one mnemonic, the mnemonic is shown in the format instead of OP.

The following abbreviations are used in these lists:

OP = opcode: instruction mnemonic

reg = byte register (AL,AH,BL,BH,CL,CH,DL,DH)
or word register (AX,BX,CX,DX,SI,DI,BP,SP)

r/m = register or memory address or indexed and/or based

accum = AX or AL register

immed = immediate

mem = memory operand

segreg = segment register (CS,DS,SS,ES)

General 2 operand instructions

Mnemonics	Argument Types
ADC	OP reg,r/m
ADD	OP r/m,reg
AND	OP accum,immed
CMP	OP r/m,immed
OR	
SBB	
SUB	
TEST	
XOR	

In addition, add to the arguments a sign extent for word immediate.

CALL and JUMP type instructions

Mnemonics	Argument Types
CALL	OP mem {NEAR}{FAR} direction
JMP	OP r/m (indirect data — DWORD, WORD)

Relative jumps

Argument Type

OP addr (+ 129 or − 126 of IP at start, or 127 at end of jump instruction)
Mnemonics

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

Loop instructions : same as Relative jumps

LOOP	LOOPE	LOOPZ	LOOPNE	LOOPNZ

Return instruction

Mnemonic		Argument Type
RET	[immed]	(optional, number of words to POP)

No operand instructions

Mnemonics

AAA	CLD	DAA	LODSB	PUSHF	STI
AAD	CLI	DAS	LODSW	SAHF	STOSB
AAM	CMC	HLT	MOVSB	SCASB	STOSW
AAS	CMPSB	INTO	MOVSW	SCASW	WAIT
CBW	CMPSW	IRET	NOP	STC	XLATB
CLC	CWD	LAHF	POPF	STD	

Load instructions

Mnemonics		Argument Type
LDS	OP r/m	(except that OP reg is illegal)
LEA		
LES		

Move instructions

Mnemonic	Argument Types
MOV	OP mem,accum
	OP accum,mem
	OP segreg,r/m
	(except CS is illegal)
	OP r/m,segreg
	OP r/m,reg
	OP reg,r/m
	OP reg,immed
	OP r/m,immed

Push and pop instructions

Mnemonics	Argument Types
PUSH	OP word-reg
POP	OP segreg
	(POP CS is illegal)
	OP r/m

Shift/rotate type instructions

Mnemonics	Argument Types
RCL	OP r/m,1
RCR	OP r/m,CL
ROL	
ROR	
SAL	
SHL	
SAR	
SHR	

Input/output instructions

Mnemonics	Argument Types
IN	IN accum,byte-immed
	(immed = port 0-255)
	IN accum,DX
OUT	OUT immed,accum
	OUT DX,accum

Increment/decrement instructions

Mnemonics	Argument Types
INC	OP word-reg
DEC	OP r/m

Arith. multiply/divide/negate/not

Mnemonics	Argument Type
DIV	OP r/m (implies AX OP
IDIV	r/m, except NEG)
MUL	
IMUL	
NEG	(NEG implies AX OP NOP)
NOT	

Interrupt instruction

Mnemonic	Argument Types
INT	INT 3 (value 3 is one-byte instruction)
	INT byte-immed

Exchange instruction

Mnemonic	Argument Types
XCHG	XCHG accum,reg
	XCHG reg,accum
	XCHG reg,r/m
	XCHG r/m,reg

Miscellaneous instructions

Mnemonics	Argument Types	
XLAT	XLAT byte-mem	(only checks argument, not in opcode)

ESC ESC 6-bit-number,r/m

String primitives

These instructions have bits to record only their operand(s), if they are byte or word, and if a segment override is involved.

