IBM Macro Assembler/2™ Computer Language Series

Language Reference

Programming Family

00F8619

IBM Macro Assembler/2™ Computer Language Series

Language Reference

Programming Family

First Edition (1987)

About This Book

This book is part of the library covering the IBM *Macro Assembler/2*™.[1] The other two books in the library are the IBM *Macro Assembler/2 Fundamentals* book and the IBM *Macro Assembler/2 Assemble, Link, and Run* book.

This book contains detailed descriptions of pseudo operations (pseudo-ops) and instructions used by the IBM Macro Assembler/2.

This book is intended for experienced assembler language programmers. It is not expected, however, that you are experienced with the IBM Macro Assembler/2, or with any particular personal computer operating system environment.

Library Guide

If You Want To...	Refer to...
Install the product	Assemble, Link, and Run
Learn basic facts about the language	Fundamentals
Know the syntax of an instruction	Language Reference
Understand error messages	Language Reference
Debug a program	Assemble, Link, and Run
Assemble a program	Assemble, Link, and Run
Link a program	Assemble, Link, and Run

[1] ™Macro Assembler/2 is a trademark of the IBM corporation.

If You Want To...	Refer to...
Write a program	Fundamentals, Language Reference, and Assemble, Link, and Run

Related Publications

The following books will provide additional detailed information:

IBM Disk Operating System Reference

IBM Disk Operating System Technical Reference

IBM Operating System/2™ [2] User's Reference

IBM Operating System/2 Technical Reference

IBM Operating System/2 Programmer's Guide.

IBM Operating System/2 User's guide

---

[2] Operating System/2™ is a trademark of IBM Corporation.

Contents

Chapter 1. Introduction

This book provides you with instruction mnemonics and pseudo operations (pseudo-ops) that you need to use the IBM Macro Assembler/2. Before using this book, you need to be aware of the material in this chapter.

Notational Conventions

Certain conventions in this book define commands, formats of instructions and pseudo-operations, and terms:

- Items in square brackets **[ ]** are optional.

  Note: Do not confuse this convention with the square bracket notation used with registers, as described in "The Operand Field" section in the chapter, "Assembler Language Format" of the IBM *Macro Assembler/2 Fundamentals* book.

- Items separated by a vertical bar (|) mean that you can enter one of the separated items. For example:

  ON|OFF

  Means you can enter ON or OFF, but not both.

- An Ellipsis (...) shows that you can repeat an item as many times as you want.

- You must include all punctuation (commas, parentheses, angle brackets, slashes, or semicolons), except square brackets, where it is shown.

- Ctrl + Break and Ctrl + C perform the same function. Anytime Ctrl + Break is documented, you may also use Ctrl + C.

- The term *assembler* refers to the IBM Macro Assembler/2.

- Italics are used for:

  New terms when they are first defined in a book.
  Example: An *object module* is produced...

  Variables, including all-caps variables, in command formats and within text. You supply these items.

Example: TIME [*hh.mm.ss.xx*]

Book titles.
Example: IBM *Macro Assembler/2 Fundamentals*.

Boldface is used for:

Anything that you must type exactly as it appears in the book.
Example: Now, type **dir** and press...

Anything that appears on a screen that is referred to in text.
Example: The **Stack Overflow** message tells you...

Single alphabetic keys on the keyboard.
Example: Type **S** and...

- Small capital letters are used for:

  Sample file names in text.
  Example: Use the AUTOEXEC file...

  DOS programming commands.
  Example: The COPY command...

  Suffixes (file or language extensions) used alone.
  Example: A .BAT file is required...

  All acronyms and other fully capitalized words.
  Example: IBM

  Library names.
  Example: LIB1.LIB.

Hexadecimal Representation

This book represents hexadecimal numbers with the letter **H**, such as **59H**.

Operating Systems

Throughout these books, the references to operating systems have the following meaning:

Abbreviation	Meaning
DOS	IBM Disk Operating System Version 3.30
OS/2™[1]	IBM Operating System/2

Reference Material

The following books provide additional information about topics discussed in this book:

IBM Macro Assembler/2 Fundamentals, included with this product.

IBM Macro Assembler/2 Assemble, Link, and Run, included with this product.

The 8086 Book (includes the 8088), Rector, Russell and Alexy, George, Osborne/McGraw-Hill, Berkeley, California, 1980.

The iAPX 86,88 User's Manual 210201, *The 8086 Family User's Manual* 9800722, Literature Department, Intel®[2] Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051.

For 8087 users: *The 8086 Family User's Manual Numerics Supplement* 121586, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051.

For 80286 users: The *iAPX 286 Programmer's Reference Manual* (includes the iAPX 286 Numeric Supplement 210498), Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051.

---

1 OS/2™ is a trademark of the IBM Corporation.

2 Intel® is a registered trademark of the Intel Corporation.

Chapter 2. Getting Started

This chapter describes the IBM Macro Assembler/2 pseudo operations and instructions. It describes macro definitions and also discusses the 8088, 8087, 80286, and 80287 instruction sets, instruction fields, and instruction symbols.

Pseudo Operations

Pseudo operations, also called pseudo-ops, tell the assembler what to do with conditional branches, data, listings, and macros. Pseudo operations are sometimes called directives to the assembler. Pseudo-ops (except for certain data definition pseudo-ops) do not produce machine language code, although they have mnemonics similar to the mnemonics of the assembler instructions.

Each pseudo-op is described in more detail in Chapter 3, "Pseudo Operations," in this book. The pseudo-ops are arranged in alphabetical order in that chapter for ease of reference.

For the purpose of describing general information, this chapter groups the pseudo-ops by type. The seven types of pseudo-ops are:

- Conditional
- Conditional Error
- Data
- Listing
- Macro
- Mode
- Segment Order.

Type	Pseudo-ops		

Conditional	ELSE
	ENDIF
	IF
	IFB
		IFDEF
		IFDIF
		IFE
		IFIDN
			IFNB
			IFNDEF
			IF1
			IF2

Conditional Error	.ERR .ERRB .ERRDEF .ERRDIF	.ERRE .ERRIDN .ERRNB .ERRNDEF	.ERRNZ .ERR1 .ERR2
Data	ASSUME COMMENT DB DD DQ DT DW END ENDP	ENDS EQU = (Equal Sign) EVEN EXTRN GROUP INCLUDE LABEL	NAME ORG PROC PUBLIC .RADIX RECORD SEGMENT STRUC
Listing	.CREF .LALL .LFCOND .LIST %OUT	PAGE .SALL .SFCOND SUBTTL .TFCOND	TITLE .XALL .XCREF .XLIST
Macro	ENDM EXITM IRP	IRPC LOCAL MACRO	PURGE REPT
Mode	.186 .286C	.286P .287	.8086 .8087
Segment Order	.ALPHA	.SEQ	

Conditional Pseudo-Ops

The conditional pseudo-ops can be nested to any level. They are not limited to usage within a macro. Any operand of a conditional pseudo-op must be known on pass 1 to avoid errors and incorrect evaluation. Each IFXX pseudo-op must be matched with a corresponding ENDIF pseudo-op. All conditional pseudo-ops use the format:

```
IFxx operand
.
.
.
[ELSE]
.
.
.
ENDIF
```

Note: Do not confuse the assemble-time conditional pseudo-ops IFXX, ELSE, and ENDIF with the run-time conditional run structure statements $IF, $ELSE, and $ENDIF used by SALUT. See the discussion of SALUT in the IBM *Macro Assembler/2 Assemble, Link, and Run* book and in the IBM *Macro Assembler/2 Fundamentals* book for more detail on $IF, $ELSE, and $ENDIF.

Conditional Error Pseudo-Ops

The conditional error pseudo-ops and the errors they produce are listed in the table below. You can use these pseudo-ops to debug programs. By inserting a conditional error pseudo-op at a key point in your code, you can test assemble-time conditions at that point. You also can use these pseudo-ops to test for boundary conditions in macros.

All conditional error pseudo-ops, except .ERR1, produce unrecoverable errors. Like other unrecoverable assembler errors, those the conditional error pseudo-ops produce cause the assembler to return exit code 7. If the assembler finds an unrecoverable error during assembly, it erases the object module.

Pseudo-op	#	Message
.ERR1	87	**Forced error - pass1**
.ERR2	88	**Forced error - pass2**
.ERR	89	**Forced error**
.ERRE	90	**Forced error - expression equals 0**
.ERRNZ	91	**Forced error - expression not equal 0**
.ERRNDEF	92	**Forced error - symbol not defined**
.ERRDEF	93	**Forced error - symbol defined**
.ERRB	94	**Forced error - string blank**
.ERRNB	95	**Forced error - string not blank**
.ERRIDN	96	**Forced error - strings identical**
.ERRDIF	97	**Forced error - strings different**

Refer to the individual conditional pseudo-ops in Chapter 3, "Pseudo Operations," in this book.

Data Pseudo-Ops

Refer to the individual data pseudo-ops in Chapter 3, "Pseudo Operations," in this book.

Listing Pseudo-Ops

False Conditional Blocks

The listing of false conditional blocks is controlled by the pseudo-ops: .LFCOND (List False Conditionals), .SFCOND (Suppress False Conditionals), and .TFCOND (Toggle False Conditionals).

In the following example, one or the other of the %OUT statements is a false conditional block, depending on which pass the assembler is running.

```
IF2
 %OUT END OF IBM VERSION
ELSE
 %OUT START OF IBM VERSION
ENDIF
```

The assembler can either list or suppress the listing of false conditional blocks of an IF conditional pseudo-op. The decision to list or suppress is based on the following conditions.

At assemble start time, the default mode is to suppress the printing of false conditionals. You can set the default mode control to print false conditional blocks. To do this, use the /X option on the MASM command line.

When the assembler has started, the definition of the default condition can be toggled, that is, switched to its opposite condition, by including in the source file the pseudo-op, .TFCOND, Toggle False Condition. If the default condition is suppress, the .TFCOND changes the default to list. On the other hand, if the default condition is list, the .TFCOND sets the default to suppress.

Putting a .TFCOND at the start of the source has the same effect as entering the /X option on the MASM command line. However, having .TFCOND at the start of the listing and the /X on the command line produces the same effect as doing neither the pseudo-op nor the option.

Instead of allowing the default state to control the printing of false conditionals, you can expressly state, for a particular section of the code, that the default state is to be ignored and false conditionals are to be listed by including within the source the pseudo-op, .LFCOND, List False Conditionals. Similarly, false conditionals can be suppressed regardless of the default state by expressly using in the source the pseudo-op, .SFCOND, Suppress False Conditionals.

At the end of the particular section of code whose false conditionals are either to be listed or suppressed as commanded by the .LFCOND or .SFCOND pseudo-ops, you can tell the assembler to return to its default state regarding false conditionals by including within the source the pseudo-op, .TFCOND. This pseudo-op, however, not only indicates the

end of the section expressly controlled, but also, as previously stated, toggles the default state. If you want to end the section under express control and to return to the default state, issue the .TFCOND twice, and the second one toggles the default state definition back like it was.

Note: The listing of false conditionals defined within MACROS is suppressed unless the .LALL pseudo-op is used. For the /X option, .LFCOND, .SFCOND, and .TFCOND to be effective within MACROS, the .LALL pseudo-op must also be used.

This chart shows the effects of the three pseudo-ops when found with the /X and no /X options given on the MASM command line. If you used these pseudo-ops in the following order, this is what the result would be:

PSEUDO-OP	/X	NO /X
(default)	list	suppress
.SFCOND	suppress	suppress
.LFCOND	list	list
.TFCOND	suppress	list
.TFCOND	list	suppress
.SFCOND	suppress	suppress
.TFCOND	suppress	list
.TFCOND	list	suppress
.TFCOND	suppress	list

Note: The .SFCOND and .LFCOND pseudo-ops are absolute overrides. The .TFCOND pseudo-op toggles the default state. Vertical ellipses are intervening lines of code.

Macro and Repeat Block Pseudo-Ops

The macro facilities provided by the IBM Macro Assembler/2 include MACRO, REPT, IRP, and IRPC. The ENDM pseudo-op ends each of these four macro operations.

Terms Used

These terms are used with the MACRO, REPT, IRP, and IRPC pseudo-ops:

- A *dummy* represents a dummy parameter. All dummy parameters are valid symbols that appear in the body of a macro expansion.
- A *dummylist* is a list of *dummies* separated by commas.
- An <*operandlist*> is a list of operands separated by commas. The <*operandlist*> must be delimited by angle brackets. Two angle brackets with no intervening character entries (< >) or two commas with no intervening character entries become a null operand in the list. An operand is a character or series of characters ended by a comma or the end angle bracket (>). With angle brackets that are nested inside an <*operandlist*>, one level of brackets is removed each time the bracketed operand is used in an <*operandlist*>. A quoted string is an acceptable operand.
- A *parmlist* is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is ended by the end of line or a comment).
- A *macro* is a set of assembler statements that you can use several times in a program, with some optional changes each time.

Using a Macro

To properly use a macro, you must:

- Define a macro using the MACRO and ENDM pseudo-ops.
- Call the macro by using the name of the macro just as if it were an opcode.

You must define a macro before you call it. Once defined, a macro can be used, then redefined.

Defining a Macro: To define a macro, you must first understand the various parts of a MACRO definition:

- The beginning of a macro definition is the MACRO pseudo-op.
- The end of a macro definition is the ENDM pseudo-op.
- The body of a macro is all the lines of assembler statements between MACRO and ENDM.
- The *name* of the macro (the pseudo-op that is used to call it) is defined in the name field of the MACRO pseudo-op.
- The *parameters* (optional) are a series of one or more dummy symbol names listed as operands on the MACRO pseudo-op. If more than one parameter is used, the dummy symbols must be separated by commas or blanks. In some cases, there are no parameters. The number of parameters is limited to the number you can define on a 128-character MACRO pseudo-op statement.

This example shows the naming of the macro and the dummy parameter list. In this example, the string **VAL1** is put wherever the dummy **ARG1** appears in the macro, and **VAL2** for **ARG2**.

```
; This source file defines and uses the macro ADD2.
; The name of the macro is ADD2. ARG1 and ARG2 are dummies.
ADD2    MACRO ARG1,ARG2      ;; The beginning
        MOV AX,ARG1          ;; The body
        ADD AX,ARG2          ;; arg1 and arg2 will be substituted
        ENDM                 ;; The end
DSEG    SEGMENT
VAL1    DW 0
VAL2    DW 0
DSEG    ENDS
        ASSUME CS:CSEG,DS:DSEG
CSEG    SEGMENT

        ADD2 VAL1,VAL2       ;; Macro invocation statement
                             ;; MASM expands the macro here at
                             ;; assemble time with VAL1 &VAL2
                             ;; as the actual parameters
DSEG    ENDS
        END
```

Macro Parts: The body of the macro can have several parts:

- LOCAL — The local pseudo-op creates unique labels for use in macros. Normally, if a macro containing a label is used more than once, the assembler displays an error message indicating that the file contains a label or symbol with multiple definitions, because the same label appears in both expansions.

If this pseudo-op is used, it must be the next statement following the MACRO pseudo-op. No comment, not even a blank line, may appear between MACRO and LOCAL. The assembler substitutes a unique label for the label you used, so each time the macro is called, a different label is produced; this avoids duplication. Multiple, consecutive LOCAL statements are allowed.

Macro example for LOCAL pseudo-op:

```
LOOPZERO MACRO ARG1
LOCAL LOOPLABEL
        MOV AX,ARG1
LOOPLABEL:
        DEC AX
        JNZ LOOPLABEL
        ENDM
```

- Remarks or Comments — They are indicated either by the COMMENT pseudo-op or by a semicolon. They are listed in a macro expansion only if they are under the .LALL mode. A comment beginning with double semicolons is not listed when the macro is called.

- Pseudo-ops and Instructions — These statements become effective only when the MACRO containing them is called. For example, a MACRO definition can be contained within the body of another macro. The inner macro is not defined until the outer macro is called, as shown in the following example:

```
TITLE SAMPLE MACRO REDEFINING ITSELF
.LALL   ;set print mode
;           so (;) comments print
CALLSUB MACRO
            JMP   SHORT SKIP
;;Produce the subroutine,
;; one time only
SUBRT   PROC   NEAR

;   (body of subroutine here)
        RET    ;return from subrt
;               to main caller
SUBRT   ENDP
SKIP:   CALL SUBRT
;Redefine the 'callsub' macro so
; SUBRT is produced once only.
   CALLSUB MACRO
   CALL    SUBRT
           ENDM ;end of inner macro

        ENDM    ;end of outer macro
;**********************************
CSEG    SEGMENT
        ASSUME  CS:CSEG
MAIN      PROC  FAR
          CALLSUB  ;produce subrt,
;                    then call it
          CALLSUB  ;just call subrt
MAIN      ENDP
CSEG    ENDS
        END
```

Another useful operator is the IRP pseudo-op. It causes a block of the macro to be repeated. The block that gets repeated is from the IRP line to the next ENDM, which does not end the macro, just the repeating block. Using the double semicolon to suppress comments in the macro expansion is also illustrated in this example. Only comments preceded by a single semicolon show up when the Macro Assembler expands the macro. The IRPC is used in a similar fashion, but repeats once for each character in a string instead of once for each parameter in the parameter list.

```
:XAM  MACRO A1,A2,A3
      IRP X,<A1,A2,A3> ;; 'X' is the dummy
      MOV AX,X         ;This comment appears in listing
      PUSH AX          ;;This one doesn't
      ENDM             ;;End the repeating part
      CALL A1          ;;Non-repeating part
      ENDM             ;;End the macro

)SEG  SEGMENT
/AL1  DW 0
/AL2  DW 0
/AL3  DW 0
)SEG  ENDS
      ASSUME CS:CSEG,DS:DSEG
:SEG  SEGMENT
      EXAM VAL1,VAL2,VAL3 ;Call the macro
      MOV AX,VAL1      ;This comment gets listed
      PUSH AX
      MOV AX,VAL2      ;This comment gets listed
      PUSH AX
      MOV AX,VAL3      ;This comment gets listed
      PUSH AX
      CALL VAL1
:SEG  ENDS
      END
```

The section between the IRP and the first ENDM was repeated once for each symbol in the operand list, thereby pushing the values on the stack. The comment with the single semicolon was listed three times because it is in the repeating part, which had three parameters, and, therefore, repeated three times.

The example below illustrates the REPT pseudo-op, which repeats by a counter instead of for each parameter on a list like the IRP pseudo-op. The block between REPT line and the next ENDM will be repeated *n* times where *n* is a call argument at assembly time that has a numerical value associated with it. The **%** operator is also shown. It causes the numerical value of a symbol to be used in place of the symbol. In the first line %CNT is converted to the value of CNT; in the last line the **%** is left off and you can see that CNT was substituted in the expansion rather than the value of CNT. The **&** operator is used to substitute a dummy parameter in a string. If the **&** operator had been left out, the text would have read STRINGB instead of STRINGLENGTH.

```
SHOWOPS MACRO X,Y
CNT     = 0
        REPT X
        MKLINE %CNT,Y       ;;The value of cnt will be used
CNT     = CNT + 1
        ENDM                ;;end rept
        MKLINE CNT,Y        ; The symbol 'cnt' will be used
        ENDM                ;;End the macro
;THIS MACRO IS CALLED BY        'SHOWOPS'
MKLINE MACRO A,B
LINE&A  DB 'THE STRING&B',0
        ENDM

DSEG    SEGMENT
DSEG    ENDS
        ASSUME CS:CSEG,DS:DSEG
CSEG    SEGMENT

        SHOWOPS 3,LENGTH        ;invoke the macro 'SHOWOPS'
LINE0   DB 'THE STRINGLENGTH',0
LINE1   DB 'THE STRINGLENGTH',0
LINE2   DB 'THE STRINGLENGTH',0
LINECNT DB 'THE STRINGLENGTH',0
CSEG    ENDS
        END
```

Implementing Macro Libraries

The INCLUDE pseudo-op provides an excellent way of implementing macro libraries. You should create a source or include file that contains the macro definitions. Place the INCLUDE statement at the beginning of your source program. It specifies the include file, for instance, MACLIB.INC. Thus, MACLIB.INC is a separate text file that contains all macro definitions.

You can speed up assembly and print time if you run INCLUDE only on pass 1 by using the conditional pseudo-op IF1, as in:

```
IF1
  INCLUDE  maclib.inc
ENDIF
```

The macro definitions are not listed (because printing the listing is a pass 2 operation). Also, on pass 2, this file is not opened and read again, which is an unnecessary operation. This use of IF1 before the INCLUDE is not allowed if:

- The library contains source statements that actually produce code.
- MACRO redefined itself on the first call, as shown in the preceding example.

However, IF1 is useful if the library has macro definitions only. (The library can also have the definitions of structures and records but no expansions of these.)

The parameters are defined on the MACRO pseudo-op and are separated by commas. These parameters are position dependent.

```
FOO MACRO P1,P2,P3
```

P1 is the first parameter, P2 is the second, and P3 is the third.

These symbols, defined as parameters, can appear anywhere in the body of the macro.

An example of instructions using these parameters within a macro is:

```
MOV  AX,P1
P2   P3
```

When this macro is called (expanded), the macro call statement is whatever has been defined in the *name* field of a previously defined MACRO pseudo-op. The call statement can specify parameters that correspond by relative position to the dummy parameters defined in the macro definition. For example:

```
FOO  WORDVAR,INC,AX
```

In this example, WORDVAR is the first parameter, INC is the second, and AX is the third. The macro processor uses these specific character strings as direct replacements for the dummy parameters:

P1 is replaced by WORDVAR
P2 is replaced by INC
P3 is replaced by AX.

Now the instructions within the body of the macro are produced, showing the above substitutions, and become:

```
MOV  AX,WORDVAR
INC  AX
```

One parameter is used as a variable name, another as an opcode, and the last as a register name.

A dummy parameter within the body of a macro must be separated by commas, or it is not recognized as a dummy parameter and is not substituted. If the body has this statement:

```
JMP TAP1
TAP1:            ;TAP1 is a label
```

the symbol TAP1 is considered a label. The P1 portion is not replaced by the first parameter taken from the call statement. However, a dummy parameter can be recognized within a string by prefixing it with the ampersand (&) sign. For example:

```
JMP TA&P1
```

This time the operand is substituted, producing:

```
JMP TAWORDVAR
```

An instruction op-code can be built by joining together several pieces in this manner using the ampersand. For example:

```
LEAP MACRO   COND,LAB
     J&COND  LAB
     ENDM
```

When called by:

```
LEAP  Z,THERE
```

Producing:

```
JZ  THERE
```

The equal sign (=) pseudo-op is also helpful within a macro because it can be used to redefine the value of a counter. For example:

```
BUMP   MACRO
CNTR   =       CNTR+1
       ENDM
```

Note: Initialize CNTR before calling BUMP.

This counter has a value that you can get by using the special prefix, %, before the parameter in a macro call statement:

```
ERRMSG    MACRO  TEXT
CNTR      =      CNTR+1
          MSG    %CNTR,TEXT
          ENDM
;
MSG       MACRO  COUNT,STRING
MSG&COUNT DB STRING
          ENDM
```

Note: In the above example, one macro calls another.

Call the first macro by:

```
ERRMSG  'SYNTAX ERROR'
```

Which produces:

```
MSG1 DB 'SYNTAX ERROR'
```

Call it again with:

```
ERRMSG  'INVALID OPERAND'
```

Which produces:

```
MSG2 DB 'INVALID OPERAND'
```

The % prefix causes a different kind of substitution, a substitution of the value of a parameter, not the character string of the name of that parameter.

On closer examination of your results, you may wonder what happened to the CNTR = CNTR+1 statement, because the macro expansion does not show anything for this line. Apparently, the CNTR variable is being increased, as shown by the two variable names, MSG1 and MSG2. For a solution, try putting .LALL at the top of the source. The previous try was done in the .XALL mode (by default), which suppresses the listing of any statement that does not produce code.

Mode Pseudo-Ops

The mode facilities provided by the IBM Macro Assembler/2 include the /R and /E options and the .186, .286C, .286P, .287, .8086, and .8087 pseudo-ops. The Mode pseudo-ops, the /R option, and the /E option allow the IBM Macro Assembler/2 to be used with various machine setups and microprocessors.

Note: See Chapter 3, "Pseudo Operations," in this book for details on each of the mode pseudo-ops, and see the discussion of the /R and /E options in the IBN *Macro Assembler/2 Assemble, Link, and Run* book.

The IBM Macro Assembler/2 can assemble instructions for various machine setups and microprocessors.

For example:

- 8086/8088-based IBM Personal Computers and IBM Personal System/2 Computers
- 8086/8088-based IBM Personal Computers and IBM Personal System/2 Computers with the 8087 Math Coprocessor feature
- 80286/80386-based IBM Personal Computers and IBM Personal System/2 Computers
- 80286/80386-based IBM Personal Computers and IBM Personal System/2 Computers with the 80287 Math Coprocessor feature.

The instructions are considered upwardly compatible at the source-code level and generally upwardly compatible at the object-code level. This means that the instructions that are only for the 80286 are not compatible with the instructions for the 8088. However, the 8088 type instructions are compatible with the 80286 instructions. The IBM Macro Assembler/2 can recognize the different types of instructions. The mode pseudo-ops tell the Macro Assembler what type of instructions to accept.

The default mode is 8088/8086. In this mode, the 8087, 80287, and 80286 instructions cause a syntax error during assembly.

The functions of each mode pseudo-op are described below:

- A .186 mode pseudo-op allows the Macro Assembler to recognize 8086 instructions and instructions for the 80186 microprocessor.
- A .286C mode pseudo-op is required for the Macro Assembler to recognize 80286 nonprotected instructions in addition to 8088 or 8086 instructions.
- A .286P mode pseudo-op is required for the Macro Assembler to recognize 80286 protected instructions, in addition to 8088, 8086, and 80286 nonprotected instructions.
- A .287 mode pseudo-op is required for the assembly of 80287 floating-point coprocessor instructions. You can use the **/E** or **/R** options for the Macro Assembler to generate 80287-type instructions. If you specify **.287** and **/E**, all 80287 instructions will be generated with an FWAIT instruction.
- The .8087 mode pseudo-op lets you assemble 8087 instructions. You can use the **/E** or **/R** assembly options to produce 8087-type instructions, but these options are not required.
- The .8086 mode pseudo-op lets you assemble 8086 instructions and the identical 8088 instructions. This is the default mode.