Mnemonics	Argument Types
CMPS	CMPS byte-word,byte-word (CMPS right operand is ES)
LODS	LODS byte/word,byte/word (LODS one argument = no ES)
MOVS	MOVS byte/word,byte/word (MOVS left operand is ES)
SCAS	SCAS byte/word,byte/word (SCAS one argument = ES)
STOS	STOS byte/word,byte/word (STOS one argument = ES)

Repeat prefix to string instructions

Mnemonics

LOCK
REP
REPE
REPZ
REPNE
REPNZ

E.2 8087 Instruction Mnemonics by Argument Type

No operands

F2XM1	FABS	FCHS	FCLEX	FCOMPP	FDECSTP
FDISI	FENI	FINCSTP	FINIT	FLD1	FLD2E
FLD2T	FLDLG2	FLDLN2	FLDPI	FLDZ	FNCLEX
FNDISI	FNENI	FNINIT	FNOP	FPATAN	FPREM
FPTAN	FRNDINT	FSCALE	FSQRT	FTST	FXAM
FXTRACT	FYL2X	FYL2XP1	FWAIT		

2-Argument Floating Arithmatic

Mnemonics	Argument Types
FADD	Blank
FDIV	mem 4,8 bytes
FDIVR	ST,ST(i)
FMUL	ST(i),ST
FSUB	
FSUBR	

Stack only floating point arithmatic

Mnemonics	Argument Types
FADDP	ST(i)
FDIVP	ST
FDIVRP	
FMULP	
FSUBP	
FSUBRP	

Compare and store using stack

Mnemonics	Argument Types
FCOM	ST
FCOMP	ST(i)
FST	blank

Stack

Mnemonics	Argument Types
FFREE	ST(i)
FXCH	blank

Integer arithmatic

Mnemonics	Argument Types
FIADD	mem 2,4 bytes
FICOM	
FICOMP	
FIDIV	
FIDIVR	
FIMUL	
FIST	
FISUB	
FISUBR	

Floating point load/store memory

Mnemonics	Argument Types
FLD	mem 4,8, or 10 bytes
FSTP	

Integer load/store memory

Mnemonics	Argument Types
FILD	mem 2,4, or 8 bytes
FISTP	

Load/store control or status

Mnemonics	Argument Types
FLDCW	mem 2 bytes
FNSTCW	
FNSTSW	
FSTCW	
FSTSW	

Save/Restore 8087 environment

Mnemonics	Argument Types
FLDENV	mem 14 bytes
FNSTENV	
FSTENV	

94-byte memory (8087 Save/Restore entire state)

Mnemonics	Argument Types
FNSAVE	mem 94 bytes
FRSTOR	
FSAVE	

BCD load/store

Mnemonics	Argument Types
FBLD	mem 10 bytes
FBSTP	

Appendix F / Directives (Pseudo-Ops) by Type

F.1 Memory Directives

```
             ASSUME <seg-reg>:<seg-name>
                  [,<seg-reg>:<seg-name>. . . ]
             ASSUME NOTHING
             COMMENT <delim><text><delim>

      <name> DB <exp>
      <name> DD <exp>
      <name> DQ <exp>
      <name> DT <exp>
      <name> DW <exp>

             END [<exp>]
      <name> EQU <exp>
      <name>  = <exp>
             EXTRN <name>:<type>[,<name>:
                  <type>. . . ]
             PUBLIC <name>[,<name>. . . ]
      <name> LABEL <type>
             NAME <module-name>

      <name> PROC [NEAR]
      <name> PROC [FAR]
               |
 <proc-name> ENDP

             .RADIX <exp>
      <name> RECORD <field>:<width>[=<exp>]
                  [, . . . ]

      <name> GROUP <segment-name>[, . . . ]
      <name> SEGMENT [<align>][<combine>]
                  [<class>]
               |
  <seg-name> ENDS
             EVEN
             ORG <exp>

      <name> STRUC
               |
<struc-name> ENDS
```

F.2 Macro Directives

```
        ENDM
        EXITM
        IRP <dummy>,<parameters in angle brackets>
        IRPC <dummy>,string
        LOCAL <parameter>[,<parameter>. . . ]
<name>  MACRO <parameter>[,<parameter>. . . ]
        PURGE <macro-name>[, . . . ]
        REPT <exp>
```

Special Macro Operators
& (ampersand) - concatenation
<text> (angle brackets - single literal)
;; (double semicolons) - suppress comment
! (exclamation point) - next character literal
% (percent sign) - convert expression to number

F.3 Conditional Directives

```
ELSE
IF <exp>
IFB <arg>
IFDEF <symbol>
IFDIF <arg1>,<arg2>
IFE <exp>
IFIDN <arg1>,<arg2>
IFNB <arg>
IFNDEF <symbol>
IF1
IF2
```

F.4 Listing Directives

.CREF
.LALL
.LFCOND
.LIST
%OUT <text>
PAGE <exp>
.SALL
.SFCOND
SUBTTL <text>
.TFCOND
TITLE <text>
.XALL
.XCREF
.XLIST

F.5 Attribute Operators

Override operators

Pointer (PTR)
    <attribute>   PTR   <expression>
Segment Override (:) (colon)
    <segment-register>:<address-expression>
    <segment-name>:<address-expression>
    <group-name>:<address-expression>
SHORT
    SHORT <label>
THIS
    THIS <distance>
    THIS <type>

Value Returning Operators

```
SEG
    SEG <label>
    SEG <variable>
OFFSET
    OFFSET <label>
    OFFSET <variable>
TYPE
    TYPE <label>
    TYPE <variable>
.TYPE
    .TYPE <variable>
LENGTH
    LENGTH <variable>
SIZE
    SIZE <variable>
```

Record Specific operators

```
Shift-count - (Record fieldname)
    <record-fieldname>
MASK
    MASK <record-fieldname>
WIDTH
    WIDTH <record-fieldname>
    WIDTH <record>
```

F.6 Precedence Of Operators

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: parentheses ( )
   angle brackets < >
   square brackets [ ]
   structure variable operand: <variable>.<field>
2. segment override operator: colon (:)
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MOD, SHL, SHR

6. +, − (both unary and binary)
7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT, .TYPE

Appendix G / ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	033	21H	!
001	01H	SOH	034	22H	“
002	02H	STX	035	23H	#
003	03H	ETX	036	24H	$
004	04H	EOT	037	25H	%
005	05H	ENQ	038	26H	&
006	06H	ACK	039	27H	'
007	07H	BEL	040	28H	(
008	08H	BS	041	29H	)
009	09H	HT	042	2AH	*
010	0AH	LF	043	2BH	+
011	0BH	VT	044	2CH	,
012	0CH	FF	045	2DH	-
013	0DH	CR	046	2EH	.
014	0EH	SO	047	2FH	/
015	0FH	SI	048	30H	0
016	10H	DLE	049	31H	1
017	11H	DC1	050	32H	2
018	12H	DC2	051	33H	3
019	13H	DC3	052	34H	4
020	14H	DC4	053	35H	5
021	15H	NAK	054	36H	6
022	16H	SYN	055	37H	7
023	17H	ETB	056	38H	8
024	18H	CAN	057	39H	9
025	19H	EM	058	3AH	:
026	1AH	SUB	059	3BH	;
027	1BH	ESCAPE	060	3CH	<
028	1CH	FS	061	3DH	=
029	1DH	GS	062	3EH	>
030	1EH	RS	063	3FH	?
031	1FH	US	064	40H	@
032	20H	SPACE			

Dec = decimal, Hex = hexadecimal (H), CHR = character, LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Dec	Hex	CHR	Dec	Hex	CHR
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[	123	7BH	{
092	5CH	\	124	7CH	|
093	5DH	]	125	7DH	}
094	5EH	^	126	7EH	~
095	5FH	_	128	7FH	DEL
096	60H	‘			

Dec = decimal, Hex = hexadecimal (H), CHR = character, LF = Line Feed, FF = Form Feed, CR = Carriage Return, DEL = Rubout

Appendix H / MS-Assembler and MS-CREF Messages

H.1 MS-Assembler Operating Messages

Banner Message and Command Prompts:

```
MS-Assembler v2.0 Copyright (C) Microsoft, Inc.

Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing [NUL.LST]:
Cross reference [NUL.CRF]:
```

End of Assembly Message:

```
Warning          Fatal
Errors           Errors
n                n                  (n = number of errors)
```

(your disk operating system's prompt)

H.2 MS-Assembler Error Messages

If the assembler encounters errors, error messages are output, along with the numbers of warning and fatal errors. Control is then returned to your disk operating system. The message is output either to your computer screen or to the listing file if you command one be created.

Error messages are divided into three categories: assembler errors, I/O handler errors, and runtime errors. In each category, messages are listed in alphabetical order with a short explanation where necessary. At the end of this appendix, the error messages are listed in numerical order without explanations.