Segment Order Pseudo-Ops

The segment order pseudo-ops tell the assembler how to order the segments from the source file when organizing the object file. These pseudo-ops cancel the /A and /S options.

Refer to the individual segment order pseudo-ops in Chapter 3, "Pseudo Operations," in this book.

Instructions

Each instruction is described in detail in Chapter 4, "Instruction Mnemonics," in this book. The instructions are arranged in alphabetical order for ease of reference.

8087 Instructions

Some instructions are available for 8088-based IBM Personal Computers with the 8087 Math Coprocessor. These 8087 instructions can be recognized by "(8087)," which appears next to the instruction headings in Chapter 4, "Instruction Mnemonics," in the table of contents, and in the index of this book.

For example:

- **FADD (8087) / Add Real**
- **FDIV (8087) / Divide Real**
- **FIST (8087) / Integer Store.**

Note: All 8087 instructions begin with the letter **F**, making them easier to distinguish from other instructions.

80286 Instructions

The 80286 has two modes of operation:

- Real address mode
- Protected virtual address mode.

See Chapter 6 of the IBM *Macro Assembler/2 Fundamentals* book for details about the 80286 architecture, and Chapter 3 of the IBM *Macro Assembler/2 Fundamentals* book for information on addressing memory in protected mode.

The 80286 instructions can be recognized by "(80286)," which appears next to the instruction headings in Chapter 4, "Instruction Mnemonics," in the table of contents, and in the index of this book. These instructions can be used only on an 80286/80386-based IBM Personal Computer or IBM Personal System/2.

For example:

- **PUSHA (80286)** Push All General Registers
- **POPA (80286)** Pop All General Registers
- **INS/INSB/INSW (80286)** Input from Port to String.

Some of the (80286) instructions are enhanced variations of 8088 instructions. For example, compare IMUL with IMUL (80286) as described in Chapter 4, "Instruction Mnemonics," in this book.

80287 Instructions

The 80287 instruction set consists of all 8087 instructions, plus three additional instructions: **FSETPM, FSTSW, and FNSTSW.**

Instruction Fields

In Chapter 4, "Instruction Mnemonics," the encoding section of each instruction shows what the assembled code looks like. There are 1 to 4 bytes represented, depending on the particular instruction or form of the instruction.

The general format of an assembled instruction is:

operation-code-byte [*addressing-mode-byte*[*displacements*]]

The *operation-code byte* determines if there is an *addressing-mode-byte*; and this, in turn, determines if *displacements* (disp(s)) are added to the end of the instruction.

The second instruction byte, if required, is the addressing mode byte (*modregr/m*), consisting of the *mode field* (mod), *register field* (reg), and *register/memory field* (r/m).

Mode Field:

If mod = 00, then disp = 0, disp-low and disp-high are absent (unless, mod = 00 and r/m = 110); then EA = disp-high and disp-low.

Note: See the section that follows, "Instruction Symbols and Definitions," for clarification of EA, disp-high, and disp-low.

If mod = 01, then disp = disp-low sign extended to 16 bits; disp-high is absent.

If mod = 10, then disp = disp-high and disp-low.

If mod = 11, then r/m is a reg field.

Register Field: Each of the general 8-bit, general 16-bit, base (BX and BP), and index (SI and DI) registers can be used in arithmetic and logical operations.

The following table shows the 3-bit binary code assignment for the general, base, and index registers and the 2-bit binary code assignment for the segment registers. w is a 1-bit field in an instruction, identifying byte instructions (w = 0) and word instructions (w = 1).

8-Bit Registers (w = 0)	16-Bit Registers (w = 1)	Segment Registers
AL = 000	AX = 000	ES = 00
CL = 001	CX = 001	CS = 01
DL = 010	DX = 010	SS = 10
BL = 011	BX = 011	DS = 11
AH = 100	SP = 100	
CH = 101	BP = 101	
DH = 110	SI = 110	
BH = 111	DI = 111	

Register/Memory Field

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 = 110, and mod = 00,
then EA = disp-high and disp-low
If r/m = 111, then EA = [BX] + disp

Flag Register

```
X NT IOPL OF DF IF TF SF ZF X AF X PF X CF
15               8  7                 0
```

X = Reserved

CF = Carry Flag

PF = Parity Flag

AF = Auxiliary Carry Flag

ZF = Zero Flag

SF = Sign Flag

TF = Trap Flag

IF = Interrupt Flag

DF = Direction Flag

OF = Overflow Flag

For the 80286 only:

NT = Nested Task Flag (1-bit)
IOPL = Input Output Privilege Level (2-bit)

These fields are reserved for the 8088.

Instruction Symbols and Definitions

The abbreviations and symbols that are used in assembler instructions are defined below. Abbreviations specific to protected mode instructions are listed separately after the more general symbols are defined.

Registers:

AX	Word Accumulator (16 bits), which can be addressed as two 8-bit registers (AH and AL).
AH	High-order byte of Word Accumulator (AX).
AL	Byte Accumulator (low-order byte of Word Accumulator AX).
BX	Data Base Register (16 bits), which can be addressed as two 8-bit registers (BH and BL).
BH	High-order byte of BX register.
BL	Low-order byte of BX register.
CX	Count Register (16 bits), which can be addressed as two 8-bit registers (CH and CL).
CH	High-order byte of CX register.
CL	Low-order byte of CX register.
DX	Data Register (16 bits), which can be addressed as two 8-bit registers (DH and DL).
DH	High-order byte of DX register.
DL	Low-order byte of DX register.
SP	Stack Pointer (16 bits).
BP	Base Pointer (16 bits).
DI	Destination index register (16 bits).
SI	Source Index register (16 bits).
CS	Code Segment register (16 bits).
DS	Data Segment register (16 bits).

ES Extra Segment register (16 bits).

SS Stack Segment register (16 bits).

IP Instruction Pointer (16 bits).

Flags 16-bit register word, where 9 flags reside.

Operands:

REG8 The name or encoding of an 8-bit CPU register location.

REG16 The name or encoding of a 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 left-most operand is also called the *destination* operand, and the right-most is called the *source* operand.

Addressing Mode Byte:

modregr/m The second byte of the instruction (usually identifies the operands of the instruction).

mod Bits 7 and 6 of the modregr/m byte. This 2-bit field defines the addressing mode.

reg Bits 5, 4, and 3 of the modregr/m byte. This 3-bit field usually specifies REG8 or REG16 in the description of an instruction, or three op-code bits of an 8087 instruction.

r/m Bits 2, 1, and 0 of the modregr/m byte used in getting memory operands. This 3-bit field defines EA with the mode and w fields. (EA and w are defined below.)

Other:

ST	The 8087 current top of the stack register.
ST(*i*)	An 8087 register stack element which is the *i*th register below the current top of stack register.
(*i*)	Bits 2, 1, and 0 of the modregr/m byte used in getting 8087 register stack element operands.
EA	Effective address (16 bits).
w	A 1-bit field in an instruction, identifying byte instructions (w = 0) and word instructions (w = 1).
s	If s = 0, 2-byte immediate. If s = 1, 1-byte immediate; sign bit extended to 16 bits.
d	A 1-bit field, d, identifies direction. For d = 0, register is the *source*. For d = 1, register is the *destination*.
(...)	Parentheses mean the contents of the enclosed register or memory location.
(BX)	The contents of the BX register, which is the address of an 8-bit operand.
((BX))	An 8-bit operand, which is the contents of the memory location pointed to by the contents of register BX.
(BX) + 1:(BX)	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))	The 16-bit operand that resides at address (BX) + 1:(BX).

Concatenation	For example, ((DX) + 1:(DX)). A word that is the concatenation of two 8-bit bytes, the low-order byte in the memory location pointed to by DX and the high-order byte in the next sequential memory location.
addr	Address (16 bits) of a byte in memory.
addr-low	Byte, least significant of an address.
addr-high	Byte, most significant of an address.
addr + 1:addr	Addresses of two consecutive bytes in memory, beginning at addr.
data	Immediate operand (8 bits if w = 0; 16 bits if w = 1).
data-low	Least significant byte of 16-bit data word.
data-high	Most significant byte of 16-bit data word.
disp	16-bit displacement.
disp-low	Least significant byte of 16-bit displacement.
disp-high	Most significant byte of 16-bit displacement.
<-	Assignment.
=	Assignment or equals.
+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.
%	Modulo.
&	AND.

IOR Inclusive OR.

XOR Exclusive OR.

The abbreviations and symbols used for protected mode instructions are:

CPL Current privilege level.

DPL Descriptor privilege level.

EA Effective address.

ecode Error code.

EPL Effective privilege level.

GDT Global descriptor table.

GDTR Global descriptor table register.

GP General protection.

IDT Interrupt descriptor table.

IEM Interrupt enable mask.

IOPL Input output privilege level.

LDT Local descriptor table.

LDTR Local descriptor table register.

MSW Machine status word.

NP Not present fault.

PL Privilege level.

RPL Requested privilege level.

SS Stack segment.

TOS	Top of stack.
TR	Task register.
TSS	Task state segment.
UD	Undefined fault.
VA	Virtual address.
VADW	Virtual address double word.

Chapter 3. Pseudo Operations

The pseudo-ops are arranged in this chapter in alphabetical order for ease of reference. Refer to Chapter 2, "Getting Started," in this book for general information on pseudo operations.

.186
Set 80186 Mode

Purpose

The .186 pseudo-op tells the Macro Assembler to recognize and assemble 8086 or 8088 instructions and the additional instructions for the 80186 microprocessor.

Format

.186

Remarks

The .186 does not have an operand. Use it only for programs that run on an 80186 microprocessor.

Note: See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

.286C
Set 80286 Mode

Purpose

The .286C pseudo-op tells the Macro Assembler to recognize and assemble nonprotected 80286 instructions. The 80286 nonprotected instruction set also includes all 8088 and 8086 instructions.

Format

.286C

Remarks

The .286C does not have an operand. You can end this mode by issuing the .8086 pseudo-op.

Note: See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

.286P
Set 80286 Protected Mode

Purpose

The .286P pseudo-op tells the Macro Assembler to recognize and assemble the protected instructions of the 80286 in addition to the 8086, 8088, and non-protected 80286 instructions.

Format

.286P

Remarks

The .286P pseudo-op does not have an operand. Use it only for programs run on an 80286 processor using both protected and non-protected instructions.

Note:

> See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

.287
Set 80287 Floating Point Mode

Purpose

The .287 pseudo-op tells the Macro Assembler to recognize and assemble instructions for the 80287 floating-point math coprocessor. The 80287 instruction set consists of all 8087 instructions, plus three additional instructions.

Format

.287

Remarks

The .287 pseudo-op does not have an operand. Use it only for programs that have 80287 floating-point instructions and run on an 80287 math coprocessor.

Note:

See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

.8086
Reset 80286 Mode

Purpose

The .8086 pseudo-op tells the Macro Assembler not to recognize and assemble 80286 instructions. This pseudo-op assembles only 8086 and 8088 instructions. (The 8088 instructions are identical to the 8086 instructions.) The Macro Assembler assembles 8086 instructions by default.

Format

.8086

Remarks

The .8086 does not have an operand. This pseudo-op ends the Macro Assembler 80286 assembler mode.

Notes:

1. The .8086 pseudo-op does not end the Macro Assembler 8087/80287 mode.
2. See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

.8087
Set 8087 Mode

Purpose

The .8087 pseudo-op tells the Macro Assembler to recognize and assemble 8087 instructions and data formats. The Macro Assembler assembles 8087 instructions by default.

Format

.8087

Remarks

The .8087 does not have an operand. Even though a source file may contain the .8087 pseudo-op, the Macro Assembler also requires the **/E** option on the MASM command line if the machine that the program will be run on does not have an 8087 math coprocessor. The .8087 pseudo-op has the same effect as issuing the /R parameter on the MASM command line.

Note: See "Mode Pseudo-Ops" in Chapter 2, "Getting Started," in this book for general information.

&
Special Macro Operator

Purpose

An ampersand (&) in a macro expansion forces the assembler to replace a dummy value with its corresponding actual parameter value.

Format

&*dummy* or *dummy*&

Remarks

The assembler does not substitute a dummy parameter that is in a quoted string or not preceded by a delimiter in the expansion unless it is immediately preceded by an ampersand (&). To form a symbol from text and a dummy, put an ampersand (&) between them.

Example

```
ERRGEN          MACRO   X
ERROR&X:        PUSH    BX
ABX             MOV     BX,"A"
AB&X            JMP     ERROR
                ENDM
```

The statement ERRGEN A produces this code:

```
ERRORA: PUSH    BX
ABX     MOV     BX,"A"
ABA     JMP     ERROR
```

;;
Special Macro Operator

Purpose

In a MACRO or repeat, a comment preceded by two semicolons (;;) is not produced as part of the expansion.

Format

;;*text*

Remarks

When two semicolons (;;) are used, the comment does not appear in the listing even when the .LALL pseudo-op is used. However, a comment preceded by one semicolon (;) is preserved and appears in the macro expansion.

Note: Under the default option of .XALL, a single semicolon remark is not printed because it does not produce any machine code. Whether or not regular comments are listed in macro expansions depends on the use of .LALL, .XALL, and .SALL.

The double semicolon can significantly reduce the amount of memory workspace used by the definition of a macro. As a macro definition is read, a single semicolon comment is kept in memory; the double semicolon comment is immediately discarded.

The double semicolon comment can be used on a line by itself or can be used on the comment fields of individual instructions or other statements.

< >
Literal-Text Operator

Format
<text>

Purpose

The literal-text operator directs the assembler to treat *text* as a single literal element regardless of whether it contains commas, spaces, or other separators. The operator is most often used with macro calls and the IRP pseudo-op to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force MASM to treat special characters such as the semicolon (;) or the ampersand (&) literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle-brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

!

Special Macro Operator

Purpose

When you use an exclamation point (!) in an operand, the Macro Assembler treats the next character literally. (!) is typically used to treat special characters such as the semicolon (;) or the ampersand (&) literally.

Format

!*character*

Remarks

For example, !; and <;> are equivalent entries.

%
Special Macro Operator

Purpose

The % special macro operator converts the expression that follows (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.

Format

%expression

Remarks

The percent sign (%) is used only in a macro operand. 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 operand substituting exactly for the dummy.

Example

```
MAKERR   MACRO   X
LB       =       0
         REPT    X
LB       =       LB+1
         MAKLIB  %LB
         ENDM    ;;END OF REPT
         ENDM    ;;END OF MACRO

MAKLIB   MACRO   Y
ERR&Y:   DB      'ERROR  &Y',0
         ENDM

         MAKERR 3
ERR1:    DB      'ERROR 1',0
ERR2:    DB      'ERROR 2',0
ERR3:    DB      'ERROR 3',0
```

=

Equal Sign

Purpose

The = pseudo-op lets you set and then redefine constant symbols. You can redefine the *symbol* more than once.

Format

symbol = *expression*

Remarks

The = pseudo-op is similar to the EQU pseudo-op except you can redefine the *symbol* without producing any message. However, unlike EQU, the = pseudo-op is limited only to numeric expressions (no text strings) that may also include assigning a type by using the PTR operator.

Example

```
EMP = 6      ;THIS IS THE SAME AS EMP EQU 6
EMP EQU 6    ;ERROR, EMP CANNOT BE REDEFINED
;             BY EQU
EMP = 7      ;EMP CAN BE REDEFINED
EMP = EMP+1 ;CAN REFER TO ITS PREVIOUS
;             DEFINITION
```

Note: Optionally, a TYPE attribute can be assigned to the symbol by using the PTR operator. For example:

```
VECTOR   =    DWORD PTR 4
```

Note: If that value were to be changed by some subsequent =, the typing symbol and PTR should be repeated, or the attribute of the symbol will be changed. For example:

```
VECTOR   =    DWORD PTR VECTOR+4
```

.ALPHA

Purpose

.ALPHA tells the assembler to arrange the segments in the object file in alphabetical order, by segment names.

Format

.ALPHA

Remarks

.ALPHA cancels the /S option used on the MASM command line.

Example

```
.ALPHA
DATA SEGMENT
     .
     .
     .
DATA ENDS
     .
     .
     .
CODE SEGMENT
     .
     .
     .
CODE ENDS
```

Segment CODE will precede segment DATA in the object module.

ASSUME

Purpose

ASSUME tells the assembler the segment register to which a segment belongs. When the assembler encounters a *seg-name*, it automatically assembles the *seg-name* reference under the proper segment register. ASSUME NOTHING cancels any previous ASSUME for the indicated register.

Format

ASSUME *seg-reg*: *seg-name*[,...]
or
ASSUME NOTHING

Remarks

The valid *seg-reg* entries are CS, DS, ES, and SS.

The possible entries for *seg-name* are:

- The name of a segment declared with the SEGMENT pseudo-op
- The name of a group declared with the GROUP pseudo-op
- An expression, either:

  ```
  SEG variable-name
  or
  SEG label-name
  ```

- The keyword NOTHING.

Note: If you do not use ASSUME or if you enter NOTHING for *seg-name*, you must prefix each *seg-name* type reference with a segment-register override.

If you temporarily use a *seg-reg* to contain a value other than the *seg-id* identified in the ASSUME statement, then you should use the ASSUME NOTHING to indicate that the *seg-reg* no longer has the old value. You can use this redefined *seg-reg* in an explicit reference. When the contents of the *seg-reg* are used for addressability, a *seg-reg* should never have a value that contradicts what the ASSUME says it has.

Example

```
    ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHING
    MOV    SI,OFFSET SOURCE
    MOV    DI,OFFSET DEST
    MOV    CX,LENGTH SOURCE
REP MOVS   BYTE PTR [DI],ES:[SI]  ;ES WILL BE USED AS THE
;                                  BASE REG FOR BOTH SOURCE AND DEST
```

Warning: If the ASSUME CS:seg-name is placed before the code segment it is referencing, the assembler will ignore the ASSUME. The ASSUME CS:seg-name statement must follow the SEGMENT definition statement of the code segment it is referencing.

The ASSUME statement for the CS register should be placed immediately following the code SEGMENT statement, before any labels are defined in that code segment. See the SKELCOM.ASM, SKELEXE.ASM, and SKELEXEP.ASM sample files included on your IBM Macro Assembler/2 diskettes for examples of ASSUME usage.

COMMENT

Purpose

COMMENT lets you enter comments about your program without having to enter semicolons (;) for each line.

Format

COMMENT *delimiter text delimiter*

Remarks

The first non-blank character after COMMENT is the first delimiter. The COMMENT pseudo-op causes the assembler to treat all *text* between *delimiter* and *delimiter* as a comment. The text must not contain the delimiter character. This pseudo-op is used for multiple-line comments. A COMMENT defined in the body of a macro does not appear unless .LALL is requested.

Note: Do not use the COMMENT pseudo-op in structured Assembler source code to be processed by the SALUT utility.

Example

```
COMMENT *You can enter as many lines
of text between the delimiters
.
.
.
as you need to describe your program.*
```

.CREF/.XCREF

Purpose

The output of the cross-reference information is controlled by these pseudo-ops. The default condition is the .CREF pseudo-op. When the assembler finds a .XCREF pseudo-op, cross-reference information results in no output until the assembler finds .CREF.

Format

.CREF
  or
.XCREF [*operand*]

Remarks

You must use the cross-reference utility to let these pseudo-ops function.

Note: See the discussion of the Cross-Reference Utility (CREF) in the IBM *Macro Assembler/2 Assemble, Link, and Run* book for more information.

The .XCREF pseudo-op can have an optional operand consisting of a list of one or more variable names suppressed in the cross-reference listing.

DB
Define Byte

Purpose

DB defines a variable or initializes memory. DB reserves one or more bytes (8 bits).

Format

[*variable-name*] DB *expression*[,...]

Remarks

If you enter a *variable-name*, the DB pseudo-op defines *variable-name* as a variable. Do not follow the *variable-name* with a colon (:). If the *variable-name* is followed by a colon (:), it is no longer a *variable-name* but is considered a code-relative label with the attribute of NEAR and is addressable only in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant expression. The number of constants is limited only by the length of the line. Use a comma to separate the elements. Each element must have a value of 255 or less.
- The question mark (?), for undefined initialization.
- A character string (ASCII characters occupying succeeding character positions).
- A duplicated clause, for example:

```
repeat count DUP(expression)...
```

Note: If you use an expression in the form of *repeat count* DUP(?) as the only operand of a DB, DW, DD, DT, or DQ definition statement, it produces an uninitialized data block. This data block is suitable for references to locations, which cannot or should not be overwritten by the loader. Any other form of ? initializes the data block with data of an unknown value.

Example

```
NUM_BASE    DB 16
FILLER      DB  ?
;   INITIALIZE WITH INDETERMINATE VALUE
ONE_CHAR    DB 'B'
MULT_CHAR   DB 'JENNIFER JEFFREY AMARYLLES TOM'
MSG         DB 'MSGTEST',13,10
;   MESSAGE,CARRIAGE RET, & LINEFEED
BUFFER      DB 10 DUP(?)
TABLE       DB 100 DUP(5 DUP(4),7)
;   100 COPIES OF BYTES=4,4,4,4,4,7
NEW_PAGE    DB 0CH
;   PRINTER CONTROL
ARRAY       DB 1,2,3,4,5,6,7
```

DD
Define Doubleword

Purpose

Use the DD pseudo-op to define a variable or to initialize memory. DD reserves two words (4 bytes).

Format

[*variable-name*] DD *expression*[,...]

Remarks

If you enter a *variable-name*, the DD pseudo-op defines *variable-name* as a variable. Do not follow *variable-name* with a colon (:). If *variable-name* is followed by a colon (:), it is considered a code-relative label with the attribute of NEAR and addressable only in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant, in which case:

  Integers are produced unless a decimal point or scientific notation is used. Then floating point is produced.

  Note: Refer to the assembler options /R and /E for their effect on floating-point constants.

  The number of constants is limited only by the length of the line. Use a comma to separate the elements.

- A question mark (?) for unknown initialization.
- An address expression, which produces a 16-bit offset, then the 16-bit segment base-value of the symbol. (Vector).
- A duplicated clause; for example:

```
repeat-count DUP(expression)...
```

Note: If you use an expression in the form of *repeat count* DUP(?) as the only operand of a DB, DW, DD, DT, or DQ definition statement, it produces an uninitialized data block. This data block is suitable for references to locations, which cannot or should not be

overwritten by the loader. Any other form of ? initializes the data block with data of an unknown value.

Example

```
DBPTR       DD  TABLE     ;16-bit OFFSET,
;                          THEN 16-bit SEG
;                          BASE VALUE
LIST        DD  2 DUP(?)
HIGH_NUM    DD  4294967295  ;MAXIMUM
LOW_NUM     DD  -4294967295 ;MINIMUM
PI          DD  3.14159     ;FLOATING POINT
;                           SINGLE PRECISION
EPSILON     DD  1.0E-7      ;FLOATING POINT
;                           SCIENTIFIC NOTATION
```

DQ
Define Quadword

Purpose

Use the DQ pseudo-op to define a variable or to initialize memory. DQ reserves four words (8-bytes).

Format

[*variable-name*] DQ *expression*[,...]

Remarks

If you enter a *variable-name*, the DQ pseudo-op defines *variable-name* as a variable. Do not follow *variable-name* with a colon (:). If *variable-name* is followed by a colon (:), it is considered a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant, in which case:

  Integers are produced unless a decimal point or scientific notation is used. Then floating point is produced.

  Note: Refer to the assembler options /R and /E for their effect on floating-point constants.

  The number of constants is limited only by the length of the line. Use a comma to separate each element.

- A question mark (?) for undefined initialization.

- A duplicated clause, for example:

```
repeat-count DUP expression
```

Note: If you use an expression in the form of *repeat count* DUP(?) as the only operand of a DB, DW, DD, DT, or DQ definition statement, it produces an uninitialized data block. This data block is suitable for references to locations, which cannot or should not be overwritten by the loader. Any other form of ? initializes the data block with data of an unknown value.

The *expression* entry cannot be:

- An external reference
- A variable
- A label.

Example

```
LONG_REAL   DQ  3.14159265 ;DECIMAL MAKES IT REAL
STRING      DQ  'AB'       ;NO MORE THAN 2 CHARS
HIGH_NUM    DQ  18446744073709551615    ;MAXIMUM
LOW_NUM     DQ -18446744073709551615    ;MINIMUM
SPACER      DQ  2 DUP(?)     ;UNINITIALIZED DATA
FILLER      DQ  1 DUP(?,?)   ;INITIALIZE WITH
;                             INDETERMINATE VALUE
HEX_REAL    DQ  0FDCBA9A98765432105R
PI          DQ  3.14159      ;FLOATING POINT
;                             DOUBLE PRECISION
EPSILON     DQ  1.0E-14      ;FLOATING POINT
;                             SCIENTIFIC NOTATION
```

DT
Define Tenbytes

Purpose

Use the DT pseudo-op to define a variable or to initialize memory. DT produces 10 bytes of packed decimal.

Format

[*variable-name*] DT *expression*[,...]

Remarks

If you enter *variable-name*, the DT pseudo-op defines *variable-name* as a variable. *Variable-name* must not be followed by a colon (:). If *variable-name* is followed by a colon (:), it is considered a code-relative label with the attribute of NEAR, and only addressable in the code segment (CS).

The *expression* entry can be one of the following:

- A constant, in which case:

  Packed decimal is produced unless a decimal point or scientific notation is used. Then 10-byte temporary format floating-point is produced.

  Note: Refer to the assembler options /R and /E for their effect on floating-point constants.