Assembler Errors

Already defined locally (Code 23)

Tried to define a symbol as EXTERNAL that had already been defined locally.

Already had ELSE clause (Code 7)

Attempted to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF. . . ENDIF).

Already have base register (Code 46)

Tried to double base register.

Already have index register (Code 47)

Tried to double index address.

Block nesting error (Code 0)

Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is close of an outer level of nesting with inner level(s) still open.

Byte register is illegal (Code 58)

Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

Can't override ES segment (Code 67)

Tried to override the ES segment in an instruction where this override is not legal. For example, store string.

Can't reach with segment reg (Code 68)

There is no ASSUME that makes the variable reachable.

Can't use EVEN on BYTE segment (Code 70)

Segment was declared to be byte segment and attempt to use EVEN was made.

Circular chain of EQU aliases (Code 83)

An alias EQU eventually points to itself.

Constant was expected (Code 42)

Expected a constant and received something else.

CS register illegal usage (Code 59)

Tried to use the CS register illegally. For example, XCHG CS,AX.

Directive illegal in STRUC (Code 78)

All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

Division by 0 or overflow (Code 29)

An expression is given that results in a divide by 0.

DUP is too large for linker (Code 74)

Nesting of DUP's was such that too large a record was created for the linker.

8087 opcode can't be emulated (Code 84)

Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

Extra characters on line (Code 1)

This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

Field cannot be overridden (Code 80)

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Forward needs override (Code 71)

This message is not currently used.

Forward reference is illegal (Code 17)

Attempted to forward reference something that must be defined in pass 1.

Illegal register value (Code 55)

The register value specified does not fit into the "reg" field (the reg field is greater than 7).

Illegal size for item (Code 57)

Size of referenced item is illegal. For example, shift of a double word is not permitted.

Illegal use of external (Code 32)

Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external is not permitted.

Illegal use of register (Code 49)

Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

Illegal value for DUP count (Code 72)

DUP counts must be a constant that is not 0 or negative.

Improper operand type (Code 52)

Use of an operand such that the opcode cannot be generated.

Improper use of segment reg (Code 61)

Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

Index displ. must be constant (Code 54)

Illegal use of index display.

Label can't have seg. override (Code 65)

Illegal use of segment override.

Left operand must have segment (Code 38)

Used something in right operand that required a segment in the left operand (for example, ":").

More values than defined with (Code 76)

Too many fields given in REC or STRUC allocation.

Must be associated with code (Code 45)

Use of data-related item where code item was expected.

Must be associated with data (Code 44)

Use of code-related item where data-related item was expected (for example, MOV AX,<code-label>).

Must be AX or AL (Code 60)

Specification of some register other than AX or AL where only these are acceptable (for example, the IN instruction).

Must be index or base register (Code 48)

Instruction requires a base or index register and some other register was specified in square brackets, [ ].

Must be declared in pass 1 (Code 13)

Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

Must be in segment block (Code 69)

Attempted to generate code when not in a segment.

Must be record field name (Code 33)

Expected a record field name but received something else.

Must be record or field name (Code 34)

Expected a record name or field name and received something else.

Must be register (Code 18)

Register unexpected as operand but you furnished a symbol — was not a register.

Must be segment or group (Code 20)

Expected segment or group and something else was specified.

Must be structure field name (Code 37)

Expected a structure field name but received something else.

Must be symbol type (Code 22)

Must be WORD, DW, QW, BYTE, or TB but received something else.

Must be var, label or constant (Code 36)

Expected a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)

Use of one of the prefix instructions without specifying any opcode after it.

Near JMP/CALL to different CS (Code 64)

Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)

Immediate mode specified for an opcode that cannot accept the immediate (for example, PUSH).

No or unreachable CS (Code 62)

Tried to jump to a label that is unreachable.

Normal type operand expected (Code 41)

Received STRUCT, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Not proper align/combine type (Code 25)

SEGMENT parameters are incorrect.

One operand must be const (Code 39)

This is an illegal use of the addition operator.

Only initialize list legal (Code 77)

Attempted to use STRUC name without angle brackets, < >.

Operand combination illegal (Code 63)

Specification of a two-operand instrucion where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)

Illegal use of the subtraction operator.

Operand must have segment (Code 43)

Illegal use of SEG directive.

Operand must have size (Code 35)

Expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)

Access of operand is impossible because it is not in the current IP segment.

Operand types must match (Code 31)

Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

Operand was expected (Code 27)

Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)

Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)

In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

Override with DUP is illegal (Code 79)

In a STRUC initialization statement, you tried to use DUP in an override.

Phase error between passes (Code 6)

The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes (see Section 7.4, "MS-Assembler Command Switches").

Redefinition of symbol (Code 4)

This error occurs on pass 2 and succeeding definitions of a symbol.

Reference to mult defined (Code 26)

The instruction references something that has been multi-defined.

Register already defined (Code 2)

This occurs only if the assembler has internal logic errors.

Register can't be forward ref (Code 82)

Relative jump out of range (Code 53)

Relative jumps must be within the range -128 +127 of the current instruction, and the specific jump is beyond this range.

Segment parameters are changed (Code 24)

List of arguments to SEGMENT were not identical to the first time this segment was used.

Shift count is negative (Code 30)

A shift expression is generated that results in a negative shift count.

Should have been group name (Code 12)

Expected a group name but something other than this was given.

Symbol already different kind (Code 15)

Attempted to define a symbol differently from a previous definition.

Symbol already external (Code 73)

Attempted to define a symbol as local that is already external.

Symbol has no segment (Code 21)

Tried to use a variable with SEG, and the variable has no known segment.

Symbol is multi-defined (Code 5)

This error occurs on a symbol that is later redefined.

Symbol is reserved word (Code 16)

Attempted to use an assembler reserved word illegally (for example, to declare MOV as a variable).

Symbol not defined (Code 9)

A symbol is used that has no definition.

Symbol type usage illegal (Code 14)

Illegal use of a PUBLIC symbol.

Syntax error (Code 10)

The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)

The type specified is of an unacceptable size.

Unknown symbol type (Code 3)

Symbol statement has something in the type field that is unrecognizable.

Usage of ? (indeterminate) bad (Code 75)

Improper use of the "?" (for example, ? + 5).

Value is out of range (Code 50)

Value is too large for expected use (for example, MOV AL,5000).

Wrong type of register (Code 19)

Directive or instruction expected one type of register, but another was specified (for example, INC CS).

H.3 I/O Handler Errors

These error messages are generated by the I/O handlers. These messages appear in a different format from the Assembler Errors:

```
MASM Error — error-message-text
 in: filename
```

The *filename* is the name of the file being handled when the error occurred.

The *error-message-text* is one of the following messages:

```
Data format (Code 114)
Device full (Code 108)
Device name (Code 102)
Device offline (Code 105)
File in use (Code 112)
File name (Code 107)
File not found (Code 110)
File not open (Code 113)
File system (Code 104)
Hard data (Code 101)
Line too long (Code 115)
Lost file (Code 106)
Operation (Code 103)
Protected file (Code 111)
Unknown device (Code 109)
```

H.4 Runtime Errors

These messages may be displayed as your assembled program is being executed.

Internal Error

Usually caused by an arithmetic check. If it occurs, notify Tandy Corporation.

Out of Memory

This message has no corresponding number. Either the source was too big or too many labels are in the symbol table.

H.5 Numerical Order List of Error Messages

Code	*Message*
0	Block nesting error
1	Extra characters on line
2	Register already defined
3	Unknown symbol type
4	Redefinition of symbol
5	Symbol is multi-defined
6	Phase error between passes
7	Already had ELSE clause
8	Not in conditional block
9	Symbol not defined
10	Syntax error
11	Type illegal in context
12	Should have been group name
13	Must be declared in pass 1
14	Symbol type usage illegal
15	Symbol already different kind
16	Symbol is reserved word
17	Forward reference is illegal
18	Must be register
19	Wrong type of register
20	Must be segment or group
21	Symbol has no segment
22	Must be symbol type
23	Already defined locally
24	Segment parameters are changed
25	Not proper align/combine type
26	Reference to mult defined
27	Operand was expected
28	Operator was expected
29	Division by 0 or overflow
30	Shift count is negative
31	Operand types must match
32	Illegal use of external
33	Must be record field name
34	Must be record or field name
35	Operand must have size
36	Must be var, label or constant
37	Must be structure field name