  The number of constants is limited only by the length of the line. Use a comma to separate each element.

- A question mark (?) for undefined initialization.

- A duplicated clause, for example:

```
repeat-count DUP (expression)...
```

Note: If you see an expression in the form of *repeat count* DUP(?) as the only operand of a DB, DW, DD, DT, or DQ definition statement, it produces an uninitialized data block. This data block is suitable for references to locations, which cannot or should not be

overwritten by the loader. Any other form of ? initializes the data block with data of an unknown value.

The *expression* entry cannot be:

- An external reference
- A variable
- A label.

Example

```
ACCUMULATOR     DT  ?
STRING          DT  'CD'    ;NO MORE THAN
;                            2 CHARACTERS
PACKED_DECIMAL  DT  1234567890
TEMP_REAL       DT  3.14159 ;FLOATING POINT
;                            TEMP REAL FORMAT
EPSILON         DQ  1.0E-16 ;FLOATING POINT
;                            SCIENTIFIC NOTATION
```

DW
Define Word

Purpose

Use the DW pseudo-op to define a variable or to initialize memory. DW reserves one word (2 bytes).

Format

[*variable-name*] DW *expression*[,...]

Remarks

If *variable-name* is entered, the DW pseudo-op defines *variable-name* as a variable. *Variable-name* must not be followed by a colon (:). If *variable-name* is followed by a colon (:), it is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant. The number of constants is limited only by the length of the line. Use a comma to separate the elements.
- A question mark (?) for undefined initialization.
- An address expression.
- A duplicated clause, for example:

```
repeat-count DUP (expression)...
```

Note: If you use an expression in the form of *repeat count* DUP(?) as the only operand of a DB, DW, DD, DT, or DQ definition statement, it produces an uninitialized data block. This data block is suitable for references to locations, which cannot or should not be overwritten by the loader. Any other form of ? initializes the data block with data of an unknown value.

Example

```
ITEMS    DW  TABLE,TABLE+10,TABLE+20  ;OFFSETS
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(1))  ;2,1,1,1,2,1,1,1
DISTANCE DW  START_TAB - END_TAB
;               DIFFERENCE OF LABELS IS CONSTANT
```

ELSE

Purpose

Each conditional pseudo-op can be used with the ELSE pseudo-op to provide the statements to be considered for conditional assembly. The ELSE pseudo-op allows the assembly of the statements following it when the IF condition is not true.

Format

ELSE

Remarks

Only one ELSE is permitted for a given IF. A conditional pseudo-op with more than one ELSE or an ELSE without a conditional pseudo-op causes an error. ELSE does not have an operand.

Note: The conditional pseudo-ops can be nested to any level. Their use is not limited to use within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation. All conditional pseudo-ops use the format:

```
IFxx operand
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

Example

```
IF DEFBUF
    BUF DB 100 DUP(0)
ELSE
    EXTRN BUF:BYTE
ENDIF
```

END

Purpose

The END pseudo-op has two functions:

- END identifies the end of the source program.
- The *expression* on the END pseudo-op identifies the symbol that is the name of the entry point (starting address).

Format

END *[expression]*

Remarks

All source files must have the END pseudo-op as the last statement. Any lines following the END statement are ignored by the assembler.

When LINK builds an application program from one or more object modules, it needs to know where the entry point is for DOS to pass initial control. If you do not specify an entry point, none is assumed. Only one module can identify a label as the entry point by specifying that label on its END statement. Any module lacking a DOS entry point must not have an entry point identified on its END statement. If you fail to define an entry point for the main module, your program may not be able to initialize correctly. It will assemble and link without error, but it cannot run.

Note: For applications that are .COM files, the entry point must be at 100H. This entry point must be identified on the END statement of the first module in the list of linked .OBJ modules. Otherwise, an error occurs during the EXE2BIN conversion process.

For example, observe the use of the END statements in the two sample source modules, TRYCOM1 and TRYCOM2, which are included with the IBM Macro Assembler/2 software.

Example

The following example is the END statement for the section of code that starts with the name BEGIN.

```
END BEGIN
```

ENDIF

Purpose

ENDIF ends the corresponding IF conditional pseudo-op. Each IF conditional pseudo-op must be ended by a matching ENDIF conditional pseudo-op.

Format

ENDIF

Remarks

If the IF conditional pseudo-op is not ended by an ENDIF, an **unterminated conditional** message is produced at the end of each pass of the assembler. An ENDIF without a matching IF causes an error. ENDIF does not have an operand.

Note: The conditional pseudo-ops can be nested to any level. They are not limited to use within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation. All conditional pseudo-ops use the format:

```
IFxx [expression]
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

Example

```
IF debug
   EXTRN dump:FAR
   EXTRN trace:FAR
   EXTRN breakpoint:FAR
ENDIF
```

ENDM

Purpose

End each MACRO, REPT, IRP, and IRPC pseudo-op with the ENDM pseudo-op.

Format

ENDM

Remarks

If the ENDM pseudo-op is not used with the MACRO, REPT, IRP, and IRPC pseudo-ops, an error occurs. An unmatched ENDM also causes an error.

If the assembler produces an error message stating that it found the end-of-file on the source and cannot find an END statement when there was an END, the likely cause is a missing ENDM or ENDIF statement. Without ENDM, the assembler treats the rest of the source as part of the MACRO definition.

Note: The *name field* is not allowed. Do not confuse the ENDM pseudo-op with other ending pseudo-ops that do require the name of the block being ended, such as ENDP or ENDS.

Example

```
addup  MACRO  ad1,ad2,ad3
       MOV    AX,ad1    ;;first parameter in AX
       ADD    AX,ad2    ;;add next two parameters
       ADD    AX,ad3    ;;leave sum in AX
       ENDM
```

ENDP

Purpose

Every PROC pseudo-op must be ended with the ENDP pseudo-op.

Format

procedure-name ENDP

Remarks

If the ENDP pseudo-op is not used with the PROC pseudo-op, an error occurs. An unmatched ENDP also causes an error.

Note: See the PROC pseudo-op in this chapter for more detail and examples of ENDP use.

Example

```
 PUSH   AX                ; Push third parameter
 PUSH   BX                ; Push second parameter
 PUSH   CX                ; Push first parameter
 CALL   ADDUP             ; Call the procedure
 ADD    SP,6              ; Bypass the pushed parameters
 .
 .
 .
ADDUP   PROC   NEAR       ; Return address for near call
                          ;   takes two bytes
        PUSH   BP         ; Save base pointer - takes two more
                          ;   so parameters start at 4th byte
        MOV    BP,SP      ; Load stack into base pointer
        MOV    AX,[BP+4]  ; Get first parameter
                          ;   4th byte above pointer
        ADD    AX,[BP+6]  ; Get second parameter
                          ;   6th byte above pointer
        ADD    AX,[BP+8]  ; Get third parameter
                          ;   8th byte above pointer
        POP    BP         ; Restore base
        RET               ; Return
ADDUP   ENDP
```

In this example, three numbers are passed as parameters for the procedure ADDUP. Parameters are often passed to procedures by pushing them before the call so that the procedure can read them off the stack.

ENDS

Purpose

Every SEGMENT and STRUC pseudo-op must end with a corresponding ENDS pseudo-op.

Format

structure-name ENDS
or
segname ENDS

Remarks

If the ENDS pseudo-op is not used with the corresponding SEGMENT or STRUC pseudo-op, an error occurs. An unmatched ENDS also causes an error.

Note: See SEGMENT and STRUC pseudo-ops in this chapter for more details and examples of the use of ENDS.

Example

```
CONST   SEGMENT  word public 'CONST'
SEG1    DW       ARRAY_DATA
SEG2    DW       MESSAGE_DATA
CONST   ENDS
```

EQU

Purpose

The EQU pseudo-op assigns the value of *expression* to *name*.

Format

name EQU *expression*

Remarks

If the *name* already has a value other than *expression* or if *expression* is external, a message is produced. Unlike the = (equal sign), the name entry for EQU cannot be redefined to be different from a previous EQU definition of that same symbol.

The *expression* entry can be any of the following:

- A symbol
- An index reference
- A segment prefix and operands
- An instruction name
- A record expression
- A 16-bit absolute constant
- A floating-point value.

Example

```
B    EQU [BP+8]     ;(an index reference)
P8   EQU DS:[BP+8]  ;(a seg prefix and operand)
CBD  EQU AAD        ;(an instruction name)
EMP  EQU 6          ;(a constant value)
```

Note: Optionally a TYPE attribute can be assigned to the symbol by using the PTR operator. For example:

```
VECTOR  EQU   DWORD PTR 4
```

.ERR/.ERR1/.ERR2

Purpose

The .ERR, .ERR1, and .ERR2 pseudo-ops cause errors at the points at which they occur in the source file.

Format

.ERR
  or
.ERR1
  or
.ERR2

Remarks

The .ERR pseudo-op causes an error regardless of the pass. .ERR1 causes an error on the first pass only. .ERR2 causes an error on the second pass only. If you use the /D option to request a first pass listing, the .ERR1 error message appears on the screen and in the listing file. Unlike the other conditional error pseudo-ops, it causes a warning, not an unrecoverable error.

You can place these pseudo-ops within conditional assembly blocks and macros to tell which blocks are being expanded.

Example

This example ensures that you define either the DOS or the XENIX symbol. If you define neither, the assembler assembles the nested ELSE condition and produces an error message. The .ERR pseudo-op causes an error on each pass.

```
IFDEF DOS
      .
      .
      .
ELSE
      IFDEF XENIX
      .
      .
      .
      ELSE
      .ERR
      ENDIF
ENDIF
```

.ERRB/.ERRNB

Purpose

The .ERRB and .ERRNB pseudo-ops test the given *string*.

Format

.ERRB <*string*>
or
.ERRNB <*string*>

Remarks

If *string* is blank, the .ERRB pseudo-op produces an error. If *string* is not blank, .ERRNB produces an error. The string can be a name, number, or expression. You must provide the angle brackets (< >).

You can test for the existence of parameters by using these pseudo-ops within macros.

Example

```
WORK  MACRO  REALARG,TESTARG
      .ERRB  <REALARG>     ;; ERROR IF NO PARAMETERS
      .ERRNB <TESTARG>     ;; ERROR IF MORE THAN ONE PARAMETER
             .
             .
             .
      ENDM
```

In this example, the pseudo-ops ensure that only one argument is passed to the macro. If no argument is passed to the macro, the .ERRB pseudo-op produces an error. If more than one argument is passed, the .ERRNB pseudo-op produces an error.

.ERRDEF/.ERRNDEF

Purpose

The .ERRDEF and .ERRNDEF pseudo-ops test whether *name* has been defined.

Format

.ERRDEF *name*
or
.ERRNDEF *name*

Remarks

f *name* is defined as a label, a variable, or a symbol, the .ERRDEF ɔseudo-op produces an error. If you have not defined *name*, .ERRNDEF ɔroduces an error. When *name* is a forward reference, the assembler ɔonsiders it undefined on the first pass and defined on the second ɔass.

Example

n this example, .ERRDEF ensures that SYMBOL is not defined before ɜntering the blocks, and .ERRNDEF ensures that you defined SYMBOL ɜomewhere within the blocks.

```
ERRDEF   SYMBOL
FDEF     CONFIG1
         .
         .   SYMBOL  EQU  0
         .
NDIF
FDEF     CONFIG2
         .
         .   SYMBOL  EQU  1
         .
NDIF
ERRNDEF  SYMBOL
```

.ERRE/.ERRNZ

Purpose

The .ERRE and .ERRNZ pseudo-ops test the value of an *expression.*

Format

.ERRE *expression*
or
.ERRNZ *expression*

Remarks

If the *expression* evaluates to be false (zero), the .ERRE pseudo-op produces an error. If the *expression* evaluates to be true (not zero), the .ERRNZ pseudo-op produces an error. The *expression* must evaluate to an absolute value and cannot contain forward references.

Example

```
BUFFER  MACRO  COUNT,BNAME
        .ERRE  COUNT LE 128       ;; RESERVE MEMORY, BUT
        BNAME  DB    COUNT DUP (0);;   NO MORE THAN 128 BYTES
        ENDM

BUFFER  128,BUF1        ; DATA RESERVED - NO ERROR
BUFFER  129,BUF2        ; ERROR PRODUCED
```

In this example, the .ERRE pseudo-op checks the boundaries of a parameter that the program passes to the macro BUFFER. If COUNT is less than or equal to 128, the expression that the pseudo-op tests is true (not zero) and the pseudo-op produces no error. If COUNT is greater than 128, the expression is false (zero) and the pseudo-op produces an error.

.ERRIDN/.ERRDIF

Purpose

The .ERRIDN and .ERRDIF pseudo-ops test whether two strings are identical.

Format

.ERRIDN <*string1*>,<*string2*>
or
.ERRDIF <*string1*>,<*string2*>

Remarks

If the strings are identical, the .ERRIDN pseudo-op produces an error. To be identical, each character in *string1* must match the corresponding character in *string2*. These tests are case-sensitive. If the strings are different, .ERRDIF produces an error. The strings can be names, numbers, or expressions. You must provide the angle brackets (< >).

Example

```
ADDEM  MACRO AD1,AD2,SUM
       .ERRIDN <ax>,<AD2> ;; ERROR IF AD2 IS ax
       .ERRIDN <AX>,<AD2> ;; ERROR IF AD2 IS AX
       MOV   AX,AD1       ;; WOULD OVERWRITE IF AD2 WERE AX
       ADD   AX,AD2
       MOV   SUM,AX       ;; SUM MUST BE REGISTER OR MEMORY
       ENDM
```

In this example, the .ERRIDN pseudo-op protects against passing the AX register as the second parameter, because the macro does not work if this happens. This example uses the .ERRIDN pseudo-op twice to protect against the two most likely spellings.

EVEN

Purpose

The EVEN pseudo-op causes the program counter to go to an even boundary (an address that begins a word). This ensures that the code or data that follows is aligned on an even boundary.

Format

EVEN

Remarks

If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP (no operation) so that the counter reaches an even boundary. An error message occurs if EVEN is used with a byte-aligned segment. If the program counter is already at an even boundary, EVEN does nothing.

Example

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

```
EVEN
```

After: PC points to 001A hex (26 decimal).

EXITM

Purpose

Use the EXITM pseudo-op when a block contains a pseudo-op that tests for some condition and you want to end the REPT, IRP, IRPC, or MACRO call when the test proves that the remainder of the expansion is not required. When an EXITM pseudo-op is run, the expansion is stopped immediately, and any remaining expansion or repetition is not produced.

Format

EXITM

Remarks

If the block containing EXITM is nested within another block, the outer level continues to be expanded; however, in all cases, the assembler still checks for an ENDM for the MACRO or repeat pseudo-op and still returns an error if ENDM is missing. ENDM and EXITM are not interchangeable.

Example

```
DSEG    SEGMENT
        .
        .
        .
SYM     =     0
        REPT  16
;;CHECK FOR PARA BOUNDARY
        IF    ($-DSEG) MOD 16  EQ  0
        EXITM  ;;QUIT IF PADDED TO BOUNDARY
        ENDIF
SYM     =     SYM + 1
        DB    SYM ;;PRODUCE NUMBERED PADDING
        ENDM
```

EXTRN

Purpose

The EXTRN pseudo-op specifies symbols used in this assembler module whose attributes are defined in another assembler module.

Format

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

Remarks

The symbol in the other assembler module must be declared PUBLIC. If the EXTRN pseudo-op is given within a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the EXTRN pseudo-op outside all segments and either use an explicit segment prefix or an ASSUME pseudo-op.

The *name* entry is the symbol that is defined in another assembler module. The *type* entry can be BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR, or ABS.

Note: If the *type* of EXTRN is ABS, the defining module must define it as a constant.

For example:

```
FOO     =    5
PUBLIC   FOO
```

Example

IN THE SAME SEGMENT	IN ANOTHER SEGMENT
IN MODULE 1: CSEG SEGMENT PUBLIC TAGN ⋮ TAGN: ⋮ CSEG ENDS	**IN MODULE 1:** CSEGA SEGMENT PUBLIC TAGF ⋮ TAGF: ⋮ CSEGA ENDS
IN MODULE 2: CSEG SEGMENT EXTRN TAGN:NEAR ⋮ JMP TAGN CSEG ENDS	**IN MODULE 2:** EXTRN TAGF:FAR CSEGB SEGMENT ⋮ JMP TAGF CSEGB ENDS

GROUP

Purpose

The GROUP pseudo-op associates a group *name* with one or more segments, and causes all labels and variables defined in the given segments to have addresses relative to the beginning of the group, rather than to the segments where they are defined.

Format

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

Remarks

The *seg-name* entry can be

- A unique segment name assigned by the SEGMENT pseudo-op.
- A SEG *variable* operator.
- A SEG *label* operator.

The GROUP pseudo-op does not affect the order in which segments of a group are loaded. You can specify the loading order by:

- Using the **/A** or **/S** options when activating the assembler
- Specifying the *'class'* entry in the SEGMENT pseudo-op
- Responding to the **Object Module** prompt of the linker by specifying the object modules in the desired order.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do belong to the group. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65535 bytes.

Group names can be used with the ASSUME pseudo-op and as an operand prefix with the segment override operator (:).

Note: A group name must not be used in more than one GROUP pseudo-op in any source file. If several segments within the source file belong to the same group, all segment names must be given in the same GROUP pseudo-op.

Example

The following example shows how to use the GROUP pseudo-op to combine segments:

In Module A:

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

In Module B:

```
CGROUP GROUP ZZZ
ZZZ    SEGMENT
       ASSUME CS:CGROUP
       .
       .
       .
ZZZ    ENDS
```

The next example shows how to set DS with the paragraph number of the group called DGROUP.

As immediate:

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

In assume:

```
ASSUME  DS:DGROUP
```

As an operand prefix:

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

Notes:

1. DW FOO returns the offset of the symbol within its segment.
2. DW DGROUP:FOO returns the offset of the symbol within the group.

The next example shows how you can use the GROUP pseudo-op to create a .COM file type.

```
  PAGE  ,132
  TITLE GRPCOM - USE GROUP TO CREATE .COM FILE
;USE EXE2BIN TO CONVERT GRPCOM.EXE
; TO GRPCOM.COM.
CG  GROUP  CSEG,DSEG ;ALL SEGS IN ONE GROUP
DISPLAY MACRO   TEXT
        LOCAL   MSG
DSEG    SEGMENT BYTE PUBLIC 'DATA'
MSG     DB      TEXT,13,10,"$"
DSEG    ENDS
;;MACRO PRODUCES PARTLY IN DSEG,
;;PARTLY IN CSEG
        MOV     AH,9
        MOV     DX,OFFSET CG: MSG
;;NOTE USE OF GROUP NAME
;;IN PRODUCING OFFSET
        INT     21H
        ENDM
DSEG    SEGMENT BYTE PUBLIC 'DATA'
;INSERT LOCAL CONSTANTS AND WORK AREAS HERE
DSEG    ENDS
CSEG    SEGMENT BYTE PUBLIC 'CODE'
  ASSUME CS:CG,DS:CG,SS:CG,ES:CG ;SET BY LOADER
        ORG   100H ;SKIP TO END OF THE PSP
ENTPT   PROC  NEAR ;COM FILE ENTRY AT 100H
     DISPLAY "USING MORE THAN ONE SEGMENT"
     DISPLAY "YET STILL OBEYING .COM RULES"
        RET             ;NEAR RETURN TO DOS
ENTPT ENDP
CSEG  ENDS
      END     ENTPT
```

IFxxxx
Conditional Pseudo-ops

Purpose

You can use each conditional pseudo-op with the ELSE and ENDIF pseudo-ops to provide the statements to be considered for conditional assembly. The Macro Assembler assembles the statements following the IF pseudo-op only if this condition is true.

Format

Conditional pseudo-ops use the format:

```
IFxxxx operand
    .
    .
    .
[ELSE] (optional)
    .
    .
    .
ENDIF
```

Remarks

You can nest the conditional pseudo-ops to any level. They are not limited to use within a macro. The assembler must know any operand to a conditional on pass 1 to avoid errors and incorrect evaluation.

IF

Format

IF *expression*

Remarks

This is true if *expression* is not 0.

IFE

Format

IFE *expression*

Remarks

This is true if *expression* is 0.

IF1

Format

IF1

Remarks

This is true on pass 1. IF1 does not have an operand.

Note: See the discussion of Pass 1 and Pass 2 in the IBM *Macro Assembler/2 Assemble, Link, and Run* book.

IF2

Format

IF2

Remarks

This is true on pass 2. IF2 does not have an operand.

Note: See the discussion of Pass 1 and Pass 2 in the IBM *Macro Assembler/2 Assemble, Link, and Run* book.

IFDEF

Format

IFDEF *symbol*

Remarks

This is true if *symbol* has been defined as a label, variable, or symbol.

IFNDEF

Format

IFNDEF *symbol*

Remarks

This is true if *symbol* has not yet been defined as a label, variable, or symbol.

IFB

Format

IFB <*operand*>

Remarks

This is true if the *operand* is blank. The angle brackets around <*operand*> are required.

IFNB

Format

IFNB <*operand*>

Remarks

This is true if the *operand* is not blank. Use IFB and IFNB for testing when dummy parameters are supplied. The angle brackets around the <*operand*> are required.

IFIDN

Format

IFIDN <*operand-1*>,<*operand-2*>

Remarks

This is true if the string *operand-1* is identical to the string *operand-2*. The angle brackets around the operands are required.

IFDIF

Format

IFDIF <*operand-1*>,<*operand-2*>

Remarks

This is true if the string *operand-1* is different from the string *operand-2*. The angle brackets around the operands are required.

ENDIF

Format

ENDIF

Remarks

Each IF pseudo-op must have a matching ENDIF to end the conditional; otherwise, the instruction produces an **unterminated conditional** message at the end of each pass of the assembler. An ENDIF without a matching IF causes an error. ENDIF does not have an operand.

ELSE

Format

ELSE

Remarks

You can use each conditional pseudo-op with the ELSE pseudo-op, which allows the assembler to produce alternate code when the opposite condition exists. Only one ELSE is permitted for a given IF. A conditional pseudo-op with more than one ELSE or an ELSE without a conditional pseudo-op causes an error. ELSE does not have an operand.

INCLUDE

Purpose

The INCLUDE pseudo-op inserts source statements from an alternate source file into the current source file. If you use the INCLUDE pseudo-op, you need not repeat a sequence of statements that are common to several source files.

Format

INCLUDE [drive:][path]*filename*[.ext]

Remarks

The *filename* entry can be any valid DOS file specification.

You can specify directories in INCLUDE path names with either the backslash (\) or the forward slash (/).

If you plan to set a search path with the /I option, you should specify a filename but no path name with the INCLUDE pseudo-op.

The assembler first looks for the INCLUDE filename in any paths specified with the /I option. It then checks the current directory. If the named file is not found, the assembler displays an error message and stops.

When the assembler finds an INCLUDE pseudo-op, it opens the source file and assembles its statements. When the assembler reaches the end of the file, assembly resumes with the statement following the pseudo-op.

The assembler prints the letter C between the assembled code and the source listing of each line that is assembled from an INCLUDE file.

The assembler allows nested INCLUDES; however, each open INCLUDE file requires an additional block of memory from available work space.

When nesting INCLUDE files using DOS, you need to add a FILES=*n* parameter to the CONFIG.SYS file, where *n* is greater than the default

of 8. Otherwise, if there is any nesting of INCLUDEs, the assembler may not be able to open all of the INCLUDE files, plus the necessary output files required by MASM. For each level of nesting of INCLUDE, the assembler must open one more output file. An error will result if the assembler attempts to open more files than the currently specified number of FILES in CONFIG.SYS.

Example

```
IF1
INCLUDE B:MYMACR.LIB
ENDIF
```

IRP

Purpose

The IRP pseudo-op, used in combination with the ENDM pseudo-op, designates a block of statements to be repeated, once for each parameter in the list enclosed by angle brackets. Each repetition substitutes the next item in the *<operandlist>* entry for every occurrence of *dummy* in the block.

Format

```
IRP dummy,<operandlist>

    .
    .
    .
  ENDM
```

Remarks

You must enclose the *<operandlist>* entry in angle brackets. If a null (< >) parameter is found in *<operandlist>*, the *dummy* name is replaced by a null value. If the parameter list is empty, the IRP pseudo-op is ignored and no statements are copied. The assembler processes the block once with each occurrence of *<dummy>* replaced by an element from *<operandlist>*.

The IRP—ENDM block does not have to be within a macro definition.

Example

In this example, the assembler produces the code DB 1 through DB 10.

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

In the next example:

```
IRP  DUMMY,<"first line",13,10,"second line",13,10>
DB   DUMMY
ENDM
```

The assembler produces the code:

```
DB       "first line"
DB       13
DB       10
DB       "second line"
DB       13
DB       10
```

IRPC

Purpose

The assembler repeats the statements in the block once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of *dummy* in the block.

Format

IRPC *dummy,string* (or <*string*>)

.
.
.
ENDM

Remarks

The IRPC pseudo-op is similar to the IRP pseudo-op except that a *string* is used instead of <*operandlist*>, and the angle brackets around the string are optional. The string should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters.

The IRPC–ENDM block does not have to be within a macro definition.

Example

In this example, the assembler produces the code DB 1 through DB 8:

```
IRPC    X,12345678
DB      X
ENDM
```

LABEL

Purpose

The LABEL pseudo-op defines the attributes of *name*:

- Segment: current segment being assembled
- Offset: current position within this segment
- Type: an operand.

Format

name LABEL *type*

Remarks

The *type* entry must be one of the following:

For data areas:

- BYTE
- WORD
- DWORD
- QWORD
- TBYTE
- *structure name*
- *record name*

For executable code:

- NEAR
- FAR

Example

To refer to a data area but use a length different from the original definition of that area:

```
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
```

To define multiple entry points within a procedure:

```
SUBRT   PROC    FAR
        .
        .
        .
SUB2    LABEL   FAR ;SHOULD HAVE
;                   SAME ATTRIBUTE
;                   AS CONTAINING
;                   PROC
        .
        .
        .
        RET
SUBRT   ENDP
```

.LALL/.SALL/.XALL

Purpose

You use the .LALL pseudo-op to list the complete macro text for all expansions. The .LALL pseudo-op also allows for the display of false conditional blocks by using the pseudo-ops .TFCOND, .LFCOND, .SFCOND or the MASM command line option /X.

The .SALL pseudo-op suppresses listing of all text and object code that the macros produce.

The .XALL pseudo-op is the default condition; the assembler lists a source line only if it produces object code.

Format

```
.LALL
or
.SALL
or
.XALL
```

.LFCOND
(List False Conditionals)

Purpose

You use the .LFCOND (List False Conditionals) to list conditional blocks that are evaluated as false.

Format

.LFCOND

Remarks

.LFCOND does not have an operand. You can end this state either by issuing .TFCOND, which reverts to the default state concerning listing of false conditionals (but with the default state redefined as being in the opposite state,) or by issuing the .SFCOND, which suppresses the listing of false conditionals.

The assembler does not print false conditionals within macros when .LALL is set.

Note: See the discussion of false conditional blocks in Chapter 2, "Getting Started," in this book for details.

.LIST/.XLIST

Purpose

These two pseudo-ops control output to the listing file.

Format

```
 .LIST
or
 .XLIST
```

Remarks

If a listing is not being created, these pseudo-ops have no effect. The .LIST is the default condition. When the assembler finds an .XLIST, the assembler does not list the source and the object code until it finds a .LIST pseudo-op.

LOCAL

Purpose

LOCAL lets you code multiple macro calls which contain labels. When you run LOCAL, the assembler creates a unique symbol for each *dummy* entry in *dummylist* and substitutes that symbol for each occurrence of the *dummy* in the expansion. You use these unique symbols to define a label within a macro, thus eliminating duplicate definitions of labels on successive expansions of the macro.

Format

LOCAL *dummylist*

Remarks

The assembler allows the LOCAL pseudo-op only inside a MACRO pseudo-op. The symbols created by the assembler range from ??0000 to ??FFFF. You must avoid using the form ??*nnnn* for your own symbols, because doing so produces a label or a symbol with multiple definitions. If you use LOCAL statements, they must be the first statements in the MACRO pseudo-op. You cannot place COMMENT or semicolon remarks between MACRO and LOCAL.

You can use multiple LOCAL statements if the *dummylist* is too long to fit on one line, or if you want a vertical list of LOCAL symbols.

Example

```
DISPLAY MACRO   TT
        LOCAL   AGAIN
;; DISPLAY MSG POINTED TO BY BX TT TIMES
        MOV     CX,TT
        MOV     AH,9
        MOV     DX,BX
AGAIN:  INT     21H
        LOOP    AGAIN
        ENDM
```

MACRO

Purpose

This pseudo-op produces a given sequence of statements from various places in your program, even though different parameters may be required each time you call the sequence.

Format

name MACRO [*dummylist*]

.
.
.
ENDM

Remarks

Use PURGE to delete the macro. Use the *name* entry to call the macro.

A macro consists of three essential parts:

- The MACRO pseudo-op, defining the *name* and the *dummylist*
- The body of the macro, containing the prototypes of statements to produce when you call the name of the macro.
- The ENDM pseudo-op, ending the definition of the macro.

The calling of a macro specifies the macro *name* (defined in the *name* field of the MACRO definition statement) as a pseudo-op optionally followed, after at least one blank, with the *parmlist*. For example:

name [*parmlist*]

Each time the name of the macro is called, the assembler produces the statements in the body of the macro and marks them in the assembler listing with a plus (+) symbol located to the left of the source line.

The *dummylist* is a series of one or more symbols, separated by a comma, defining the symbols that are replaced when referred to within the body of the macro. When the macro is expanded, each

symbol defined in the *dummylist* is replaced by an entry in the *parmlist*, specified on the macro call statement. The *parmlist* is a series of elements separated by a comma.

The relative positions of the elements are important, because dummy parameters correspond to *parmlist* symbols in order. The assembler replaces the first symbol defined in the *dummylist* with the first element defined in the *parmlist* of the calling statement, the second symbol with the second element, and so on for the entire list. The substitution is a replacement of the dummy symbol with the character string specified as the corresponding parameter. The number of parameters used when the macro is called need not be the same as the number of *dummylist* entries. If you have more parameters than *dummylist* entries, the assembler ignores the extra ones; if fewer, the extra *dummylist* entries become nulls. If the parameter has a prefix of % in the *parmlist*, the assembler substitutes the dummy with the value of the parameter. If the parameter does not have a prefix of % in the *parmlist*, the assembler substitutes the dummy with the character string of the parameter.

Because the assembler treats a blank after an element in the *parmlist* as a null value for the next positional parameter, blanks must not appear in the *parmlist*. To show that you want a parameter to contain a significant blank or comma, enclose the entire parameter element in angle brackets. For example:

```
PUSHVEC MACRO   PARM1,PARM2
        MOV     AX,PARM1
        PUSH    AX
        MOV     AX,PARM2
        PUSH    AX
        ENDM
        .
        .
        .
        PUSHVEC DS,<OFFSET VARNAME>
;PUSH DWORD VECTOR OF VARNAME ONTO STACK
```

You can also use angle brackets to produce variable lengths of results. For example:

```
STRING  MACRO   NUMBERS
        DB      NUMBERS
        ENDM
            .
            .
            .
        STRING  <1,2,3,4>
;PRODUCE 4 BYTES OF INTEGER NUMBERS
```

Note: The assembler recognizes a dummy parameter in a MACRO only as a dummy parameter. The assembler changes register names such as AL and BX in the expansion if they are used as dummy parameters. The assembler does not replace dummy parameters in comments. See Chapter 2, "Getting Started," in this book for additional information about macros.

Example

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

When the call is made, for example:

```
        GEN     ED,KISER,SUM
```

The assembler produces the following code:

```
        MOV     AX,ED
        ADD     AX,KISER
        MOV     SUM,AX
```

NAME

Purpose

The NAME pseudo-op assigns a module a name.

Format

NAME *module-name*

Remarks

The *module-name* cannot be a reserved word.

The assembler names every module and selects the name from the following list in the order shown:

1. The *module-name* in the NAME pseudo-op statement. The assembler allows only one NAME pseudo-op per assembly.
2. If a NAME pseudo-op is not present, the first six characters of text in a TITLE statement, if these six characters are valid for use in the name field.
3. If a TITLE statement is not present or if the first six characters are not valid for use in a name field, the default name A becomes the module name.

The assembler passes the *module-name* to the linker and the Library Manager.

ORG

Purpose

The ORG pseudo-op sets the location counter to the value of *expression*. Subsequent instructions are generated beginning at this new location.

Format

ORG *expression*

Remarks

The assembler must know all names used in *expression* on pass 1, and the value must be either absolute or in the same segment as the location counter.

The *expression* must evaluate to a 2-byte absolute number.

You can use the dollar sign ($) to refer to the current value of the location counter.

Example

```
ORG  120H
ORG  $+2        ;SKIP NEXT 2 BYTES
```

To conditionally skip to the next 256-byte boundary:

```
CSEG   SEGMENT PAGE
BEGIN  = $
       .
       .
       .
       IF ($-BEGIN) MOD 256
;IF NOT ALREADY ON 256 BYTE BOUNDARY
       ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
       ENDIF
```

%OUT

Purpose

The %OUT pseudo-op displays progress through a long assembly or displays the value of conditional assembly parameters.

Format

%OUT *text*

Remarks

The assembler lists the *text* entry on the display during assembly when the assembler finds %OUT.

Note: The assembler sends %OUT text to the screen even if the output list file is also sent to the screen by CON.

Example

Example 1:

```
        IF   IBM
        %OUT IBM VERSION
        ENDIF

        IF2
        %OUT STARTING SECOND PASS...
        ENDIF
```

Example 2:

```
INNER   MACRO   TEXT,VAL
        %OUT    TEXT  VAL
        ENDM
        .
        .
        .
HERE    =       $ - CSEG
        INNER   <CURRENT LOCATION>,%HERE
```

PAGE

Purpose

The PAGE pseudo-op controls the length and width of each listing page. Place the PAGE pseudo-op in the source file to control the format of the listing file produced during assembly.

Format

PAGE [*operand-1*] [,*operand-2*]
or
PAGE +

Remarks

Using the PAGE pseudo-op without the operand entries causes the printer to go to the top of the page and increases the page number by 1.

Each page of the listing produced by the assembler contains a chapter number and a page number separated by a dash. The assembler increases the page number when a page is full or when PAGE (no operand) is encountered. The assembler increases the chapter number only when it finds PAGE +. Both cause the printer to go to the top of the next page.

The *operand-1* entry can be:

- A number from 10 to 255 showing the number of lines printed per page (length). The page length without a specified number is 58.

  Note: The assembler prints a printer eject character on the 58th line of each page if you do not specify a number. This setting allows a margin at the top and bottom of each page with the standard 66 lines per page.

- The + sign shows that the chapter number is to be increased by 1 and the page number set to 1. If you use the + sign, it cannot be used with any other operand.

Use the *operand-2* entry to control the width of the page. The page width without a specified number is 80. You can specify *operand-2* from 60 to 132.

Note: The PAGE pseudo-op does not set the printer to the desired line width. For proper formatting of the listing, initialize the printer to operate at a corresponding line width. Do this before printing the listing file by some other method, such as the DOS MODE command.

You can imbed the printer control characters that set the character size into the comment field of the PAGE statement. (For an example, see the SKELEXE.ASM example file included with the IBM Macro Assembler/2 software).

PROC

Purpose

The PROC pseudo-op identifies a block of code. By dividing the code into blocks, each of which performs a distinct function, you can clarify the overall function of the complete module.

The PROC pseudo-op also identifies the type of linkage used by any RET instruction contained within a block of code.

Format

procedure-name PROC NEAR
or
procedure-name PROC FAR
.
.
.
RET
.
.
.
procedure-name ENDP

Remarks

You can execute the block of code identified by the PROC pseudo-op in-line, jump to it, or start it with a CALL. For the *procedure-name* to be accessible to any external CALL or other external reference, you must also use the PUBLIC pseudo-op. If the PROC is the entry point of an .EXE module that is called directly by the DOS loader, code the PROC attribute as FAR. If the PROC is called from code that has another ASSUME CS value, you must use the FAR attribute.

The NEAR specification causes any RET coded within the procedure to be an intra-segment return that pops a return offset from the stack. You can call a NEAR subroutine only from the same segment. However, FAR causes RET to be an inter-segment return that pops both a return offset and a segment base from the stack. You can call a FAR subroutine from any segment; a FAR subroutine is usually called from a segment other than the one containing the subroutine.

Example

In this example, the lower subroutine is called by the upper subroutine.

```
            PUBLIC  FAR_NAME
FAR_NAME    PROC    FAR
            CALL    NEAR_NAME
            RET     ;POPS RETURN OFFSET AND SEG BASE VALUE
FAR_NAME    ENDP


            PUBLIC  NEAR_NAME
NEAR_NAME   PROC    NEAR
                .
                .
                .
            RET     ;POPS ONLY RET OFFSET
NEAR_NAME   ENDP
```

You can call the lower subroutine directly from a NEAR segment by using:

```
            CALL NEAR_NAME
```

A FAR segment can indirectly call the second subroutine by first calling the upper subroutine with:

```
            CALL FAR_NAME
```

A CALL to a forward-referenced symbol assumes the symbol is NEAR. If that symbol is FAR, the CALL must have an override, for example:

```
            CALL FAR PTR FORREF
```

PUBLIC

Purpose

The PUBLIC pseudo-op makes defined symbols available to other programs that are to be linked. The information referred to by the PUBLIC pseudo-op is passed to the linker.

Format

PUBLIC *symbol*[,...]

Remarks

Symbol can be a variable or a label (including PROC labels). Register names and any symbols defined by EQU or = to floating-point numbers or integers larger than 2 bytes are incorrect entries.

Note: You should use the PUBLIC pseudo-op to identify any symbol names you would like to reference within the CodeView debugger. See the IBM *Macro Assembler/2 Assemble, Link, and Run* book for information about CodeView.

Example

```
        PUBLIC GETINFO
GETINFO PROC   FAR
        PUSH   BP     ;SAVE CALLER'S REG
        MOV    BP,SP  ;GET ADDR PARMS
                      ;BODY OF SUBROUTINE
        POP    BP     ;RESTORE CALLER'S REG
        RET           ;RETURN TO CALLER
GETINFO ENDP
```

PURGE

Purpose

The PURGE pseudo-op deletes the definition of a specified MACRO entry, letting you reuse space.

Format

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

Remarks

It is not necessary to PURGE a MACRO before redefining it. You may use PURGE to recover memory during assembly by deleting the contents of unreferenced macros. An **Out of Memory** condition can occur if a large, general-purpose macro library is included.

Example

The pseudo-op:

```
PURGE   MACRONAME
```

performs the same function as redefining a macro with no contents, as in:

```
MACRONAME   MACRO
            ENDM
```

In the following example, assume that MAC1 is a macro included in MACRO.LIB.

```
INCLUDE   MACRO.LIB
PURGE     MAC1
MAC1      ;CALLS THE PURGED MACRO
          ; BUT PRODUCES NOTHING
```

.RADIX

Purpose

The .RADIX pseudo-op lets you change the default RADIX (decimal) to any base from 2 to 16.

Format

.RADIX *expression*

Remarks

The *expression* entry is in decimal RADIX regardless of the current RADIX.

The .RADIX pseudo-op does not affect **real numbers** initialized as variables with DD, DQ, or DT. Please note that this works differently from the IBM Macro Assembler Version 2.00, which **always** ignored the current RADIX when initializing variables with DD, DQ, or DT.

When using .RADIX 16, be aware that if the hex constant ends in either B or D, the assembler thinks that the B or D is a request to cancel the current RADIX specification with a base of BINARY or DECIMAL, respectively. In such cases, add the H base override (just as if .RADIX 16 were not in use).

Example

The statement:

```
.RADIX   16
DW   120B
```

produces an error, because 2 is not a valid binary number.
The correct specification is:

```
DW   120BH
```

The following example:

```
.RADIX   16
DW   89CD
```

also produces an error, because C is not a valid decimal number. The correct specification is:

```
DW   89CDH
```

The dangerous case is when no error is produced. For example:

```
.RADIX   16
DW   120D
```

produces a constant whose value is 120 decimal, not '120D' hex, which might have been the intended value.

The following two move instructions are the same:

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

The following example:

```
.RADIX  4
DQ  15.0   ;Treated as decimal
```

produces a constant whose value is 15 decimal because 15.0 is a real number. However, if you leave off the decimal point, the following:

```
.RADIX  4
DQ  15     ;uses current radix
```

produces a syntax error because 5 is not a valid number in RADIX 4.

RECORD

Purpose

A record is a bit pattern you define to format bytes and words for bit-packing. The *recordname* itself becomes a pseudo-op used to reserve memory.

Format

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

Remarks

The *recordname* and *fieldname* entries are unique identifiers and you must use them. You must enter the colon (:) between the *fieldname* and *width*. The *fieldname* entry is the name of the field. The value of *fieldname*, when used in an expression, is the shift count needed to move the field to the far right. The MASK operator returns a bit mask for the field.

The *width* entry evaluates to a constant from 1 to 16, and specifies the number of bits defined by *fieldname*. If the total width is larger than 8 bits, the assembler uses 2 bytes; otherwise, it uses 1 byte. If the total number of bits defined is fewer than 8 (a byte) or 16 (a word), the assembler right-justifies the fields into the least-significant bit positions of the byte or word.

The *exp* entry contains the default value for the field. If the field is at least 7 bits wide, you can initialize it to an ASCII character (for example, FIELD:7 = 'Z').

Records can be used in expressions in the form:

```
recordname <[init-list]>
```

The <[*init-list*]> entry is an optional list of the initialization values. The angle brackets must be coded as shown, but the values do not have to be given.

To initialize a record, use the form:

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

The *name* entry is an optional name for the first byte or word of the reserved memory. The *recordname* specifies the name you assigned to the record from the RECORD pseudo-op that defines the format and optional default field values. The <[*exp*][,...]> entry specifies a list of field-initialization or optional override values so that trailing fields default.

In the second form above, the angle brackets (shown outside the square brackets) around the second *exp* are required. If you leave [*exp*] blank, either the default value applies, or the value is unknown.

Example

Define the record fields; begin with most significant fields first:

```
MODULE RECORD R:7,E:4,D:5
```

Fields are 7 bits, 4 bits, and 5 bits; the assembler gives no default value. Most significant byte first, this looks like:

```
    RRRR RRRE EEED DDDD
```

To reserve its memory:

```
STG_FLD  MODULE  <7,,2>
```

This defines R=7 and D=2, and leaves E unknown; it produces 2 bytes, the least significant byte first:

```
        02 0E
```

Define the record fields:

```
AREA  RECORD FLA:8='A',FLB:8='B'
```

To reserve its memory:

```
CHAR_FLD AREA <,'P'>
```

This defines FLA='A' (the default) and changes FLB='P'.

Note: Be aware of the 132-character limit on the length of lines allowed by the assembler when defining a RECORD. Because of the 132-character limit, you might want to select short names

for the fields in order to fit them on one line. See the FNSTSW instruction in Chapter 4, "Instruction Mnemonics," in this book for a sample RECORD defining the status bits in the 8087 STATUS WORD.

To use a field in the record:

```
DEFFIELD  RECORD   X:3,Y:4,Z:9
          .
          .
          .
TABLE     DEFFIELD 10 DUP(<0,2,255>)
          .
          .
          .
          MOV DX,TABLE[2]
;MOVE DATA FROM RECORD TO REGISTER
; OTHER THAN SEGMENT REGISTER
          AND DX,MASK Y
;MASK OUT FIELDS X AND Z
; TO REMOVE UNWANTED FIELDS
; The MASK of Y equals 1E00H
; 0001111000000000B (1E00H) IS THE VALUE
          MOV CL,Y    ;GET SHIFT COUNT
;                      9 IS THE VALUE
          SHR DX,CL   ;FIELD TO LOW-ORDER
; BITS OF REGISTER, DX IS NOW EQUAL TO
; THE VALUE OF FIELD Y
          MOV CL,WIDTH Y ;GET NUMBER OF
;                         BITS IN FIELD
;                         4 IS THE VALUE
;The WIDTH of Y equals 4.
```

REPT

Purpose

The assembler repeats the block of statements between REPT and ENDM the number of times in the *expression* entry.

Format

REPT *expression*

.
.
.
ENDM

Remarks

If the *expression* entry contains any external or undefined terms, an error is produced.

The REPT—ENDM block does not have to be within a macro definition.

Example

This example produces a series of ordered bytes up to a paragraph boundary.

```
DSEG    SEGMENT
          .
          .
          .
SYM     =     0
        REPT  16
;;CHECK FOR PARA BOUNDARY
        IF    ($-DSEG) MOD 16  EQ  0
        EXITM
        ENDIF
SYM     =     SYM + 1
        DB    SYM
        ENDM
```

SEGMENT

Purpose

At run time, each instruction and each variable of your program lies within some segment. Use the SEGMENT pseudo-op to define all code-producing instructions as being within a segment. Your assembly module can be a part of a segment, a whole segment, parts of several segments, several whole segments, or a combination of these.

Format

segname SEGMENT [*align-type*][*combine-type*] ['*class*']

.
.
.

segname ENDS

Remarks

The assembler assigns the symbol entry for a segment attribute of an *align-type* entry, a *combine-type*, and a character string defining *'class'* (if present); the assembler also keeps the length of the segment.

Note: If you specify two or three attributes, separate them with a space.

The *align-type* entry can be PARA, BYTE, WORD, or PAGE. The definition of these types are:

- PARA (Default): This align-type specifies that the segment begin on a paragraph boundary (the address is divisible by 16). That is, the least significant hexadecimal digit of the address equals 0H.
- BYTE: This align-type specifies that the segment can begin anywhere.
- WORD: This align-type specifies that the segment begins on a word boundary (an even address where the least significant bit of the address equals 0).

- PAGE: This align-type specifies that the segment begins on a page boundary (an address divisible by 256 and whose two least-significant hexadecimal digits are equal to 00H).

The *combine-type* entry can be PUBLIC, COMMON, AT *expression*, STACK, or no entry (which defaults to not combinable). The definitions for these types are:

- PUBLIC: This combine-type specifies that this segment is connected to others of the same name when linked.
- COMMON: This combine-type specifies that this segment and all other segments of the same name that are linked together begin at the same address, and thus overlap. The length of a linked COMMON is the maximum of the linked segments.
- AT *expression*: This combine-type specifies that this segment is located at the 16-bit paragraph number evaluated from a given expression. The expression can be any valid constant; however, it cannot be a forward reference. You cannot use the AT *expression* to force loading of code at a fixed address; rather, it permits you to define labels or variables at fixed offsets within fixed areas of memory, such as Read Only Memory or the vector space in low memory. No code is produced for this segment.
- STACK: This combine-type specifies that this segment is part of the run-time stack segment, called last-in, first-out (LIFO) using the assembler instructions such as: PUSH, POP, CALL, INT, IRET, POPF, PUSHF, and RET. If you are writing the application to be an .EXE file, a STACK segment is required; if a .COM file, a STACK segment should not be assembled. The linker issues a warning message if no STACK segment is defined. Ignore that linker warning message if you are converting the application from an .EXE file to a .COM file.
- MEMORY: This combine-type, although recognized by the assembler, cannot be used with the linker and should not be used in your code. If used, it is treated as PUBLIC.

The *'class'* entry is the name (must be enclosed in single quotes) used to group segments at link time.

You can nest segment definitions. When segments are nested, the assembler acts as if they are not nested and processes them by attaching the second part of the split segment to the first. When the

assembler detects the ENDS pseudo-op, the assembler takes up the next segment, completes it, and continues. Overlapping segments are not permitted.

Example

The following example combines logical segments into a physical segment:

In module A—

```
SEGA    SEGMENT  PUBLIC
        ASSUME   CS:SEGA
        .
        .
        .
SEGA    ENDS
```

In module B—

```
SEGA  SEGMENT  PUBLIC
      ASSUME   CS:SEGA
      . (Note: LINK adds this
      .  segment to same named
      .  segment in module A)
SEGA  ENDS
```

Nesting of Segments:

```
CSEG  SEGMENT
      .
      .
      .
      MOV  AX,BX
DSEG  SEGMENT
  P1  DW   6
DSEG  ENDS
      .  (CSEG continues ...)
      .
      .
CSEG  ENDS
```

.SEQ

Purpose

.SEQ tells the assembler to arrange the segments in the object file in the order they appear in the source file.

Format

.SEQ

Remarks

.SEQ cancels the /A option or .ALPHA pseudo-op.

Example

```
.SEQ
DATA  SEGMENT
      .
      .
      .
DATA  ENDS

CODE  SEGMENT
      .
      .
      .
CODE  ENDS
```

Segment DATA will precede segment CODE in the object module.

.SFCOND

Purpose

.SFCOND suppresses the listing of false conditional blocks.

Format

.SFCOND

Remarks

.SFCOND does not have an operand. End the state caused by .SFCOND either by issuing .TFCOND (which reverts to the default state concerning listing of false conditionals but with the default state redefined as being in the opposite state) or by issuing .LFCOND (which forces the listing of false conditionals).

Note: See the discussion of false conditional blocks in Chapter 2, "Getting Started," in this book for detail.

STRUC

Purpose

The STRUC pseudo-op is like the RECORD pseudo-op except that STRUC has a multi-byte capability. Reserving memory for and initializing the values in a STRUC block is the same as for the RECORD pseudo-op.

Format

structurename STRUC

.
.
.

[*fieldnames*] (a DEFINE pseudo-op) *exp*

.
.

structurename ENDS

Remarks

The *structurename* itself becomes a pseudo-op that reserves memory. Inside the STRUC-ENDS block, DB, DW, DD, DQ, and DT pseudo-ops can reserve space. It is not necessary for a data field defined within a STRUC to have a variable name. Any labels on a DEFINE pseudo-op inside the STRUC-ENDS block become the *fieldnames*. First values given in the STRUC-ENDS block are default values for the various fields. These values, or *fields* are of two types:

- Overridable
- Not overridable.

A simple field (a field with only one entry) can be overridden. A multiple field (a field with more than one entry) cannot be overridden.

If the *field* contains a string, another string can override it. However, if the overriding string is shorter than the initial string, the assembler pads the space to the right with blanks. If the overriding string is longer, the assembler cuts off the extra characters.

The format that refers to an item defined within the STRUC-ENDS block (for example, an operand) is:

```
variable.field
```

where the *variable* represents a variable. The *field* entry represents a label given to a define data pseudo-op inside the STRUC-ENDS block. (Code the period as shown.) The value of the *field* entry is the offset within the addressed structure.

You cannot use any pseudo-ops, except DB, DD, DQ, DT, and DW, within a STRUC-ENDS structure definition.

Note: A simple DD field initialized with 0 or any value other than "?" cannot be overridden by an address; it can only be overridden by a constant value. You must initialize a DD field with "?" in a STRUC if you want to later override that DD field with an address. This differs from the previous version of the Macro Assembler, the IBM Personal Computer Macro Assembler Version 2.00, which did allow addresses to override DD fields regardless of how the field was initialized.

Example

Definition:

```
STR     STRUC
FIELD1  DB 1,2            ;CANNOT BE OVERRIDDEN
FIELD2  DB 10 DUP(?)      ;CANNOT BE OVERRIDDEN
FIELD3  DB 5              ;CAN BE OVERRIDDEN
FIELD4  DB 'BREINER'      ;CAN BE OVERRIDDEN
FIELD5  DD ?              ;CAN BE OVERRIDDEN
FIELD6  DD 0              ;CANNOT BE OVERRIDDEN BY AN ADDRESS
STR     ENDS
```

Allocation:

```
DB_AREA STR <,,7,'TOM C'> ;OVERRIDES 3rd & 4th
;                          FIELDS ONLY
```

Reference:

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

SUBTTL

Purpose

The SUBTTL pseudo-op specifies a subtitle listed on the line after the title on each page heading.