38 Left operand must have segment
39 One operand must be const
40 Operands must be same or 1 abs
41 Normal type operand expected
42 Constant was expected
43 Operand must have segment
44 Must be associated with data
45 Must be associated with code
46 Already have base register
47 Already have index register
48 Must be index or base register
49 Illegal use of register
50 Value is out of range
51 Operand not in IP segment
52 Improper operand type
53 Relative jump out of range
54 Index displ. must be constant
55 Illegal register value
56 No immediate mode
57 Illegal size for item
58 Byte register is illegal
59 CS register illegal usage
60 Must be AX or AL
61 Improper use of segment reg
62 No or unreachable CS
63 Operand combination illegal
64 Near JMP/CALL to different CS
65 Label can't have seg. override
66 Must have opcode after prefix
67 Can't override ES segment
68 Can't reach with segment reg
69 Must be in segment block
70 Can't use EVEN on BYTE segment
71 Forward needs override
72 Illegal value for DUP count
73 Symbol already external
74 DUP is too large for linker
75 Usage of ? (indeterminate) bad
76 More values than defined with
77 Only initialize list legal
78 Directive illegal in STRUC
79 Override with DUP is illegal
80 Field cannot be overridden

81 Override is of wrong type
82 Register can't be forward ref
83 Circular chain of EQU aliases
84 8087 opcode can't be emulated

101 Hard data
102 Device name
103 Operation
104 File system

105 Device offline
106 Lost file
107 File name
108 Device full
109 Unknown device
110 File not found
111 Protected file
112 File in use
113 File not open
114 Data format
115 Line too long

H.6 MS-CREF Error Messages

All errors cause MS-CREF to abort. Control is returned to the operating system.

All error messages are displayed in the following format:

Fatal I/O Error <error number> in File: <filename>

where: <filename> is the name of the file where the error occurs.

<error number> is one of the numbers in the following list of errors:

Number	Error
101	Hard data error Unrecoverable disk I/O error
102	Device name error Illegal device specification (for example, X:FOO.CRF)
103	Internal error Report to Tandy Corporation
104	Internal error Report to Tandy Corporation

1Ø5	Device offline Disk drive door open, no printer attached, or similar device is offline.
1Ø6	Internal error Report to Tandy Corporation
1Ø8	Disk full
11Ø	File not found
111	Disk is write protected
112	Internal error Report to Tandy Corporation
113	Internal error Report to Tandy Corporation
114	Internal error Report to Tandy Corporation
115	Internal error Report to Tandy Corporation

Glossary

Glossary

Allocation — the assigning of a resource (memory space, etc) for the performance of an operation.

Argument — a reference factor used to locate an item in a table. A variable whose value will determine the value of the specified function. In a statement line, the arguments of a function are listed in parentheses after the function name.

ASCII — the American Standard Code for Information Interchange. This group of standard 8-bit codes is used by most computers, data terminals, and other computer devices. The eighth bit is usually not used or is only used for parity coding. With the remaining 7 bits there are 128 possible characters (four groups of 32 each).

One group of 32 is reserved for upper-case letters and common punctuation marks. A second group is used for numbers, spacing, and other punctuation. A third group is assigned to lower-case letters and rare punctuation marks. The remaining group is used for machine and control command codes.

Attributes — a subdivision of an entity. For example, in a data base the entity might be a person's name. The attributes could be the person's address, phone number, or job description. For MS-Assembler the entity might be a label or variable and the attributes would be items such as segment, offset, and type.

Binary — the base 2 numbering system used by computers at the machine language level. In the binary system all data is represented by combinations of two digits (0 or 1).

Bit — a binary digit (0 or 1).

Byte — a term that describes a group of binary digits (bits) that are acted on as a group. Most often, bytes consist of 8 or 16 binary digits.

Call — temporarily diverting control of the computer from the main routine or program to a designated (or "called") subroutine.

Code — the rules governing the manner in which data or instructions must be represented for a given computer.

Comment — a part of a program line that describes the effect or function of the line. The comment portion of a line has no effect on the operation of the computer. Comments are usually preceded by a character, such as an apostrophe, that tells the computer to ignore the characters that follow. Or comments may be restricted to a particular section of the display. MS-Assembler uses the fourth column for comments.