Format

SUBTTL [*text*]

Remarks

The *text* entry is cut off after 60 characters. You can give any number of SUBTTLs in a program.

Subtitles are turned off unless the SUBTTL pseudo-op is used. To turn off subtitles again for a portion of the listing after the SUBTTL pseudo-op is given, use another SUBTTL pseudo-op but with a null entry in the *text* string.

SUBTTL does not cause a skip to the top of the page.

SUBTTL is usually followed by PAGE.

.TFCOND

Purpose

The .TFCOND pseudo-op toggles the default setting that controls the listing of false conditionals, then sets the current condition to that of the new default, thus ending the effect of a .SFCOND or .LFCOND pseudo-ops.

Format

.TFCOND

Remarks

The .TFCOND pseudo-op does not have an operand. The .TFCOND pseudo-op sets the current and default setting to the non-default condition. If the /X option is given on the MASM command line for a file that contains .TFCOND, the /X option reverses the effect of the .TFCOND pseudo-op.

Note: See the discussion of false conditional blocks in Chapter 2, "Getting Started," in this book for details.

TITLE

Purpose

The TITLE pseudo-op specifies a title to be listed on the second line of each page.

Format

TITLE *text*

Remarks

If more than one TITLE is given, an error results. The first six characters of the title are used as the module name unless a NAME pseudo-op is used. If you do not use a NAME or TITLE pseudo-op, the module name defaults to "A".

If a TITLE is not given, or if any of the six characters of the TITLE text are not valid in a name field, the TITLE is not used for the name.

The TITLE pseudo-op is placed in the source file to control the formatting and the printing of the listing file produced during assembly.

The *text* entry is limited to 60 characters.

Chapter 4. Instruction Mnemonics

This chapter describes the instructions used by your IBM Macro Assembler/2.

The instructions are arranged in alphabetical order for ease of reference.

See Chapter 2, "Getting Started," in this book for general information about instructions.

AAA
ASCII Adjust for Addition

Purpose

AAA corrects the result in AL of adding two unpacked decimal operands, resulting in an unpacked decimal sum.

Format

AAA

Remarks

If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary carry flag has been set, 6 is added to AL and 1 is added to AH. AF and CF are set. The new value of AL has an upper half-byte of all zeroes, and the lower half-byte is the number between 0 and 9 created by the above addition.

Logic

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) <- (AL) + 6
  (AH) <- (AH) + 1
  (AF) <- 1
  (CF) <- (AF)
  (AL) <- (AL) & 0FH
```

Flags

Affected— AF,CF
Undefined— OF,PF,SF,ZF

Encoding

00110111

37

Example

```
AAA    ;AFTER THE ADDITION
```

AAD
ASCII Adjust for Division

Purpose

AAD adjusts the dividend in AL before a following instruction divides two unpacked decimal operands, so that the result of the division is an unpacked decimal quotient.

Format

AAD

Remarks

The high-order byte (AH) of the accumulator **is multiplied** by 10 and added to the low byte (AL). The result is stored into AL. AH is set to zero.

Logic

(AL) <- (AH)*0AH + (AL)
(AH) <- 0

Flags

Affected— PF,SF,ZF
Undefined— AF,CF,OF

Encoding

11010101 00001010

D5 0A

Example

```
AAD   ;BEFORE THE DIVISION
```

AAM
ASCII Adjust for Multiply

Purpose

AAM corrects the result in AX of multiplying two unpacked decimal operands, resulting in an unpacked decimal product.

Format

AAM

Remarks

The contents of AH are replaced by the result of dividing AL by 10. Then, the contents of AL are replaced by the remainder of that division, that is, by AL module 10.

Logic

(AH) <- (AL)/0AH
(AL) <- (AL)%0AH

Flags

Affected— PF,SF,ZF
Undefined— AF,CF,OF

Encoding

11010100 00001010

D4 0A

Example

```
AAM   ;AFTER THE MULTIPLICATION
```

AAS
ASCII Adjust for Subtraction

Purpose

AAS corrects the result in the AL register of subtracting two unpacked decimal operands, resulting in an unpacked decimal difference.

Format

AAS

Remarks

If the lower half of AL is greater than 9, or if the auxiliary carry flag is set, 6 is subtracted from AL and 1 is subtracted from AH. The AF and CF flags are set. The old value of AL is replaced by a byte whose upper half-byte is all zeroes and whose lower half-byte is a number from 0 to 9 created by the above subtraction.

Logic

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) <- (AL) - 6
  (AH) <- (AH) - 1
  (AF) <- 1
  (CF) <- (AF)
  (AL) <- (AL) & 0FH
```

Flags

Affected— AF,CF
Undefined— OF,PF,SF,ZF

Encoding

00111111

3F

Example

```
AAS    ;AFTER THE SUBTRACTION
```

ADC
Add with Carry

Purpose

ADC adds the two operands, adds 1 if the CF flag is set, and returns the result to the *destination* (left-most operand).

Format

ADC *destination,source*

Remarks

If the carry flag is set to 1, ADC adds 1 to the sum of the two operands before storing the result into the *destination* (left-most operand). If the carry flag is set to 0, 1 is not added.

Logic

If (CF) = 1, (DEST) <- (LSRC) + (RSRC) + 1
ELSE (DEST) <- (LSRC) + (RSRC)

Flags

Affected— AF,CF,OF,PF,SF,ZF

Memory or Register Operand with Register Operand

Encoding

000100dw modregr/m

10 + dw modregr/m

If d = 1, LSRC = REG, RSRC = EA,DEST = REG
If d = 0, LSRC = EA, RSRC = REG, DEST = EA

Example

```
ADC   AX,SI
ADC   DI,BX
ADC   CH,BL

ADC   DX,MEM_WORD
ADC   AX,BETA[SI]
ADC   CX,ALPHA[BX][SI]

ADC   BETA[DI],BX
ADC   ALPHA[BX][SI],DI
ADC   MEM_WORD,AX
```

Immediate Operand to Accumulator

Encoding

0001010w data

14 + w data

If w=0, LSRC = AL, RSRC = data, DEST = AL
If w=1, LSRC = AX, RSRC = data, DEST = AX

Example

```
ADC  AL,3
ADC  AL,VALUE_13_IMM
ADC  AX,333
ADC  AX,IMM_VAL_777
```

Immediate Operand to Memory or Register Operand

Remarks

If an immediate-data byte is added to a register or memory, that byte is sign-extended to 16 bits before the addition. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding

100000sw mod010r/m data

80 + sw mod010r/m data

LSRC = EA,RSRC = data,DEST = EA

Example

```
ADC   BETA[SI],4
ADC   ALPHA[BX][DI],IMM4
ADC   MEM_LOC,7396

ADC   BX,IMM_VAL_987
ADC   DH,65
ADC   CX,432
```

ADD
Addition

Purpose

ADD adds the two operands and returns the result to the *destination* operand.

Format

ADD *destination,source*

Remarks

ADD stores the sum of the two operands into the *destination* (leftmost) operand.

Logic

(DEST) <- (LSRC) + (RSRC)

Flags

Affected— AF,CF,OF,PF,SF,ZF

Memory or Register Operand with Register Operand

Encoding

000000dw modregr/m

00 + dw modregr/m

If d=0, LSRC = REG,RSRC = EA, DEST = REG
If d=1, LSRC = EA, RSRC = REG, DEST = EA

Example

Register to register:

```
ADD   AX,BX
ADD   CX,DX
ADD   DI,SI
ADD   BX,BP
```

Memory to register:

```
ADD   CX,MEM_WORD
ADD   AX,BETA[SI]
ADD   DX,ALPHA[BX][DI]
```

Register to memory:

```
ADD   GAMMA[BP][DI],BX
ADD   BETA[DI],AX
ADD   MEM_WORD,CX
ADD   MEM_BYTE,BH
```

Immediate Operand to Accumulator

Encoding

0000010w data

04 + w data

If w=0, LSRC = AL,RSRC = data, DEST = AL.
If w=1, LSRC = AX, RSRC = data, DEST = AX.

Example

```
ADD   AL,3
ADD   AX,456
ADD   AL,IMM_VAL_12
ADD   AX,IMM_VAL_8529
ADD   AX,IMM_VAL_6AB9H   ;DESTINATION AX
```

Immediate Operand to Memory or Register Operand

Remarks

If an immediate-data byte is added from a register or memory word, that byte is sign-extended to 16 bits before the addition. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding

100000sw mod000r/m data

80 + sw mod000r/m data

LSRC = EA, RSRC = data, DEST = EA

Example

Immediate to memory:

```
ADD   MEM_WORD,48
ADD   GAMMA[DI],IMM_84
ADD   DELTA[BX],IMM_SENSOR_5
```

Immediate to register:

```
ADD   BX,ORIG_VAL
ADD   CX,STANDARD_COUNT
ADD   DX,1776
```

AND
Logical AND

Purpose

AND does the bit logical conjunction of the two operands and returns the result to the *destination* operand.

Format

AND *destination,source*

Remarks

The two operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with zeroes in all other bit positions. The result is stored into the *destination* (leftmost) operand. The carry and overflow flags are reset to 0.

Logic

(DEST) <- (LSRC) & (RSRC)
(CF) <- 0
(OF) <- 0

Flags

Affected— CF,OF,PF,SF,ZF.
Undefined— AF

Memory or Register Operand with Register Operand

Encoding

001000dw modregr/m

20 + dw modregr/m

If d=1, LSRC = REG,RSRC = EA, DEST = REG
If d=0, LSRC = EA, RSRC = REG, DEST = EA

Example

Register to register:

```
AND   AX,BX
AND   CX,DI
AND   BH,CL
```

Memory to register:

```
AND   SI,MEM_NAME_WORD
AND   DX,BETA[BX]
AND   BX,GAMMA[BX][SI]
AND   AX,ALPHA[DI]
AND   DH,MEM_BYTE
```

Register to memory:

```
AND   MEM_NAME_WORD,BP
AND   ALPHA[DI],AX
AND   GAMMA[BX][DI],SI
AND   MEM_BYTE,AL
```

Immediate Operand to Accumulator

Encoding

0010010w data

24 + w data

If w=0, LSRC = AL, RSRC = data, DEST = AL
If w=1, LSRC = AX, RSRC = data, DEST = AX

Example

```
AND   AL,7AH
AND   AH,0EH
AND   AX,IMM_VAL_MASK3
```

Immediate Operand to Memory or Register Operand

Encoding

1000000w mod100r/m data

80 + w mod100r/m data

LSRC = EA, RSRC = data, DEST = EA

Example

Immediate to register:

```
AND   BL,10011110B
AND   CH,3EH
AND   DX,7A46H
AND   SI,987
```

Immediate to memory:

```
AND   MEM_WORD,7A46H
AND   MEM_BYTE,46H
AND   GAMMA[DI],IMM_MASK14
AND   CHI_BYTE[BX][SI],11100111B
```

Another example:

```
FLAGS   DB   ?
BITMASK EQU  20H
        .
        .
        .
        AND  FLAGS,0FFH-BITMASK ;TURN OFF
                                ; FLAG BIT
```

ARPL (80286P)
Adjust Requested Privilege Level

Purpose

ARPL is used to ensure that a selector option to a subroutine requests no more privilege than allowed. The right operand is the return link selector of the caller, a copy of the CS which defines the CPL of the caller.

Format

ARPL *destination,source*

Remarks

ARPL adjusts the RPL (requested privilege level) field of a selector in the first operand (a memory location or register) to the maximum of its original value and the value of the RPL field in the second operand (a register). If the ARPL instruction changes the RPL field of the left operand, the assembler sets the zero flag. Otherwise, the zero flag is cleared.

You must use the .286P pseudo-op to enable this instruction.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

```
if LSRC.RPL < RSRC.RPL then
  LSRC.RPL <- RSRC.RPL
  (ZF) <- 1
else (ZF) <- 0
```

Flags

Affected— ZF

Encoding

01100011 modregr/m
63 modregr/m

Example

```
ARPL   BX,CX
ARPL   [BP].ARG2,AX
```

BOUND (80286)
Detect Value Out of Range

Purpose

BOUND is used to ensure that a signed array index is within the limits defined by a 2-word block of memory.

Format

BOUND *destination,source*

Remarks

The first operand (a register) must be above or equal to the first word in the memory location referred to by the second operand, and below or equal to the second word in that memory location. The *source* operand must refer to memory.

The .286C pseudo-op is required.

Logic

```
if ((REG16) < (RSRC)) or ((REG16) > ((RSRC)+2)) then
  (SP) <- (SP)-2
  ((SP)+1:(SP)) <- FLAGS
  (IF) <- 0
  (TF) <- 0
  (SP) <- (SP)-2
  ((SP)+1:(SP)) <- (CS)
  (CS) <- (16H)
  (SP) <- (SP)-2
  ((SP)+1:(SP)) <- (IP)
  (IP) <- (14H)
```

Flags

None

Encoding

01100010 modregr/m

62 modregr/m

Example

```
          DW    $+4         ;Beginning address of ARRAY
          DW    ARRAY_END   ; DESCRIPTORS
ARRAY     DW    20 DUP(0)
ARRAY_END EQU   $

          BOUND SI,DWORD PTR ARRAY-4
```

CALL
Call a Procedure

Purpose

CALL pushes the offset address of the next instruction onto the stack (for an inter-segment call, the CS segment register is pushed first) and then transfers control to the *target* operand.

Direct calls and jumps can only be made to labels relative to CS, not to variables.

The assembler generates a NEAR or FAR call depending on whether the target procedure name is defined as NEAR or FAR. As shown in the indirect-call examples that follow, calls through variables can use the PTR operator to show the intended use of one word for NEAR calls, or two words for calls to FAR labels or procedures. Indirect calls using word registers (without square brackets) are of necessity NEAR calls.

Format

CALL *target*

Remarks

If this is an inter-segment call:

1. The stack pointer is decreased by 2, and the contents of the CS register are pushed onto the stack. CS is filled by the second word (segment) of the doubleword inter-segment pointer.
2. The stack pointer is decreased by 2.
3. The contents of the Instruction Pointer (IP) are pushed onto the stack.
4. The contents of the IP are replaced by the offset of the *target* destination (the offset of the procedure's first instruction).

An intra-segment or intra-group call performs steps 2, 3, and 4 only.

Logic

1) if inter-segment then
   (SP) <- (SP) - 2
   ((SP) + 1:(SP)) <- (CS)
   (CS) <- SEG
2) (SP) <- (SP) - 2
3) ((SP) + 1:(SP)) <- (IP)
4) (IP) <- DEST

Flags

None

Direct Intra-Segment or Intra-Group

Encoding

11101000 disp-low disp-high

E8 disp-low disp-high

DEST = (IP) + disp

Example

```
CALL  NEAR_LABEL
CALL  NEAR_PROC
```

Direct Inter-Segment

Encoding

10011010 offset-low offset-high seg-low seg-high

9A offset-low offset-high seg-low seg-high

DEST = offset, SEG = seg

Example

```
CALL  FAR_LABEL
CALL  FAR_PROC
```

Indirect Inter-Segment

Encoding

11111111 mod011r/m

FF mod011r/m

DEST = (EA),SEG = (EA + 2)

Example

```
CALL  DWORD  PTR  [BX]
CALL  DWORD  PTR  VARIABLE_NAME[SI]
CALL  MEM_DOUBLE_WORD
```

Indirect Intra-Segment or Intra-Group

Encoding

11111111 mod010r/m

FF mod010r/m

DEST = (EA)

Example

```
CALL  WORD  PTR  [BX] ;BX HAS OFFSET IN DATA
                       ; SEGMENT OF WORD THAT HAS
                       ; OFFSET OF ENTRY POINT
CALL  WORD  PTR  VARIABLE_NAME
CALL  WORD  PTR  [BX][SI]
CALL  WORD  PTR  [DI]
CALL  WORD  PTR  VARIABLE_NAME[BP][SI]
CALL  MEM_WORD
CALL  BX    ;REG HAS OFFSET IN CODE
            ; SEGMENT TO ENTRY POINT
```

DS is the implicit segment register in a register-indirect call, unless you use BP or specify an override. Use the implicit segment register to build the address that contains the offset (and segment, if a FAR call) of the *target* of the call. If BP is used, the segment register SS is used. However, if a segment prefix byte is explicitly specified; such as:

```
CALL  WORD  PTR  ES:[BP][DI]
```

then the segment register ES is used. An implicit segment register for indirect calls through variables or address-expressions is determined by the address-expression in the source line and the applicable ASSUME pseudo-op. See Chapter 3, "Pseudo Operations," in this book.

When you use CALL to transfer control, a RETurn is implied. With indirect CALLS, you must carefully ensure that the type of the CALL matches the type of matches the type of RETurn, or errors can result that are difficult to trace. Be sure CS is saved and restored.

The type of RETurn is determined by the PROC pseudo-op.

CALL (80286P)
Call a Procedure

Purpose

CALL pushes the offset of the next instruction onto the stack (for an inter-segment call the CS segment register is pushed first) and then transfers control to the *target* operand.

Direct calls and jumps can only be made to labels relative to CS, not to variables. NEAR is assumed unless FAR is stated in the instruction or in the declaration of the *target* label.

As shown in the indirect-call examples that follow, calls through variables can use the PTR operator to show the intended use of one word for NEAR calls, or two words for calls to FAR labels or procedures. Indirect calls using word registers (without square brackets) are of necessity NEAR calls.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Format

CALL *target*

Direct Intra-Segment and Indirect Intra-Segment

Remarks

There are two types of intra-segment calls. One is direct; the other is indirect. Use a NEAR label as the *target* address to specify a direct (IP-relative) intra-segment call. In this case, the 16-bit displacement is relative to the first byte of the next instruction.

For an indirect intra-segment call, the *target* address is taken from a register or variable pointer without modification; thus, it is not IP-relative. The indirect call is specified when the operand is any (16-bit) general, base, or index register or when the operand is a word variable. When the effective address is a variable, DS is the implied segment register for all EAs not using BP.

The return link, which is pushed to the TOS during the CALL, is the address of the instruction following the CALL.

Logic

```
if direct then
  IP <- IP + disp
  (SP) <- (SP - 2)
  ((SP + 1):(SP)) <- return link
else
  IP <- (addr)
  (SP) <- (SP -2)
  ((SP) + 1:(SP)) <- return link
```

Flags

None are affected, except when the task switch occurs; then all are affected.

Direct (IP-Relative) Intra-Segment

Encoding

11101000 disp-low disp-high

E8 disp-low disp-high

Example

```
CALL  NEAR_LABEL
```

Indirect Intra-Segment

Encoding

11111111 mod010r/m

FF mod010r/m

Example

```
CALL  SI                ;SI is a 16-bit index register
CALL  WORD PTR [SI]     ;operand is a word variable
CALL  POINTER_TO_PLACE ;operand is a word variable
```

Direct Virtual Address and Indirect Virtual Address (VA in DWORD Variable)

Remarks

The long calls transfer control using a Virtual Address Double Word (VADW), which can be included in the instruction itself or found in a DWORD variable. The selector part of the VADW determines the type of control transfer as follows:

1. If the selector names a descriptor for an executable segment, that selector replaces CS and the offset part of the VADW replaces IP, according to protection mechanism addressability and visibility.
2. If the selector names a call-gate descriptor, the offset part of the VADW is ignored, and the virtual address of the routine being entered is taken from the call-gate. If the routine being entered is more privileged, then a new stack (both SS and SP) is called from save areas in the task state segment for the new PL, and the wordcount field of the gate then determines how many words of argument list are copied from the old stack to the new. The VADW of the top of the old stack is pushed onto the new stack before the arguments are transferred, and the arguments are followed by the VADW return link.
3. If the selector names a task gate descriptor, the context of the current task is saved in its Task State Segment (TSS), and the TSS named in the task-gate is used to load the new context. The selector for the outgoing task (from TR) is stored into the new TSS's link field, and the new task's Nested Task flag is set. The outgoing task is left marked busy, the new TSS is marked busy, and running resumes at the point at which the new task was last suspended.
4. If the selector names a TSS, the current task is suspended and the new task initiated as described above, except that there is no intervening gate.

In general, for the task-state to be considered correct, the following constraints upon the contents of the segment registers must be adhered to:

- For CS, the selector must name an executable segment. The RPL field of CS defines the CPL.
- SS must name a writable segment with privilege equal to CPL.
- DS and ES must either be zero (a value which represents an unloaded condition because it selects entry 0 in the GDT, which is incorrect), or must select a readable segment which is visible at the CPL. A segment is visible to any CPL which has a numerically smaller privilege level. Also, conforming segments, which must be able to be run, but which may be readable, are visible to any privilege level.

For long calls that do not cause a task-switch, the return link is the Virtual Address of the instruction following the CALL (that is, the CS and updated IP of the caller). Task switches called by CALL are linked by storing the TSS selector of the outgoing task in the incoming TSS link field and setting the Nested Task flag in the new task. Nested tasks must be ended by an IRET. IRET releases the nested task and follows the link to the calling task if the NT flag is set.

Logic

```
if executable segment selector then
  (SP) <- (SP - 2)
  ((SP) + 1:(SP)) <- (CS)
  (SP) <- (SP - 2)
  ((SP) + 1:(SP)) <- (IP)
  (CS) <- selector
  (IP) <- offset
else if call gate selector then
  if gate.DPL = CPL then
    (SP) <- (SP - 2)
    ((SP + 1):(SP)) <- (CS)
    (SP) <- (SP - 2)
    ((SP + 1):(SP)) <- (IP)
  else if gate.DPL < CPL then
    CPL = gate.DPL
    (SS) <- (TSS.SS)
    (SP) <- (TSS.SP)
    (CS) <- (call-gate.selector)
```

```
    (IP) <- (call-gate.offset)
else if call taskgate selector then
  (TSS.backlink) <- (TR)
  (TR) <- (task-gate.selector)
  (flags.NT) <- 1
else if call task state segment then
  (newTSS.backlink) <- (TR)
  (TR) <- instruction.selector
  (flags.NT) <- 1
```

Flags

Affected— AF,CF,DF,IF,NF,OF,PF,PLF,SF,TF,ZF

Encoding

10011010 VADW offset VADW selector

9A VADW offset VADW selector

Example

```
CALL   FAR_LABEL
CALL   CALL_GATE
CALL   TASK_GATE
CALL   TASK
```

Indirect Virtual Address (VA in DWORD variable)

Encoding

11111111 mod011r/m

FF mod011r/m

Example

```
CALL   DWORD PTR XXX
```

CBW
Convert Byte to Word

Purpose

CBW does a sign extension of the AL register into the AH register.

Format

CBW

Remarks

If the lower byte of the accumulator (AL) is less than 80H, AH is made zero. Otherwise, AH is set to FFH. This is equal to repeating bit 7 of AL through AH.

Logic

```
if (AL) < 80H then
  (AH) <- 0
else
  (AH) <- FFH
```

Flags

None

Encoding

10011000

98

Example

```
CBW
```

CLC
Clear Carry Flag

Purpose

CLC clears the CF flag.

Format

CLC

Remarks

The carry flag is reset to zero.

Note: See STC for the opposite function.

Logic

(CF) <- 0

Flags

Affected— CF

Encoding

11111000

F8

Example

```
CLC
```

CLD
Clear Direction Flag

Purpose

CLD clears the DF flag, causing the string operations to automatically increase the operand pointers.

Format

CLD

Remarks

The direction flag is reset to zero.

Note: See STD for the opposite function.

Logic

(DF) <- 0

Flags

Affected— DF

Encoding

11111100

FC

Example

```
CLD
```

CLI
Clear Interrupt Flag (Disable)

Purpose

CLI clears the IF flag and disables maskable external interrupts, which appear on the INTR line of the processor. (CLI does not disable non-maskable interrupts that appear on the NMI line.)

Format

CLI

Remarks

The interrupt flag is reset to zero.

Note: See STI for the opposite function.

In protected mode, CLI clears the IF flag if the current privilege level is at least as privileged as IOPL.

Logic

(IF) <- 0

Flags

Affected— IF

Encoding

11111010

FA

Example

```
CLI
```

CLTS (80286P)
Clear Task Switched Flag

Purpose

CLTS clears the task switched flag in the Machine Status Word (MSW).

Format

CLTS

Remarks

The task switched flag (TS flag) is set every time a task switch occurs. The TS flag is used to manage the 80287 math coprocessor in the following way. It traps every run of a WAIT or ESCAPE instruction (instructions used to control the math coprocessor) if the MSW.MP (math unit present) flag is set and the MSW.TS flag is set. Therefore, if the math unit is present and a task switch has been made since the last math-unit instruction was begun, the context of the math unit must be saved before a new instruction can be issued. The fault routine saves the context and resets the MSW.TS flag, or puts the task requesting the math unit in queue until the current instruction is completed. CLTS is a privileged instruction; it can be executed at level 0 only.

See Chapter 6, "80286./80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL=0 then
  MSW.TS <- 0
```

Flags

Affected— TS flag in MSW

Encoding

00001111 00000110
0F 06

Example

```
CLTS
```

CMC
Complement Carry Flag

Purpose

CMC complements the CF flag.

Format

CMC

Remarks

If the carry flag is 0, it is set to 1. If it is a 1, it is reset to 0.

Logic

(CF) < 1- (CF)

Flags

Affected— CF

Encoding

11110101

F5

Example

```
CMC
```

CMP
Compare Two Operands

Purpose

CMP subtracts the two operands causing the flags to be affected but does not return the result.

Format

CMP *destination,source*

Remarks

The *source* (rightmost) operand is subtracted from the *destination* (leftmost) operand. The flags are altered, but the operands remain unaffected.

The *source* (rightmost) operand must be of the same type (byte or word) as the *destination* operand. The only exception for CMP is comparing an immediate data byte with a memory word.

Logic

(LSRC) - (RSRC)

Flags

Affected— AF,CF,OF,PF,SF,ZF

Memory or Register Operand with Register Operand

Encoding

001110dw modregr/m

38 + dw modregr/m

If d=1, LSRC = REG, RSRC = EA.
If d=0, LSRC = EA, RSRC = REG.

Example

Register with register:

```
CMP   AX,DX
CMP   SI,BP
CMP   BH,CL
```

Register with memory:

```
CMP   MEM_WORD,SI
CMP   MEM_BYTE,CH
CMP   ALPHA[DI],DX
CMP   BETA[BX][SI],CX
```

Memory with register:

```
CMP   DI,MEM_WORD
CMP   CH,MEM_WORD
CMP   AX,GAMMA[BP][SI]
```

Immediate Operand with Accumulator

Encoding

0011110w data

3C + w data

If w=0, LSRC = AL, RSRC = data.
If w=1, LSRC = AX, RSRC = data.

Example

```
CMP   AL,6
CMP   AL,IMM_VALUE_DRIVE
CMP   AX,IMM_VAL_909
CMP   AX,999
```

Immediate Operand with Memory or Register Operand

Remarks

If an immediate-data byte is compared from a register or memory word, that byte is sign-extended to 16 bits before the comparison. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding

100000sw mod111r/m data

80 + sw mod111r/m data

LSRC = EA, RSRC = data

Example

Immediate with register:

```
CMP   BH,7
CMP   CL,19_IMM_BYTE
CMP   DX,IMM_DATA_WORD
CMP   SI,798
```

Immediate with memory:

```
CMP   MEM_WORD,IMM_DATA_BYTE
CMP   GAMMA[BX],IMM_BYTE
CMP   [BX][DI],6ACEH
```

CMPS/CMPSB/CMPSW
Compare Byte or Word String

Purpose

CMPS subtracts the byte (or word) operand addressed by DI from the operand addressed by SI; CMPS affects the flags but does not return the result. As a repeated operation, two strings are compared. With the appropriate repeat prefix, you can determine after which string element the two strings become unequal, thereby establishing an ordering between the strings.

Note that the operand indexed by DI is the rightmost operand in this instruction and that this operand is addressed using the ES register only. You cannot cancel this default.

Format

CMPS *source-string,dest-string*
or
CMPSB
or
CMPSW

Remarks

Using DI as an index into the extra segment, the *dest-string* right-most operand is subtracted from the *source-string* (left-most operand), which uses SI as an index. (This is the only string instruction where the DI-indexed operand appears as the right-most operand.) Only the flags are affected, not the operands. SI and DI are then increased, if the direction flag is reset (zero), or they are decreased, if DF = 1. (See the CLD and STD instructions.) Thus, they point to the next element of the strings being compared. The increase is 1 for byte strings, 2 for word strings.

Logic

(LSRC) - (RSRC)
if (DF) = 0 then
  (SI) <- (SI) + DELTA
  (DI) <- (DI) + DELTA
else
  (SI) <- (SI) - DELTA
  (DI) <- (DI) - DELTA

Flags

Affected— AF,CF,OF,PF,SF,ZF

Encoding

1010011w

A6 + w

If w=0, LSRC = (SI), RSRC = (DI),
    DELTA = 1 (BYTE).
If w=1, LSRC = (SI)+1:(SI), RSRC = (DI)+1:(DI),
    DELTA = 2 (WORD).

Example

```
MOV  SI,OFFSET STRING1
MOV  DI,OFFSET STRING2
CMPS STRING1,STRING2
    or
CMPS DS:BYTE PTR[SI],ES:[DI]
    or
CMPSB
```

The string instructions are unusual in that you:

1. Load SI with the offset of the *source* string.
2. Load DI with the offset of the *destination* string.
3. Can code each with or without symbolic memory operands:
   - If symbolic operands are coded, the assembler can check the addressability of them for you.
   - References that use hardware defaults should be coded using the forms without operands (CMPSB and CMPSW) to avoid the additional pointer information.
   - Do not use [BX] or [BP] addressing modes with the string instructions.
   - If operands are coded, you can cancel only the source segment, DS, not the destination segment, ES.
4. If the instruction mnemonic is coded without operands, the segment registers 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.

CWD
Convert Word to Doubleword

Purpose

CWD does a sign extension of the AX register into the DX register. See the IDIV instruction in this chapter.

Format

CWD

Remarks

The high-order bit of AX is repeated throughout DX.

Logic

if (AX) < 8000H then
  (DX) <- 0
else (DX) <- FFFFH

Flags

None

Encoding

10011001

  99

Example

```
CWD
```

DAA
Decimal Adjust for Addition

Purpose

DAA corrects the result in AL of adding two packed decimal operands, resulting in a packed decimal sum.

Format

DAA

Remarks

If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary carry flag has been set, 6 is added to AL and AF is set. If AL is greater than 9FH or if the carry flag has been set, 60H is added to AL and CF is set to 1.

Logic

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) <- (AL) + 6
  (AF) <- 1
if (AF) > 9FH or (CF) = 1 then
  (AL) <- (AL) + 60H
  (CF) <- 1
```

Flags

Affected— AF,CF,PF,SF,ZF
Undefined— OF

Encoding

00100111

27

Example

```
DAA
```

DAS
Decimal Adjust for Subtraction

Purpose

DAS corrects the result in the AL register of subtracting two packed decimal operands, resulting in a packed decimal difference.

Format

DAS

Remarks

If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary flag has been set, 60H is subtracted from AL and CF is set.

Logic

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) <- (AL) - 6
  (AF) <- 1
if (AL) > 9FH or (CF) = 1 then
  (AL) <- (AL) - 60H
  (CF) <- 1
```

Flags

Affected— AF,CF,PF,SF,ZF

Encoding

00101111

2F

Example

```
DAS
```

DEC
Decrease Destination by One

Purpose

DEC subtracts 1 from the operand and returns the result to that operand.

Format

DEC *destination*

Remarks

DEC decreases the specified operand, *destination*, by 1.

Logic

(DEST) <- (DEST) - 1

Flags

Affected— AF,OF,PF,SF,ZF

Register Operand (Word)

Encoding

01001 reg

48 + reg

DEST = REG

Example

```
DEC AX
DEC DI
DEC SI
```

Memory or Register Operand

Encoding

1111111w mod001r/m

FE + w mod001r/m

DEST = EA

Example

```
DEC   MEM_BYTE
DEC   MEM_BYTE[DI]
DEC   MEM_WORD
DEC   ALPHA[BX][SI]
DEC   BL
DEC   CH
```

DIV
Division, Unsigned

Purpose

DIV does an unsigned division of the double-length NUMR operand contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified *source* operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation).

Format

DIV *source*

Remarks

If the quotient is greater than MAX (when division by zero is attempted), QUO and REM are undefined, and a type 0 interrupt is produced. Flags are undefined in any DIV operation. Non-integral quotients are rounded to integers.

If the division results in a value larger than appropriate registers can hold, an interrupt of type 0 is produced. Flags are pushed onto the stack; IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack, and IP is then filled with the word at 0. Thus, this sequence includes a FAR call to the interrupt handling procedure whose segment and offset are stored at locations 2 and 0.

If the division result can fit in the appropriate registers, the quotient is stored in AL or AX (for word operands) and the remainder in AH or DX.

Logic

```
(temp) <- (NUMR)
if (temp)/(DIVR) > MAX,
 THE FOLLOWING SEQUENCE:
    (QUO),(REM) undefined
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- FLAGS
    (IF) <- 0
    (TF) <- 0
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- (CS)
    (CS) <- (2)
;CONTENTS OF LOCATIONS 2 AND 3
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- (IP)
    (IP) <- (0)
;CONTENTS OF LOCATIONS 0 AND 1
else
    (QUO) <- (temp)/(DIVR)
    (REM) <- (temp)%(DIVR)
```

Flags

Affected— No valid flags result
Undefined— AF,CF,OF,PF,SF,ZF

Encoding

1111011w mod110r/m

F6 + w mod110r/m

If w=0, NUMR=AX, DIVR=EA, QUO=AL,
REM=AH, MAX=FFH.
If w=1, NUMR=DX:AX, DIVR=EA, QUO=AX,
REM=DX, MAX=FFFH.

In the following examples, each memory operand can be any variable or valid expression so long as its type is the same as that of the source operand. For example, you could replace the NUMERATOR_WORD in the first example with the expression:

```
ARRAY_NAME[BX][SI] + 67
```

when ARRAY_NAME is of type WORD. Similarly, DIVISOR_BYTE could be:

```
RATE_TABLE[BP][DI]
```

when RATE_TABLE is of type BYTE.

Example

To divide a word by a byte:

```
MOV   AX,NUMERATOR_WORD
DIV   DIVISOR_BYTE
;QUOTIENT IN AL, REMAINDER IN AH
```

To divide a byte by a byte:

```
MOV   AL,NUMERATOR_BYTE
CBW    ;CONVERTS BYTE IN AL TO WORD IN AX
DIV   DIVISOR_BYTE
;QUOTIENT IN AL, REMAINDER IN AH
```

To divide a doubleword by a word:

```
MOV   DX,NUMERATOR_HI_WORD
MOV   AX,NUMERATOR_LO_WORD
DIV   DIVISOR_WORD
;QUOTIENT IN AX REMAINDER IN DX
```

To divide a word by a word:

```
MOV   AX,NUMERATOR_WORD
XOR   DX,DX  ;CLEAR HIGH WORD OF DX:AX DOUBLEWORD
DIV   DIVISOR_WORD
;QUOTIENT IN AX, REMAINDER IN DX
```

ENTER (80286)
Make Stack Frame for Procedure Parameters

Purpose

Use ENTER to create the stack frame required by most block-structured, high-level languages.

Format

ENTER *immediate-word,immediate-byte*

Remarks

The first operand specifies how many bytes of dynamic memory are reserved on the stack for the routine the assembler is entering. The second operand gives the nesting level of the routine within the high level-language source code. ENTER determines how many stack-frame pointers the assembler copies into the new stack frame from the preceding frame. The assembler uses BP as the current stack frame pointer.

If the second operand is 0, ENTER pushes BP, sets BP to SP, and subtracts the first operand from SP.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

The .286C pseudo-op is required.

Logic

```
(SP) <- (SP)-2
((SP)+1:(SP)) <- (BP)
(FP) <- (SP)
if LEVEL > 0 then
  Repeat (Level-1) times
    (BP) <- (BP)-2
    (SP) <- (SP)-2
    ((SP)+1:(SP)) <- (BP)
```

```
  End Repeat
  (SP) <- (SP)-2
  ((SP) + 1:(SP)) <- (FP)
End if
(BP) <- (FP)
(SP) <- (SP) - (LSRC)
```

Flags

Affected— None
Undefined— None

Encoding

11001000 data-low data-high

C8 data-low data-high

Example

A procedure with 12 bytes of local variables would have an

```
ENTER   12,0
```

at its entry point and a LEAVE instruction before every RET. The 12 local bytes would be addressed as negative offsets from [BP].

ESC
Escape

Purpose

ESC allows other processors to receive their instructions from the instruction stream and to use the addressing modes. The processor does no operation for the ESC instruction other than to get a memory operand and place it on a bus.

Format

ESC *external-opcode,source*

Logic

if mod $\neq$ 11, then data bus <- (EA)
if mod = 11, no operation.

Flags

None

Encoding

11011xxx modxxxr/m

D8 + xxx modxxxr/m

Example

```
ESC EXTERNAL_OPCODE, ADDRESS
```

Note: This opcode is a 6-bit number, which is split into the two 3-bit fields shown as xxx above.

F2XM1 (8087)
2 to the X power -1

Purpose

F2XM1 calculates the function $Y = 2^X-1$. X is taken from the top of the floating-point stack and must be in the range $0 \leq X \leq 0.5$. The result Y replaces X at the top of the floating-point stack.

Format

F2XM1

Remarks

F2XM1 produces an accurate result even when X is close to zero. To obtain $Y = 2^X$, add 1 to the result. You can raise values other than 2 to a power of X. For example:

$10X = 2X * \log_2 10$
$eX = 2X * \log_2 e$
$YX = 2X * \log_2 Y$

The 8087 has instructions, described in this chapter, for loading the constants $\log_2 10$ and $\log_2 e$, and you can use the FYL2X to calculate $X * \log_2 Y$.

Note: See FYL2X, FLDL2T, and FLDL2E for related information.

Logic

$ST <- 2^{ST} - 1$

Exception Flags

U,P (operands not checked)

Encoding

10011011 11011001 11110000

9B D9 F0

Without FWAIT:

11011001 11110000

D9 F0

Example

```
F2XM1           ;Y = 2 TO THE X POWER MINUS 1
```

FABS (8087)
Absolute value

Purpose

FABS changes the top element in the 8087 stack to its absolute value.

Format

FABS

Remarks

FABS makes the sign of the top element in the 8087 stack positive.

Logic

ST <- | ST |

Exception Flags

I

Encoding

10011011 11011001 11100001

9B D9 E1

Without FWAIT:

11011001 11100001

D9 E1

Example

```
FABS
```

FADD (8087)
Add Real

Purpose

FADD adds the *source* and *destination* operands and returns the sum to the *destination*.

Format

FADD
or
FADD *source*
or
FADD *destination,source*

Remarks

FADD stores the sum of the two operands in the *destination* (leftmost) operand. You can write FADD without operands, with only a *source*, or with a *destination* and a *source*.

Note: See FADDP and FIADD for related information.

Exception Flags

I,D,O,U,P

FADD (no operands) Stack form

Format

FADD [ST(1),ST]

Note: ST(1),ST are implied operands; they are not coded and are shown here for information only.

Remarks

FADD picks the *source* operand from the top of the 8087 stack and the *destination* operand from the next element in the 8087 stack. It then pops the 8087 stack, does the operation, and returns the result to the new top of the 8087 stack.

Note: FADD (no operands) is similar to the FADDP instruction, with ST(*i*) being ST(1).

Logic

ST(1) <- ST(1) + ST
pop 8087 stack

Encoding

10011011 11011110 11000001

9B DE C1

Without FWAIT:

11011110 11000001

DE C1

Example

```
FADD                    ;ADD REAL
```

FADD (source) Real Memory Form

Format

FADD short_real
or
FADD long_real

Remarks

The real-memory form permits you to use a real number in memory directly as a *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this form of the instruction.

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST + mem-op

Encoding

SHORT_REAL

10011011 11011000 mod000r/m disp-low disp-high
9B D8 mod000r/m disp-low disp-high

Without FWAIT:

11011000 mod100r/m disp-low disp-high

D8 mod100r/m disp-low disp-high

Example

```
FADD    SHORT_REAL
```

Encoding

LONG_REAL

10011011 11011100 mod000r/m disp-low disp-high
9B DC mod000r/m disp-low disp-high

Without FWAIT:

11011100 mod100r/m disp-low disp-high

DC mod100r/m disp-low disp-high

Example

```
FADD    LONG_REAL
```

FADD (destination,source) Register form

Remarks

Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format

FADD ST,ST(*i*)

Logic

ST <- ST + ST(*i*)

Encoding

10011011 11011000 11000(*i*)

9B D8 C0 + (*i*)

Without FWAIT:

11011000 11000(*i*)

D8 C0 + (*i*)

Example

```
FADD    ST,ST(1)        ;ADD REAL
FADD    ST,ST(0)        ;DOUBLE TOP OF STACK
FADD    ST,ST(7)
```

Format

FADD ST(*i*),ST

Logic

ST(*i*) <- ST(*i*) + ST

Encoding

10011011 11011100 11000(*i*)

9B DC C0 + (*i*)

Without FWAIT:

11011100 11000(*i*)

DC C0 + (*i*)

Example

```
FADD   ST(1),ST
FADD   ST(7),ST
```

FADDP (8087)
Add Real and Pop

Purpose

FADDP adds the *source* and *destination* operands, returns the sum to the *destination*, and pops the 8087 stack.

Format

FADDP *destination,source*

Remarks

FADDP stores the sum of the two operands into the *destination* (left-most) operand. FADDP picks the *source* operand from the ST register (the top element in the 8087 stack) and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack.

Note: FADD (no operands) is similar to the FADDP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST(*i*) + ST
pop 8087 stack

Exception Flags

I,D,O,U,P

Encoding

10011011 11011110 11000(*i*)

9B DE C0 + (*i*)

Without FWAIT:

11011110 11000(*i*)

DE C0 + (*i*)

Example

```
FADDP  ST(7),ST       ;ADD REAL AND POP
```

FBLD (8087)
Packed Decimal (BCD) Load

Purpose

FBLD converts a packed decimal to temporary real and loads (pushes) the result onto the 8087 stack.

Format

FBLD *source*

Remarks

FBLD converts the content of the *source* operand from packed decimal to temporary real and loads (pushes) the result onto the 8087 stack. FBLD preserves the sign of the *source*, including negative zero. FBLD is an exact operation; it loads the *source* with no rounding error.

FBLD assumes the packed decimal digits of the *source* are in the range (0-9)H. The instruction does not check for incorrect digits (A-F)H and the result of trying to load an incorrect digit is undefined.

Logic

push 8087 stack
ST <- mem-op

Exception Flags

I

Encoding

10011011 11011111 mod100r/m disp-low disp-high

9B DF mod100r/m disp-low disp-high

Without FWAIT:

11011111 mod100r/m disp-low disp-high

DF mod100r/m disp-low disp-high

Example

```
FBLD   PACKED_DECIMAL ;PACKED DECIMAL (BCD) LOAD
```

FBSTP (8087)
Packed Decimal (BCD) Store and Pop

Purpose

FBSTP converts the top element in the 8087 stack to a packed decimal integer, stores the result at the *destination* in memory, and pops the 8087 stack.

Format

FBSTP *destination*

Remarks

FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then setting the fractional part to zero, resulting in the nearest integer.

Note: If you are concerned about rounding, precede FBSTP with FRNDINT.

Logic

mem-op <- ST
pop 8087 stack

Exception Flags

I

Encoding

10011011 11011111 mod110r/m disp-low disp-high

9B DF mod110r/m disp-low disp-high

Without FWAIT:

11011111 mod110r/m disp-low disp-high

DF mod110r/m disp-low disp-high

Example

```
FBSTP  PACKED_DECIMAL
           ;PACKED DECIMAL (BCD) STORE AND POP
```

FCHS (8087)
Change Sign

Purpose

FCHS reverses the sign of the top element of the 8087 stack.

Format

FCHS

Remarks

FCHS complements the sign of the top element of the 8087 stack.

Logic

ST <- -ST

Exception Flags

I

Encoding

10011011 11011001 11100000

9B D9 E0

Without FWAIT:

11011001 11100000

D9 E0

Example

```
FCHS                ;CHANGE SIGN
```

FCLEX (8087)
Clear Exceptions

Purpose

FCLEX clears all exception flags, the interrupt request flag, and the busy flag in the status word.

Format

FCLEX

Remarks

The INT and BUSY lines of the 8087 become inactive.

FNCLEX is the alternate no-wait form for the FCLEX instruction.

Note: An exception handler must issue this instruction before returning to the interrupted computation, or the assembler produces another interrupt request immediately, and an endless loop can result.

Logic

Clear 8087 exceptions

Exception Flags

None

Encoding

10011011 11011011 11100010

9B DB E2

Example

```
FCLEX               ;CLEAR EXCEPTIONS
```

FCOM (8087)
Compare Real

Purpose

FCOM compares the top element of the 8087 stack to the *source* operand.

Format

FCOM
or
FCOM *source*

Remarks

The *source* operand can be a register on the 8087 stack, or a short or long real memory operand. If you do not code an operand, FCOM compares ST to ST(1).

Note: Positive and negative forms of zero compare identically as if they were unsigned.

Logic

Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST > *source* then C3 = 0 and C0 = 0
Else if ST < *source* then C3 = 0 and C0 = 1
Else if ST = *source* then C3 = 1 and C0 = 0
Else C3 = 1 and C0 = 1.

Note: FCOM cannot compare NaNs and ∞; they return C3 = 1 and C0 = 1, as shown above.

Exception Flags

I,D

8087 stack top with Memory short_real

Format

FCOM short_real

Remarks

FCOM compares ST to a short real memory operand.

Encoding

10011011 11011000 mod010r/m disp-low disp-high

9B D8 mod010r/m disp-low disp-high

Without FWAIT:

11011000 mod010r/m disp-low disp-high

D8 mod010r/m disp-low disp-high

Example

```
FCOM    SHORT_REAL
```

8087 stack top with Memory long_real

Format

FCOM long_real

Remarks

FCOM compares ST to a long real memory operand.

Encoding

10011011 11011100 mod010r/m disp-low disp-high

9B DC mod010r/m disp-low disp-high

Without FWAIT:

11011100 mod010r/m disp-low disp-high

DC mod010r/m disp-low disp-high

Example

```
FCOM   LONG_REAL
```

8087 stack top with Register on 8087 stack

Format

FCOM (no operand)

Remarks

FCOM compares ST to ST(1).

Encoding

10011011 11011000 11010001

9B D8 D1

Without FWAIT:

11011000 11010001

D8 D1

Example

```
FCOM                ;COMPARE REAL
```

Format

FCOM ST(*i*)

Remarks

FCOM compares ST to ST(*i*).

Encoding

10011011 11011000 11010(*i*)

9B D8 D0 + (*i*)

Without FWAIT:

11011000 11010(*i*)

D8 D0 + (*i*)

Example

```
FCOM   ST(7)        ;COMPARE REAL
```

FCOMP (8087)
Compare Real and Pop

Purpose

FCOMP compares the top element of the 8087 stack to the *source* operand and pops the 8087 stack.

Format

FCOMP
or
FCOMP *source*

Remarks

FCOMP operates like FCOM and also pops the 8087 stack. The *source* operand can be a register on the 8087 stack, or a short or long real memory operand. If you do not code an operand, FCOMP compares ST to ST(1).

Note: Positive and negative forms of zero compare the same as if they were unsigned.

Logic

Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST > *source* then C3 = 0 and C0 = 0
Else if ST < *source* then C3 = 0 and C0 = 1
Else if ST = *source* then C3 = 1 and C0 = 0
Else C3 = 1 and C0 = 1.

Note: FCOMP cannot compare NaNs and ∞; they return C3 = 1 and C0 = 1, as shown above.

Exception Flags

I,D

8087 stack top with Memory short_real

Format

FCOMP short_real

Remarks

FCOMP compares ST to a short real memory operand.

Encoding

10011011 11011000 mod011r/m disp-low disp-high

9B D8 mod011r/m disp-low disp-high

Without FWAIT:

11011000 mod011r/m disp-low disp-high

D8 mod011r/m disp-low disp-high

Example

```
FCOMP  SHORT_REAL ;COMPARE REAL AND POP
```

8087 stack top with Memory long_real

Format

FCOMP long_real

Remarks

FCOMP compares ST to a long real memory operand.

Encoding

10011011 11011100 mod011r/m disp-low disp-high

9B DC mod011r/m disp-low disp-high

Without FWAIT:

11011100 mod011r/m disp-low disp-high

DC mod011r/m disp-low disp-high

Example

```
FCOMP  LONG_REAL     ;COMPARE REAL AND POP
```

8087 stack top with Register on 8087 stack

Format

FCOMP (no operand)

Remarks

FCOMP compares ST to ST(1).

Encoding

10011011 11011000 11011001

9B D8 D9

Without FWAIT:

11011000 11011001

D8 D9

Example

```
FCOMP                ;COMPARE REAL AND POP
```

Format

FCOMP ST(*i*)

Remarks

FCOMP compares ST to ST(*i*).

Encoding

10011011 11011000 11011(*i*)

9B D8 D8 + (*i*)

Without FWAIT:

11011000 11011(*i*)

D8 D8 + (*i*)

Example

```
FCOMP ST(1)          ;COMP REAL AND POP
                     ;SAME AS FCOMP (no operands)
FCOMP ST(7)          ;COMP REAL AND POP
```

FCOMPP (8087)
Compare Real and Pop Twice

Purpose

FCOMPP compares the top element of the 8087 stack to ST(1) and pops the 8087 stack twice.

Format

FCOMPP

Remarks

FCOMPP operates like FCOM and also pops the 8087 stack twice, discarding both operands. FCOMPP compares ST to ST(1).

Note: Positive and negative forms of zero compare the same as if they were unsigned.

Logic

Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST > ST(1) then C3 = 0 and C0 = 0
Else if ST < ST(1) then C3 = 0 and C0 = 1
Else if ST = ST(1) then C3 = 1 and C0 = 0
Else C3 = 1 and C0 = 1.

Note: FCOMPP cannot compare NaNs and ∞; they return C3 = 1 and C0 = 1, as shown above.

Exception Flags

I,D

Encoding

10011011 11011110 11011001

9B DE D9

Without FWAIT:

11011110 11011001

DE D9

Example

```
FCOMPP              ;COMPARE REAL AND POP TWICE
```

FDECSTP (8087)
Decrease 8087 Stack Pointer

Purpose

FDECSTP subtracts 1 from the 8087 stack top pointer (TOP) in the status word.

Format

FDECSTP

Remarks

FDECSTP does not change tags or register contents, nor does it transfer data. FDECSTP is not the same as pushing the 8087 stack.

Note: Decreasing the 8087 stack pointer when TOP = 0 produces TOP = 7.

Logic

If TOP = 0 then
  TOP <- 7
Else TOP <- TOP-1

Exception Flags

None

Encoding

10011011 11011001 11110110

9B D9 F6

Without FWAIT:

11011001 11110110

D9 F6

Example