Concatenate — To link or unite together in a series.

Constant — data that has a fixed value.

Conditional — an instruction in a program that uses the values of designated variables in determining the next instruction to be executed.

Cursor — a video display character that indicates the position at which data may be entered or corrected. The character is most often an underline or flashing block on the screen.

Data — a constant or variable value.

Debug — to locate and correct errors in a program.

Default — a value automatically inserted by the computer when none is specified by the user.

Delimiter — any character that limits or ends a string of characters or a statement. Commas, plus signs, and square brackets are some of the delimiters used by MS-Assembler.

Directive — an instruction that controls the translation process (from assembler language to machine language for MS-Assembler).

Diskette Drives (A:,B:) — identifies the drives. These are used in front of a filename in a command statement.

Edit — change the contents of source code using commands such as insert, delete, change, copy, etc.

Expression — a group of characters or mnemonics that follow a required syntax and cause a desired computation to take place.

Field — a group of characters that are treated as a unit.

Fieldname — a name assigned to a group of characters (a field) during programming. When assembled, the field will be assigned an absolute address.

File — a collection of related records that are treated as a unit. A file may contain data, programs, or both.

Filename — a character or group of characters used to identify a collection of related records (a file).

Filespec — the identification of a file containing a disk drive ID (optional), filename, and filename extension (optional).

Forward Reference — a reference to a variable in a program before the variable has been introduced.

Hexadecimal — a base 16 numbering system used in computers at the assembly language level. In addition to the normal decimal digits (0 - 9), the hexadecimal system uses A - F. This provides the necessary total of 16 digits.

Inpage — a section of memory storage of less than 256 bytes that is contained in a single page.

Instruction — a step in a program that tells the computer to perform an operation.

I/O — an abbreviation for input/output.

Label — a symbol, word, or abbreviation designated to identify a specific block of information. The name is usually closely related to the information it identifies. For example, OBJ for object code.

Linker — a part of the disk operating system (an MS-DOS Utility) that creates a load module from two or more independently translated object modules.

Loop — a series of instructions that is repeated a fixed number of times or until certain conditionals are met.

Machine Code — a binary representation of the source code, that is capable of being read and acted on by the computer.

Memory Address — the exact location in memory where a byte of data may be found. Also, the memory location where a related group of information begins. Memory addresses are most often referred to by their hexadecimal number.

Mnemonic — an abbreviation of or acronym for labels, variables, codes, etc., that is intended to be easy for the programmer to remember. Most often this term refers to the operation codes and directives.

Object Code — the machine code produced by the assembler that can be acted on by the computer or which can be processed to produce executable code.

Octal Code — a code with a radix of 8, in which the code is represented by the digits 0 through 7.

Offset — an operator used in indirect addressing. It defines a position in the file, in reference to another point. The reference point could be a label, a ,variable, the beginning or end or a module, etc.

Operand — the part on which an operation is performed (data - possibly the result of a previous operation). This data could be a variable, a memory location, an argument, an attribute, etc.

Operator —'the part of a statement that says what to do with data in the statement. For example, move, add, subtract, store, load, etc.

Page — a section of memory storage with a beginning address that is a multiple of 256.

Pop — retrieving information from a stack.

Push — placing information on a stack.

Radix — the base of a numbering system. For example, binary numbering has a radix of 2, octal 8, and hexadecimal 16.

Record — a group of consecutive related fields.

Register — a temporary memory storage location used to facilitate arithmetical, logical, or transfer operation. A register is usually 8 or 16 bits of memory.

Segment — a part of a routine. If a routine is too large to fit in internal memory, it may be divided into logical subroutines. Each subroutine will reference the address of the next segment. Some assemblers will automatically divide long routines into segments.

Source Code — the code that is input by the programmer for translation into object code and/or machine code.

Stack — a portion of memory or a register used to temporarily store data.

Truncation — deletion of the trailing portion of a string of items.

Variable — An item which assumes any of a given set of values.

Word — a character or group of characters that occupy one memory storage location. A word is usually treated as a single unit by the computer.

Index

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

12/83-SP

Printed in U.S.A.