```
FDECSTP             ;DECREASE 8087 STACK POINTER
```

FDISI (8087)
Disable Interrupts

Purpose

FDISI sets the interrupt enable mask (IEM) in the control word and prevents the 8087 from issuing an interrupt request.

Format

FDISI

Remarks

FDISI disables interrupts.

Notes:

1. If the assembler decodes WAIT with pending exceptions, the 8087 produces an interrupt, masked or not.
2. FNDISI is the alternate no-wait form for the FDISI instruction. Refer to FNDISI, FENI, FNENI, and the 8087 control word for related information.

Logic

(IEM) = 1

Exception Flags

None

Encoding

10011011 11011011 11100001

9B DB E1

Example

```
FDISI              ;DISABLE INTERRUPTS
```

FDIV (8087)
Divide Real

Purpose

FDIV divides the *destination* by the *source* and returns the quotient to the *destination*.

Format

FDIV
or
FDIV *source*
or
FDIV *destination,source*

Remarks

You can write FDIV without operands, with only a *source*, or with a *destination* and a *source*.

Exception Flags

I,D,Z,O,U,P

FDIV (no operands) 8087 Stack Form

Format

FDIV [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded and are shown here for information only.

Remarks

FDIV picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FDIV (no operands) is similar to the FDIVP instruction with ST(*i*) being ST(1).

Logic

ST(1) <- ST(1) ÷ ST
pop 8087 stack

Encoding

10011011 11011110 11111001

9B DE F9

Without FWAIT:

11011110 11111001

DE F9

Example

```
FDIV                ;DIVIDE REAL
```

FDIV (source) Real Memory Form

Format

FDIV short_real
or
FDIV long_real

Remarks

With the real memory form, you can use a real number in memory directly as a *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this form of the instruction.

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST ÷ mem-op

Encoding

SHORT_REAL

10011011 11011000 mod110r/m disp-low disp-high
9B D8 mod110r/m disp-low disp-high

Without FWAIT:

11011000 mod110r/m disp-low disp-high

D8 mod110r/m disp-low disp-high

Example

```
FDIV   SHORT_REAL  ;DIVIDE REAL
```

Encoding

LONG_REAL

10011011 11011100 mod110r/m disp-low disp-high
9B DC mod110r/m disp-low disp-high

Without FWAIT:

11011100 mod110r/m disp-low disp-high

DC mod110r/m disp-low disp-high

Example

```
FDIV   LONG_REAL   ;DIVIDE REAL
```

FDIV (destination,source) Register Form

Remarks

Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format

FDIV ST,ST(*i*)

Logic

ST <- ST ÷ ST(*i*)

Encoding

10011011 11011000 11110(*i*)

9B D8 F0 + (*i*)

Without FWAIT:

11011000 11110(*i*)

D8 F0 + (*i*)

Example

```
FDIV   ST,ST(1)    ;DIVIDE REAL
FDIV   ST,ST(7)    ;DIVIDE REAL
```

Format

FDIV ST(*i*),ST

Logic

ST(*i*) <- ST(*i*) ÷ ST

Encoding

10011011 11011100 11111(*i*)

9B DC F8 + (*i*)

Without FWAIT:

11011100 11111(*i*)

DC F8 + (*i*)

Example

```
FDIV   ST(1),ST    ;DIVIDE REAL
FDIV   ST(7),ST    ;DIVIDE REAL
```

FDIVP (8087)
Divide Real and Pop

Purpose

FDIVP divides the *destination* by the *source*, returns the quotient to the *destination*, and pops the 8087 stack.

Format

FDIVP *destination,source*

Remarks

FDIVP picks the *source* operand from the ST register, or top of the 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to ST(*i*).

Note: FDIV (no operands) is similar to the FDIVP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST(*i*) ÷ ST
pop 8087 stack

Exception Flags

I,D,Z,O,U,P

Encoding

10011011 11011110 11111(*i*)

9B DE F8 + (*i*)

Without FWAIT:

11011110 11111(*i*)

DE F8 + (*i*)

Example

```
FDIVP  ST(1),ST    ;DIVIDE REAL AND POP
FDIVP  ST(7),ST    ;DIVIDE REAL AND POP
```

FDIVR (8087)
Divide Real Reversed

Purpose

FDIVR is a reversed division instruction. FDIVR divides the *source* operand by the *destination* operand and returns the quotient to the *destination*.

Format

FDIVR
or
FDIVR *source*
or
FDIVR *destination,source*

Remarks

You can write FDIVR without operands, with only a *source*, or with a *destination* and a *source*.

Exception Flags

I,D,Z,O,U,P

FDIVR (no operands) 8087 stack form

Format

FDIVR [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded and are shown here for information only.

Remarks

FDIVR picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FDIVR (no operands) is similar to the FDIVRP instruction with ST(*i*) being ST(1).

Logic

ST(1) <- ST ÷ ST(1)
pop 8087 stack

Encoding

10011011 11011110 11110001

9B DE F1

Without FWAIT:

11011110 11110001

DE F1

Example

```
FDIVR               ;DIVIDE REAL REVERSED
```

FDIVR (source) Real memory form

Format

FDIVR short_real
or
FDIVR long_real

Remarks

The real-memory form permits you to use a real number in memory directly as a *source* operand. The *destination* operand is the top of 8087 stack (register ST). It is implied in this form of the instruction.

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- mem-op ÷ ST

Encoding

SHORT_REAL

10011011 11011000 mod111r/m disp-low disp-high
9B D8 mod111r/m disp-low disp-high

Without FWAIT:

11011000 mod111r/m disp-low disp-high

D8 mod111r/m disp-low disp-high

Example

```
FDIVR  SHORT_REAL  ;DIVIDE REAL REVERSED
```

Encoding

LONG_REAL

10011011 11011100 mod111r/m disp-low disp-high
9B DC mod111r/m disp-low disp-high

Without FWAIT:

11011100 mod111r/m disp-low disp-high

DC mod111r/m disp-low disp-high

Example

```
FDIVR  LONG_REAL   ;DIVIDE REAL REVERSED
```

FDIVR (destination,source) Register form

Remarks

Specify the top element of the 8087 stack as one operand and any register on the 8087 stack as the other operand.

Format

FDIVR ST,ST(*i*)

Logic

ST <- ST(*i*) ÷ ST

Encoding

10011011 11011000 11111(*i*)

9B D8 F8 + (*i*)

Without FWAIT:

11011000 11111(*i*)

D8 F8 + (*i*)

Example

```
FDIVR  ST,ST(1)    ;DIVIDE REAL REVERSED
FDIVR  ST,ST(7)    ;DIVIDE REAL REVERSED
```

Format

FDIVR ST(*i*),ST

Logic

ST(*i*) <- ST ÷ ST(*i*)

Encoding

10011011 11011100 11110(*i*)

9B DC F0 + (*i*)

Without FWAIT:

11011100 11110(*i*)

DC F0 + (*i*)

Example

```
FDIVR  ST(1),ST    ;DIVIDE REAL REVERSED
FDIVR  ST(7),ST    ;DIVIDE REAL REVERSED
```

FDIVRP (8087)
Divide Real Reversed and Pop

Purpose

FDIVRP is a reversed division instruction. FDIVRP divides the *source* operand by the *destination* operand, returns the quotient to the *destination*, and pops the 8087 stack.

Format

FDIVRP *destination,source*

Remarks

FDIVRP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack.

Note: FDIVR (no operands) is similar to the FDIVRP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST ÷ ST(*i*)
pop 8087 stack

Exception Flags

I,D,Z,O,U,P

Encoding

10011011 11011110 11110(*i*)

9B DE F0 + (*i*)

Without FWAIT:

11011110 11110(*i*)

DE F0 + (*i*)

Example

```
FDIVRP ST(1),ST  ;DIVIDE REAL REVERSED AND POP
FDIVRP ST(7),ST  ;DIVIDE REAL REVERSED AND POP
```

FENI (8087)
Enable Interrupts

Purpose

FENI clears the interrupt enable mask (IEM) in the control word, allowing the processor to produce interrupt requests.

Format

FENI

Remarks

FENI allows the 8087 to produce interrupt requests.

Note: FNENI is the alternate no-wait form for the FENI instruction. See FNENI, FDISI, FNDISI, and 8087 control word for related information.

Logic

(IEM) = 0

Exception Flags

None

Encoding

10011011 11011011 11100000

9B DB E0

Example

```
FENI                ;ENABLE INTERRUPTS
```

FFREE (8087)
Free Register

Purpose

FFREE changes the tag of the *destination* register to empty.

Format

FFREE *destination*

Remarks

The content of the *destination* register is not affected.

Note: Refer to the 8087 tag word described in the IBM *Macro Assembler/2 Fundamentals* book for additional information.

Logic

TAG(*i*) <- 11 (empty)

Exception Flags

None

Encoding

10011011 11011101 11000(*i*)

9B DD C0 + (*i*)

Without FWAIT:

11011101 11000(*i*)

DD C0 + (*i*)

Example

```
FFREE  ST(1)       ;FREE STACK REGISTER ONE
FFREE  ST(7)       ;FREE STACK REGISTER SEVEN
```

FIADD (8087)
Integer Add

Purpose

FIADD adds the *source* and *destination* operands and returns the sum to the *destination*.

Format

FIADD *source*

Remarks

FIADD permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FIADD stores the sum of the two operands into the top of the 8087 stack (register ST).

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST + mem-op

Exception Flags

I,D,O,P

Encoding

SHORT_INTEGER

10011011 11011010 mod000r/m disp-low disp-high
9B DA mod000r/m disp-low disp-high

Without FWAIT:

11011010 mod000r/m disp-low disp-high

DA mod000r/m disp-low disp-high

Example

```
FIADD  SHORT_INTEGER ;INTEGER ADD
```

Encoding

WORD_INTEGER

10011011 11011110 mod000r/m disp-low disp-high
9B DE mod000r/m disp-low disp-high

Without FWAIT:

11011110 mod000r/m disp-low disp-high

DE mod000r/m disp-low disp-high

Example

```
FIADD  WORD_INTEGER ;INTEGER ADD
```

FICOM (8087)
Integer Compare

Purpose

FICOM compares the top element in the 8087 stack to the *source* operand.

Format

FICOM *source*

Remarks

FICOM permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. FICOM converts the *source* operand, which can refer to a word or short binary integer variable, to temporary real and compares the top of the 8087 stack to it.

Note: The assembler compares positive and negative forms of zero the same as if they were unsigned.

Logic

Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST > *source* then C3 = 0 and C0 = 0
Else if ST < *source* then C3 = 0 and C0 = 1
Else if ST = *source* then C3 = 1 and C0 = 0
Else C3 = 1 and C0 = 1.

Note: FICOM cannot compare NaNs and ∞; they return C3 = 1 and C0 = 1, as shown above.

Exception Flags

I,D

8087 stack top with Memory short integer

Format

FICOM short_integer

Remarks

FICOM compares ST to a short integer memory operand.

Encoding

10011011 11011010 mod010r/m disp-low disp-high

9B DA mod010r/m disp-low disp-high

Without FWAIT:

11011010 mod010r/m disp-low disp-high

DA mod010r/m disp-low disp-high

Example

```
FICOM  SHORT_INTEGER ;INTEGER COMPARE
```

8087 stack top with Memory word integer

Format

FICOM word_integer

Remarks

FICOM compares ST to a word integer memory operand.

Encoding

10011011 11011110 mod010r/m disp-low disp-high

9B DE mod010r/m disp-low disp-high

Without FWAIT:

11011110 mod010r/m disp-low disp-high

DE mod010r/m disp-low disp-high

Example

```
FICOM  WORD_INTEGER  ;INTEGER COMPARE
```

FICOMP (8087)
Integer Compare and Pop

Purpose

FICOMP compares the top element in the 8087 stack to the *source* operand and pops the 8087 stack.

Format

FICOMP *source*

Remarks

FICOMP operates in the same way as FICOM and also discards the value in ST by popping the 8087 stack. FICOMP permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. FICOMP converts the *source* operand, which can refer to a word or short binary integer variable, to temporary real and compares the top element in the 8087 stack to it.

Note: The assembler compares positive and negative forms of zero the same as if they were unsigned.

Logic

Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST > *source* then C3 = 0 and C0 = 0
Else if ST < *source* then C3 = 0 and C0 = 1
Else if ST = *source* then C3 = 1 and C0 = 0
Else C3 = 1 and C0 = 1.

Note: FICOMP cannot compare NaNs and ∞; they return C3 = 1 and C0 = 1 as shown above.

Exception Flags

I,D

8087 stack top with Memory short integer

Format

FICOMP short_integer

Remarks

FICOMP compares ST to a short integer memory operand.

Encoding

10011011 11011010 mod011r/m disp-low disp-high

9B DA mod011r/m disp-low disp-high

Without FWAIT:

11011010 mod011r/m disp-low disp-high

DA mod011r/m disp-low disp-high

Example

```
FICOMP SHORT_INTEGER ;INTEGER COMPARE
```

8087 stack top with Memory word integer

Format

FICOMP word_integer

Remarks

FICOMP compares ST to a word integer memory operand.

Encoding

10011011 11011110 mod011r/m disp-low disp-high

9B DE mod011r/m disp-low disp-high

Without FWAIT:

11011110 mod011r/m disp-low disp-high

DE mod011r/m disp-low disp-high

Example

```
FICOMP WORD_INTEGER  ;INTEGER COMPARE
```

FIDIV (8087)
Integer Divide

Purpose

FIDIV divides the *destination* by the *source* and returns the quotient to the *destination*.

Format

FIDIV *source*

Remarks

FIDIV permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FIDIV stores the quotient of the operands into the top of the 8087 stack (register ST).

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST ÷ mem-op

Exception Flags

I,D,Z,O,U,P

Encoding

SHORT_INTEGER

10011011 11011010 mod110r/m disp-low disp-high
9B DA mod110r/m disp-low disp-high

Without FWAIT:

11011010 mod110r/m disp-low disp-high

DA mod110r/m disp-low disp-high

Example

```
FIDIV  SHORT_INTEGER  ;INTEGER DIVIDE
```

Encoding

WORD_INTEGER

10011011 11011110 mod110r/m disp-low disp-high
9B DE mod110r/m disp-low disp-high

Without FWAIT:

11011110 mod110r/m disp-low disp-high

DE mod110r/m disp-low disp-high

Example

```
FIDIV  WORD_INTEGER  ;INTEGER DIVIDE
```

FIDIVR (8087)
Integer Divide Reversed

Purpose

FIDIVR, a reversed division instruction, divides the *source* operand by the *destination* operand and returns the quotient to the *destination*.

Format

FIDIVR *source*

Remarks

FIDIVR permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. You can use any memory-addressing mode to define the *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FIDIVR stores the quotient of the operands into the top of the 8087 stack (register ST).

Logic

ST <- mem-op ÷ ST

Exception Flags

I,D,Z,O,U,P

Encoding

SHORT_INTEGER

10011011 11011010 mod111r/m disp-low disp-high
9B DA mod111r/m disp-low disp-high

Without FWAIT:

11011010 mod111r/m disp-low disp-high

DA mod111r/m disp-low disp-high

Example

```
FIDIVR SHORT_INTEGER  ;INTEGER DIVIDE REVERSED
```

Encoding

WORD_INTEGER

10011011 11011110 mod111r/m disp-low disp-high
9B DE mod111r/m disp-low disp-high

Without FWAIT:

11011110 mod111r/m disp-low disp-high

DE mod111r/m disp-low disp-high

Example

```
FIDIVR WORD_INTEGER   ;INTEGER DIVIDE REVERSED
```

FILD (8087)
Integer Load

Purpose

FILD converts the *source* memory operand from its binary integer format to temporary real and loads (pushes) the result onto the 8087 stack.

Format

FILD *source*

Remarks

FILD tags the new top element in the 8087 stack zero if all bits in the *source* were zero and tags it valid otherwise. FILD permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be a short integer, a word integer, or a long integer. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction.

Note: You can use any storage-addressing mode to define the *source* operand.

Logic

push 8087 stack
ST <- mem-op

Exception Flags

I

Encoding

SHORT_INTEGER

10011011 11011011 mod000r/m disp-low disp-high
9B DB mod000r/m disp-low disp-high

Without FWAIT:

11011011 mod000r/m disp-low disp-high

DB mod000r/m disp-low disp-high

Example

```
FILD   SHORT_INTEGER   ;INTEGER LOAD
```

Encoding

WORD_INTEGER

10011011 11011111 mod000r/m disp-low disp-high
9B DF mod000r/m disp-low disp-high

Without FWAIT:

11011111 mod000r/m disp-low disp-high

DF mod000r/m disp-low disp-high

Example

```
FILD   WORD_INTEGER    ;INTEGER LOAD
```

Encoding

LONG_INTEGER

10011011 11011111 mod101r/m disp-low disp-high
9B DF mod101r/m disp-low disp-high

Without FWAIT:

11011111 mod101r/m disp-low disp-high

DF mod101r/m disp-low disp-high

Example

```
FILD   LONG_INTEGER    ;INTEGER LOAD
```

FIMUL (8087)
Integer Multiply

Purpose

FIMUL multiplies the *destination* by the *source* and returns the product to the *destination*.

Format

FIMUL *source*

Remarks

FIMUL permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FIMUL stores the product of the operands into the top of the 8087 stack (register ST).

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST * mem-op

Flags

I,D,O,P

Encoding

SHORT_INTEGER

10011011 11011010 mod001r/m disp-low disp-high
9B DA mod001r/m disp-low disp-high

Without FWAIT:

11011010 mod001r/m disp-low disp-high
DA mod001r/m disp-low disp-high

Example

```
FIMUL  SHORT_INTEGER  ;INTEGER MULTIPLY
```

Encoding

WORD_INTEGER

10011011 11011110 mod001r/m disp-low disp-high
9B DE mod001r/m disp-low disp-high

Without FWAIT:

11011110 mod001r/m disp-low disp-high
DE mod001r/m disp-low disp-high

Example

```
FIMUL  WORD_INTEGER   ;INTEGER MULTIPLY
```

FINCSTP (8087)
Increase 8087 Stack Pointer

Purpose

FINCSTP adds 1 to the 8087 stack top pointer (TOP) in the status word.

Format

FINCSTP

Remarks

FINCSTP does not change tags or register contents nor does it transfer data. Using FINCSTP is not the same as popping the 8087 stack, because FINCSTP does not set the tag of the previous 8087 stack top to empty.

Note: Increasing the 8087 stack pointer when TOP = 7 produces TOP = 0.

Logic

TOP <- TOP + 1

Flags

None

Encoding

10011011 11011001 11110111

9B D9 F7

Without FWAIT:

11011001 11110111
D9 F7

Example

```
FINCSTP             ;INCREASE STACK POINTER
```

FINIT/FNINIT (8087)
Initialize Processor

Purpose

FINIT does the functional equivalent of a hardware reset to the 8087, except that it does not affect the instruction-fetch synchronization of the 8087 and the 8088.

Format

FINIT (no operands)

Remarks

FINIT sets the control word to 03FFH, empties all floating point 8087 stack elements, and clears exception flags and busy interrupts. If you run FINIT while a previous 8087 memory-referencing instruction is running, the 8087 bus cycles in progress end abruptly.

Note: FNINIT is the alternate no-wait form for the FINIT instruction. See the sections on the 8087 Control Word and 8087 Initialization in the IBM *Macro Assembler/2 Fundamentals* book for additional information. See also the description of FNINIT in this chapter.

Logic

initialize 8087

Flags

None

Encoding

10011011 11011011 11100011

9B DB E3

Example

```
FINIT               ;INITIALIZE PROCESSOR
```

FIST (8087)
Integer Store

Purpose

FIST rounds the content of the 8087 stack top to an integer according to the RC field of the control word and transfers the result to the *destination*.

Format

FIST *destination*

Remarks

The *destination* can define a word or short integer variable.

Note: FIST stores negative zero in the same internal representation as positive zero: 0000...00.

Logic

mem-op <- ST

Flags

I,P

Encoding

SHORT_INTEGER

10011011 11011011 mod010r/m disp-low disp-high
9B DB mod010r/m disp-low disp-high

Without FWAIT:

11011011 mod010r/m disp-low disp-high
DB mod010r/m disp-low disp-high

Example

```
FIST   SHORT_INTEGER   ;INTEGER STORE
```

Encoding

WORD_INTEGER

10011011 11011111 mod010r/m disp-low disp-high
9B DF mod010r/m disp-low disp-high

Without FWAIT:

11011111 mod010r/m disp-low disp-high
DF mod010r/m disp-low disp-high

Example

```
FIST   WORD_INTEGER    ;INTEGER STORE
```

FISTP (8087)
Integer Store and Pop

Purpose

FISTP rounds the content of the 8087 stack top to an integer according to the RC field of the control word and transfers the result to the *destination*. FISTP pops the 8087 stack following the transfer.

Format

FISTP *destination*

Remarks

FISTP operates like FIST and also pops the 8087 stack following the transfer. The *destination* can define a word or short integer variable. The *destination* can be any of the binary integer data types.

Note: FISTP stores negative zero in the same internal representation as positive zero: 0000...00.

Logic

mem-op <- ST
pop 8087 stack

Flags

I,P

Encoding

SHORT_INTEGER

10011011 11011011 mod011r/m disp-low disp-high
9B DB mod011r/m disp-low disp-high

Without FWAIT:

11011011 mod011r/m disp-low disp-high
DB mod011r/m disp-low disp-high

Example

```
FISTP  SHORT_INTEGER    ;INTEGER STORE AND POP
```

Encoding

WORD_INTEGER

10011011 11011111 mod011r/m disp-low disp-high
  9B   DF   mod011r/m disp-low disp-high

Without FWAIT:

11011111 mod011r/m disp-low disp-high
  DF   mod011r/m disp-low disp-high

Example

```
FISTP  WORD_INTEGER     ;INTEGER STORE AND POP
```

Encoding

LONG_INTEGER

10011011 11011111 mod111r/m disp-low disp-high
  9B   DF   mod111r/m disp-low disp-high

Without FWAIT:

11011111 mod111r/m disp-low disp-high
  DF   mod111r/m disp-low disp-high

Example

```
FISTP  LONG_INTEGER     ;INTEGER STORE AND POP
```

FISUB (8087)
Integer Subtract

Purpose

FISUB subtracts the *source* operand from the *destination* and returns the difference to the *destination*.

Format

FISUB *source*

Remarks

FISUB permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. You can use any memory-addressing mode to define the *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FISUB stores the difference of the two operands in the top of the 8087 stack (register ST).

Logic

ST <- ST - mem-op

Flags

I,D,O,P

Encoding

SHORT_INTEGER

10011011 11011010 mod100r/m disp-low disp-high
9B DA mod100r/m disp-low disp-high

Without FWAIT:

11011010 mod100r/m disp-low disp-high
DA mod100r/m disp-low disp-high

Example

```
FISUB  SHORT_INTEGER  ;INTEGER SUBTRACT
```

Encoding

WORD_INTEGER

10011011 11011110 mod100r/m disp-low disp-high
9B DE mod100r/m disp-low disp-high

Without FWAIT:

11011110 mod100r/m disp-low disp-high
DE mod100r/m disp-low disp-high

Example

```
FISUB  WORD_INTEGER   ;INTEGER SUBTRACT
```

FISUBR (8087)
Integer Subtract Reversed

Purpose

FISUBR is a reversed subtraction instruction. FISUBR subtracts the *destination* from the *source* and returns the difference to the *destination*.

Format

FISUBR *source*

Remarks

FISUBR permits you to use a binary integer in memory directly as a *source* operand. The *source* operand can be either a short integer or a word integer. You can use any memory-addressing mode to define the *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this instruction. FISUBR stores the difference of the operands in the top of the 8087 stack (register ST).

Logic

ST <- mem-op - ST

Flags

I,D,O,P

Encoding

SHORT_INTEGER

10011011 11011010 mod101r/m disp-low disp-high
9B DA mod101r/m disp-low disp-high

Without FWAIT:

11011010 mod101r/m disp-low disp-high
DA mod101r/m disp-low disp-high

Example

```
FISUBR SHORT_INTEGER ;INTEGER SUBTRACT REVERSED
```

Encoding

WORD_INTEGER

10011011 11011110 mod101r/m disp-low disp-high
9B DE mod101r/m disp-low disp-high

Without FWAIT:

11011110 mod101r/m disp-low disp-high
DE mod101r/m disp-low disp-high

Example

```
FISUBR WORD_INTEGER  ;INTEGER SUBTRACT REVERSED
```

FLD (8087)
Load Real

Purpose

FLD loads (pushes) the *source* operand onto the top of the 8087 register stack.

Format

FLD *source*

Remarks

FLD loads (pushes) the *source* operand onto the top of the 8087 stack by decreasing the 8087 stack pointer by 1 and then copying the content of the *source* to the new 8087 stack top. The *source* can be a register on the 8087 stack ST(*i*) or any of the real data types in memory. FLD converts short and long real *source* operands to temporary reals.

Note: Coding FLD ST(0) copies the 8087 stack top.

Flags

I,D

Register operand to ST

Logic

T_1 <- ST(*i*)
push 8087 stack
ST <- T_1

Encoding

10011011 11011001 11000(*i*) (register)

9B D9 C0 + (*i*) (register)

Without FWAIT:

11011001 11000(*i*) (register)
D9 C0 + (*i*) (register)

Example

```
FLD ST(0) ;LOAD REAL (DUPES TOP OF STACK)
FLD ST(7) ;LOAD REAL STACK REGISTER SEVEN ST(7)
```

Memory operand to ST

Logic

push 8087 stack
ST <- mem-op

Encoding

SHORT_REAL

10011011 11011001 mod000r/m disp-low disp-high
9B D9 mod000r/m disp-low disp-high

Without FWAIT:

11011001 mod000r/m disp-low disp-high
D9 mod000r/m disp-low disp-high

Example

```
FLD     SHORT_REAL    ;LOAD REAL
```

Encoding

LONG_REAL

10011011 11011101 mod000r/m disp-low disp-high
9B DD mod000r/m disp-low disp-high

Without FWAIT:

11011101 mod000r/m disp-low disp-high
DD mod000r/m disp-low disp-high

Example

```
FLD     LONG_REAL     ;LOAD REAL
```

Encoding

TEMP_REAL

10011011 11011011 mod101r/m disp-low disp-high
9B DB mod101r/m disp-low disp-high

Without FWAIT:

11011011 mod101r/m disp-low disp-high
DB mod101r/m disp-low disp-high

Example

```
FLD     TEMP_REAL     ;LOAD REAL
```

FLD1 (8087)
Load +1.0

Purpose

FLD1 loads (pushes) +1.0 onto the 8087 stack.

Format

FLD1

Remarks

FLD1 is a constant instruction that loads (pushes) a value of +1.0 onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- 1.0

Flags

I

Encoding

10011011 11011001 11101000

9B D9 E8

Without FWAIT:

11011001 11101000
D9 E8

Example

```
FLD1                    ;LOAD +1.0
```

FLDCW (8087)
Load Control Word

Purpose

FLDCW replaces the current processor control word with the word defined by the *source* operand.

Format

FLDCW *source*

Remarks

Typically you use FLDCW to establish or change the mode of operation of the 8087. When you change modes, it is recommended to first clear any exceptions and then load the new control word.

Note: If you set an exception bit in the status word, loading a new control word that unmasks that exception and clears the interrupt enable mask produces an immediate interrupt request before the assembler executes the next instruction.

Logic

8087 control word <- mem-op

Flags

None

Encoding

10011011 11011001 mod101r/m disp-low disp-high

9B D9 mod101r/m disp-low disp-high

Without FWAIT:

11011001 mod101r/m disp-low disp-high
D9 mod101r/m disp-low disp-high

Example

```
FLDCW  TWO_BYTES   ;LOAD CONTROL WORD
```

FLDENV (8087)
Load Environment

Purpose

FLDENV reloads the 8087 environment from the memory area defined by the *source* operand.

Format

FLDENV *source*

Remarks

This data should have been written by a previous FSTENV/FNSTENV instruction. The 8088 instructions (that do not refer to the environment image) can immediately follow FLDENV, but do not execute an 8087 instruction that follows without an intervening FWAIT or assembler-produced WAIT.

Note: Loading an environment image that contains an unmasked exception causes an immediate interrupt request from the 8087 (assuming IEM = 0 in the environment image).

Logic

8087 environment <- mem-op

Flags

None

Encoding

10011011 11011001 mod100r/m disp-low disp-high

9B D9 mod100r/m disp-low disp-high

Without FWAIT:

11011001 mod100r/m disp-low disp-high
D9 mod100r/m disp-low disp-high

Example

```
FLDENV FOURTEEN_BYTES  ;LOAD ENVIRONMENT
```

FLDL2E (8087)
Load Log

Purpose

FLDL2E loads (pushes) the value $\log_2 e$ (log base 2 of e) onto the 8087 stack.

Format

FLDL2E (no operands)

Remarks

FLDL2E is a constant instruction that loads (pushes) a value of $\log_2 e$ onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- $\log_2 e$

Flags

I

Encoding

10011011 11011001 11101010

9B D9 EA

Without FWAIT:

11011001 11101010
D9 EA

Example

```
FLDL2E              ;LOAD LOG BASE 2 OF e
```

FLDL2T (8087)
Load Log

Purpose

FLDL2T loads (pushes) the value $\log_2 10$ (log base 2 of 10) onto the 8087 stack.

Format

FLDL2T

Remarks

FLDL2T is a constant instruction that loads (pushes) a value of $\log_2 10$ onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- $\log_2 10$

Flags

I

Encoding

10011011 11011001 11101001

9B D9 E9

Without FWAIT:

11011001 11101001
D9 E9

Example

```
FLDL2T              ;LOAD LOG BASE 2 OF 10
```

FLDLG2 (8087)
Load Log

Purpose

FLDLG2 loads (pushes) the value $\log_{10}2$ (log base 10 of 2) onto the top of the floating-point stack.

Format

FLDLG2

Remarks

FLDLG2 is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and runs faster than without the instruction. A constant instruction also simplifies programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- $\log_{10}2$

Flags

I

Encoding

10011011 11011001 11101100

9B D9 EC

Without FWAIT:

11011001 11101100
D9 EC

Example

```
FLDLG2              ;LOAD LOG BASE 10 OF 2
```

FLDLN2 (8087)
Load Log Base e of 2

Purpose

FLDLN2 loads (pushes) the value $\log_e 2$ (log base *e* of 2) onto the top of the floating-point stack.

Format

FLDLN2

Remarks

FLDLN2 is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and runs faster than without the instruction. A constant instruction simplifies programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- $\log_e 2$

Flags

I

Encoding

10011011 11011001 11101101

9B D9 ED

Without FWAIT:

11011001 11101101

D9 ED

Example

```
FLDLN2              ;LOAD LOG BASE e OF 2
```

FLDPI (8087)
Load PI

Purpose

FLDPI loads (pushes) the value π onto the top of the floating point stack.

Format

FLDPI

Remarks

FLDPI is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and runs faster than without the instruction. A constant instruction simplifies programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- π

Flags

I

Encoding

10011011 11011001 11101011

9B D9 EB

Without FWAIT:

11011001 11101011
D9 EB

Example

```
LDPI            ;LOAD PI
```

FLDZ (8087)
Load Zero

Purpose

FLDZ loads (pushes) the value +0.0 onto the top of the floating point stack.

Format

FLDZ

Remarks

FLDZ is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and runs faster than without the instruction. A constant instruction simplifies programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Note: Temporary real constants occupy 10 memory bytes and constant instructions are only 2 bytes long, saving memory and improving execution speed.

Logic

push 8087 stack
ST <- +0.0

Flags

I

Encoding

10011011 11011001 11101110

9B D9 EE

Without FWAIT:

11011001 11101110
D9 EE

Example

```
FLDZ            ;LOAD +0.0
```

FMUL (8087)
Multiply Real

Purpose

FMUL multiplies the *destination* operand by the *source* and returns the product to the *destination*.

Format

FMUL
or
FMUL *source*
or
FMUL *destination,source*

Remarks

FMUL stores the product of the two operands into the *destination* (left-most) operand. You can write FMUL without operands, with only a *source*, or with a *destination* and a *source*.

Flags

I,D,O,U,P

FMUL (no operands) 8087 stack form

Format

FMUL [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded and are shown here for information only.

Remarks

FMUL picks the *source* operand from the top of the 8087 stack and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new top element in the 8087 stack.

Note: FMUL (no operands) is similar to the FMULP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST(*i*) * ST
pop 8087 stack

Encoding

10011011 11011110 11001001

9B DE C9

Without FWAIT:

11011110 11001001
DE C9

Example

```
FMUL                ;MULTIPLY REAL
```

FMUL (source) Real memory form

Format

FMUL short_real
or
FMUL long_real

Remarks

The real memory form permits you to use a real number in memory directly as a *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this form of the instruction.

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST * mem-op

Encoding

SHORT_REAL

10011011 11011000 mod001r/m disp-low disp-high
9B D8 mod001r/m disp-low disp-high

Without FWAIT:

11011000 mod001r/m disp-low disp-high
D8 mod001r/m disp-low disp-high

Example

```
FMUL    SHORT_REAL
```

Encoding

LONG_REAL

10011011 11011100 mod001r/m disp-low disp-high
9B DC mod001r/m disp-low disp-high

Without FWAIT:

11011100 mod001r/m disp-low disp-high
  DC mod001r/m disp-low disp-high

Example

```
FMUL   LONG_REAL
```

FMUL (destination,source) Register form

Remarks

Specify the top of the 8087 stack as one operand and any register on the 8087 stack as the other operand.

Format

FMUL ST,ST(*i*)

Logic

ST <- ST * ST(*i*)

Encoding

10011011 11011000 11001(*i*)

  9B D8 C8 + (*i*)

Without FWAIT:

11011000 11001(*i*)
  D8 C8 + (*i*)

Example

```
FMUL   ST,ST(1)      ;MULTIPLY REAL
FMUL   ST,ST(0)      ;SQUARE TOP OF STACK
FMUL   ST,ST(7)
```

Format

FMUL ST(*i*),ST

Logic

ST(*i*) <- ST(*i*) * ST

Encoding

10011011 11011100 11001(*i*)

9B DC C8 + (*i*)

Without FWAIT:

11011100 11001(*i*)
DC C8 + (*i*)

Example

```
FMUL   ST(1),ST
FMUL   ST(7),ST
```

FMULP (8087)
Multiply Real and Pop

Purpose

FMULP multiplies the *destination* operand by the *source*, returns the product to the *destination*, and pops the floating-point stack.

Format

FMULP *destination,source*

Remarks

FMULP stores the product of the two operands into the *destination* (left-most) operand. FMULP picks the *source* operand from the ST register or top of the 8087 stack and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack.

Note: FMUL (no operands) is similar to the FMULP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST(*i*) * ST
pop 8087 stack

Flags

I,D,O,U,P

Encoding

10011011 11011110 11001(*i*)

9B DE C8 + (*i*)

Without FWAIT:

11011110 11001(*i*)
DE C8 + (*i*)

Example

```
FMULP  ST(7),ST      ;MULTIPLY REAL AND POP
```

FNCLEX (8087)
Clear Exceptions

Purpose

FNCLEX clears all exception flags, the interrupt request flag, and the busy flag in the status word.

Format

FNCLEX

Remarks

An exception handler must issue FNCLEX or FCLEX before the exception handler returns to the interrupted instruction. If not, the assembler produces a new exception immediately, causing an endless loop.

FNCLEX is the no-wait form of FCLEX, and you should use it only when there is danger of producing an infinite wait.

Note: See FCLEX and status word for related information.

Logic

Clear 8087 exceptions

Flags

None

Encoding

11011011 11100010

```
DB     E2
```

Example

```
FNCLEX          ;CLEAR EXCEPTIONS NO WAIT
```

FNDISI (8087)
Disable Interrupts

Purpose

FNDISI sets the interrupt enable mask in the control word and prevents the processor from issuing an interrupt request.

Format

FNDISI

Remarks

FNDISI is the no-wait form of FDISI; use it only when there is danger of producing an infinite wait.

Notes:

1. If the assembler decodes WAIT with pending exceptions, the 8087 produces an interrupt, masked or not.
2. See FDISI, FENI, FNENI, and the 8087 control word for related information.

Logic

(IEM) = 1

Flags

None

Encoding

11011011 11100001

```
  DB     E1
```

Example

```
FNDISI          ;DISABLE INTERRUPTS NO WAIT
```

FNENI (8087)
Enable Interrupts

Purpose

FNENI clears the interrupt enable mask in the control word, allowing the processor to produce interrupt requests.

Format

FNENI

Remarks

FNENI is the no-wait form of this instruction; use it only when there is danger of producing an infinite wait.

Note: See FENI, FDISI, FNDISI, and 8087 control word for related information.

Logic

(IEM) = 0

Flags

None

Encoding

11011011 11100000

```
DB      E0
```

Example

```
FNENI               ;ENABLE INTERRUPTS NO WAIT
```

FNINIT (8087)
Initialize Processor

Purpose

FNINIT does the functional equivalent of a hardware reset to the 8087, except that it does not affect the instruction–fetch synchronization of the 8087 and the 8088.

Format

FNINIT

Remarks

FNINIT sets the control word to 03FFH, empties all floating–point stack elements, and clears exception flags and busy interrupts. If FNINIT is run while a previous 8087 memory-referencing instruction is running, the 8087 bus cycles in progress end abruptly.

Note: FNINIT is the alternate no-wait form for the FINIT instruction. Use it only when there is danger of producing an infinite wait. See FINIT, 8087 control word, and 8087 initialization for additional information.

Logic

initialize 8087

Flags

None

Encoding

11011011 11100011

```
DB      E3
```

Example

```
FNINIT          ;INITIALIZE PROCESSOR NO WAIT
```

FNOP (8087)
No Operation

Purpose

FNOP causes no operation.

Format

FNOP

Remarks

FNOP stores the top of the 8087 stack to the top of the 8087 stack and thus, effectively, does no operation. You can use FNOP to replace a deleted instruction in an assembled program. This allows you to run the altered code without reassembling it.

Logic

ST <- ST

Flags

None

Encoding

10011011 11011001 11010000

9B D9 D0

Without FWAIT:

11011001 11010000
D9 D0

Example

```
FNOP                ;NO OPERATION
```

FNRSTOR (8087)
Restore State

Purpose

FNRSTOR, the restore state instruction, no-wait form, reloads the 8087 from the 94-byte memory area defined by *source* operand. A previous FSAVE/FNSAVE instruction should write this information.

Format

FNRSTOR *source*

Remarks

CPU (8088) instructions that do not refer to the save image can immediately follow the FNRSTOR instruction. However, an FWAIT instruction should precede any 8087 instruction following this instruction. The saved image requires 94 bytes of memory.

Note: The 8087 reacts to its new state at the finish of the FNRSTOR; it produces an immediate interrupt if the exception and mask bits in the memory image so indicate.

Logic

8087 state <- mem-op

Flags

None

Encoding

11011101 mod100r/m disp-low disp-high

DD mod100r/m disp-low disp-high

Example

```
FNRSTOR SAVE_STATE        ;RESTORE STATE
```

FNSAVE (8087)
Save State

Purpose

FNSAVE writes the full 8087 state (environment plus register stack) to the memory location specified in the *destination* operand and initializes the 8087.

Format

FNSAVE *destination*

Remarks

This is the no-wait form of FSAVE. Use it only when there is danger of producing an infinite wait.

If an instruction is running at the time the assembler decodes an FNSAVE instruction, the assembler allows the instruction to complete running before it performs FNSAVE. This means that the save reflects the state of the processor following the completion of any active instruction.

Note: See FSAVE, and 8087 save state for related information.

Logic

mem-op <- 8087 state

Flags

None

Encoding

11011101 mod110r/m disp-low disp-high

DD mod110r/m disp-low disp-high

Example

```
FNSAVE SAVE_STATE  ;SAVE CURRENT STATE OF 8087
```

FNSTCW (8087)
Store Control Word

Purpose

FNSTCW writes the current 8087 control word to the memory location defined by *destination*.

Format

FNSTCW *destination*

Remarks

FNSTCW is the no-wait form of FSTCW. Use this form only when there is danger of producing an infinite wait.

The control word stores all exception masks, the interrupt enable mask, and fields for precision control, rounding control, and infinity control.

Note: See FSTCW, and the 8087 control word for related information.

Logic

mem-op <- 8087 control word

Flags

None

Encoding

11011001 mod111r/m disp-low disp-high

D9 mod111r/m disp-low disp-high

Example

```
FNSTCW SAVE_CONTROL     ;STORE CONTROL WORD
```

FNSTENV (8087)
Store Environment

Purpose

FNSTENV writes the basic status (control, status, and tag words) of the 8087 and exception pointers to the memory location defined by the *destination* operand.

Format

FNSTENV *destination*

Remarks

This is the no-wait form of FSTENV. Use this form only if there is danger of producing an infinite wait.

Both FSTENV and FNSTENV set all exception masks in the 8087 after the environment is saved; it does not affect the interrupt enable mask. The assembler must allow FNSTENV to finish before it decodes any other instruction. An explicit FWAIT should precede the next 8087 instruction. There is no risk of the following FWAIT causing an infinite wait, because FNSTENV masks all exceptions, thus preventing an interrupt request from the 8087. If the assembler decodes FNSTENV while another instruction is running concurrently in the NEU (8087 numeric execution unit), the 8087 does not store the environment until the other instruction has completed. The data saved by this instruction reflects the state of the 8087 after the assembler executes any previously decoded instruction.

Note: See FSTENV and 8087 environment for related information.

Logic

```
mem-op <- 8087 environment
set all exception masks
```

Flags

None

Encoding

11011001 mod110r/m disp-low disp-high

D9 mod110r/m disp-low disp-high

Example

```
FNSTENV STORE_ENVIRONMENT  ;STORE ENVIRONMENT
```

FNSTSW (8087)
Store Status Word

Purpose

FNSTSW writes the current value of the 8087 status word to the memory location defined by the *destination* operand.

Format

FNSTSW *destination*

Remarks

This is the no-wait form of this instruction. Use this form only when there is danger of producing an infinite wait.

Use FNSTSW or its companion instruction FSTSW to poll the status of the processor. Then you can use this status information to do conditional branching. You can use FNSTSW (no-wait) to tell if the 8087 is busy.

Note: See FSTSW and the 8087 status word for related information.

Logic

mem-op <- 8087 status word

Flags

None

Encoding

11011101 mod111r/m disp-low disp-high
DD mod111r/m disp-low disp-high

Example

This is an example showing a method to poll the status of the 8087 looking for a not-busy condition.

```
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:CSEG

STATUS  DW      0       ;8087 STATUS
STAT    RECORD  STBUSY:1,STC3:1,STTOP:3,STC2:1,
STC1:1,STC0:1,STINT:1,STRES:1,STPFLG:1,STUFLG:1
,STOFLG:1,STZFLG:1,STDFLG:1,STIFLG:1
;The previous instruction
; (STAT   RECORD...STIFLG:1)
; must be contained on one line of code.
;LOOP HERE UNTIL THE 8087 IS NOT BUSY
POLL:   FNSTSW  STATUS  ;STORE STATUS WORD
        TEST    STATUS,MASK STBUSY
        JNZ     POLL

CSEG    ENDS
        END
```

FNSTSW AX (80287)
Store Status Word

Purpose

FNSTSW AX writes the current value of the 80287 status word directly to the AX register.

Format

FNSTSW AX

Remarks

This is the no-wait form of this instruction. As a special 80287 instruction, FNSTSW AX writes the current value of the 80287 status word directly into the 80286 AX register. This instruction optimizes conditional branching in numeric programs, where the 80286 processor must test the condition of various NPX status bits. This instruction does not check for unmasked numeric exceptions. When the assembler executes this instruction, the 80286 AX register is updated with the NPX status word before the processor executes any further instructions. In this way, the 80286 can immediately test the NPX status word without requiring any WAIT or other synchronization instructions.

You must use the .287 pseudo-op to use this instruction.

Note: See FSTSW AX for related information.

Logic

AX <- 80287 status word

Flags

None

Encoding

1101111 11100000

DF E0

Example

```
FNSTSW AX
```

FPATAN (8087)
Partial Arc Tangent

Purpose

FPATAN computes the θ = ARCTAN(Y/X) function. FPATAN takes X from the top element in the 8087 stack and Y from the next 8087 stack element. The instruction pops the 8087 stack and returns θ to the new 8087 stack top, overwriting the Y operand.

Format

FPATAN

Remarks

The FPATAN instruction assumes that the operands are valid and in-range. The valid range for X and Y is:

$0 < Y < X < +\infty$

To be considered valid, an operand must be normalized. If an operand is either incorrect or out of range, the instruction produces an undefined result without signaling an exception.

Logic

T_1 <- ARCTAN (ST(1)/ST)
pop 8087 stack
ST <- T_1

Flags

U,P (operands not checked)

Encoding

10011011 11011001 11110011

9B D9 F3

Without FWAIT:

11011001 11110011
D9 F3

Example

```
FPATAN          ;PARTIAL ARC TANGENT
```

FPREM (8087)
Partial Remainder

Purpose

FPREM modulo divides the 8087 stack top element ST by the next 8087 stack element ST(1). The sign of the remainder is the same as the sign of the original dividend.

Format

FPREM

Remarks

FPREM operates by doing successive subtractions. It can reduce a magnitude difference of up to 2^{64} in one run. If FPREM produces a remainder that is less than the modulus ST(1), the function is complete and FPREM clears bit C2 of the status word condition code. If the function is incomplete, FPREM sets C2 to 1. The result in ST is then called the partial remainder.

You can use software to inspect C2 by storing the status word following the running of FPREM and running the instruction again (using the partial remainder in ST as the dividend) until C2 is cleared. An alternate possibility is comparing ST with ST(1) to determine when the function is complete. If ST > ST(1), you must run FPREM again. If ST = ST(1), the remainder is 0 and the run is complete. If ST < ST(1), the remainder is ST and the run is complete. FPREM produces an exact result. The precision exception does *not* occur.

An important use for FPREM is to reduce arguments (operands) of 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 reduces an argument so that it is in range of FPTAN. Because FPREM produces an exact result, the argument reduction does *not* introduce round-off error into the calculation-even if several iterations are required to bring the argument into range.

FPREM also provides the least-significant 3 bits of the quotient generated by FPREM (in C3,C1,C0). This is also important for transcendental argument reduction because it locates the original angle in the correct one of 8 $\pi/4$ segments of the unit circle.

Logic

ST <- repeat (ST-ST(1))
If ST > ST(1) then C2 = 1, PREM = ST
Else if ST = ST(1) then C2 = 0, REM = 0
Else C2 = 0, REM = ST

Flags

I,D,U

Encoding

10011011 11011001 11111000

9B D9 F8

Without FWAIT:

11011001 11111000
D9 F8

Example

```
FPREM           ;PARTIAL REMAINDER
```

FPTAN (8087)
Partial Tangent

Purpose

FPTAN computes the Y/X = TAN (θ) function. FPTAN takes the value θ from the top element of the 8087 stack (ST). The result of the operation is a ratio; Y replaces θ in the 8087 stack and X is pushed, becoming the new 8087 stack top.

Format

FPTAN

Remarks

The FPTAN Instruction assumes that the operand is valid and in-range. The value of θ must be in the range $0 \leq \theta < \pi/4$. To be considered valid, an operand must be normalized. If the operand is incorrect or out-of-range, the instruction produces an undefined result without signaling an exception.

The ratio result of FPTAN and the ratio argument of FPATAN optimize the calculation of the other trigonometric functions including SIN, COS, ARCSIN and ARCTAN. You can get these functions from TAN and ARCTAN using standard trigonometric identities.

Logic

Y/X <- TAN(ST)
ST <- Y
push 8087 stack
ST <- X

Flags

I,P (operands not checked)

Encoding

10011011 11011001 11110010

9B D9 F2

Without FWAIT:

11011001 11110010
D9 F2

Example

```
FPTAN           ;PARTIAL TANGENT
```

FRNDINT (8087)
Round to Integer

Purpose

FRNDINT rounds the 8087 stack top element ST to an integer.

Format

FRNDINT

Remarks

The setting of the RC field of the control word determines the rules for rounding. There are four modes of rounding available:

RC = 00 Round to nearest integer; if exactly midway, FRNDINT chooses the even integer. (This is the default mode.)
RC = 01 Round downward (toward - ∞)
RC = 10 Round upward (toward + ∞)
RC = 11 Chop (round toward 0)

Logic

ST <- nearest integer (ST)

Exception Flags

I,P

Encoding

10011011 11011001 11111100
9B D9 FC

Without FWAIT:

11011001 11111100
D9 FC

Example

```
FRNDINT         ;ROUND TO INTEGER
```

FRSTOR (8087)
Restore State

Purpose

FRSTOR, the restore state instruction, reloads the 8087 from the 94-byte memory area defined by *source* operand. A previous FSAVE/FNSAVE instruction should write this information.

Format

FRSTOR *source*

Remarks

CPU (8088) instructions that do not refer to the save image can immediately follow the FRSTOR instruction. However, an FWAIT instruction should precede any 8087 instruction following this instruction. The saved image requires 94 bytes of memory.

Note: The 8087 reacts to its new state at the finish of the FRSTOR; it produces an immediate interrupt if the exception and mask bits in the memory image so indicate.

Logic

8087 state <- mem-op

Exception Flags

None

Encoding

10011011 11011101 mod100r/m disp-low disp-high

9B DD mod100r/m disp-low disp-high

Example

```
FRSTOR SAVE_STATE       ;RESTORE STATE
```

FSAVE (8087)
Save State

Purpose

FSAVE writes the full 8087 state (environment plus register stack) to the memory location specified in the *destination* operand, and initializes the 8087.

Format

FSAVE *destination*

Remarks

After FSAVE writes the image to memory, it initializes the 8087 as if you had run FINIT/FNINIT.

Note: See FNSAVE, FINIT, and 8087 save state for related information.

Logic

mem-op <- 8087 state

Exception Flags

None

Encoding

10011011 11011101 mod110r/m disp-low disp-high

9B DD mod110r/m disp-low disp-high

Example

```
FSAVE SAVE_STATE ;SAVE CURRENT STATE OF 8087
```

FSCALE (8087)
Scale

Purpose

FSCALE interprets the value contained in ST(1) as an integer and adds this value to the exponent of the number in ST.

Format

FSCALE

Remarks

FSCALE rapidly multiplies or divides by integral powers of 2. It is especially useful for scaling the elements of a vector.

The value contained in ST(1) must be an integer in the range $-2^{15} \leq ST(1) < 2^{15}$. If the value in ST(1) is out of range or $0 < ST(1) < 1$, the instruction produces an undefined result and does *not* signal an exception. Loading the scale factor from a word integer ensures correct operation.

Note: If the value is not an integer, but is in range and greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude.

Logic

$ST \leftarrow ST * 2^{ST(1)}$

Exception Flags

I,O,U

Encoding

10011011 11011001 11111101

9B D9 FD

Without FWAIT:

11011001 11111101
D9 FD

Example

```
FSCALE          ;SCALE
```

FSETPM (80287)
Set Protected Mode

Purpose

FSETPM sets the operating mode of the 80287 to Protected Virtual-Address mode.

Format

FSETPM

Remarks

FSETPM is meant to be executed in the power-up initialization routine of the 80286, when the 80286 is put into Protected Mode. Once FSETPM is run, the 80287 remains in Protected Mode until the next hardware reset — even after running FINIT, FSAVE, or FRSTOR.

You must use the .287 pseudo-op to enable this instruction.

Logic

Set Protected Mode

Flags

None

Encoding

```
11011011 11100100 00000000 00000000
  DB      E4      0       0
```

Example

```
FSETPM
```

FSQRT (8087)
Square Root

Purpose

FSQRT replaces the content of the 8087 stack top element ST with its square root.

Format

FSQRT

Remarks

FSQRT runs rapidly and is comparable to normal division. This allows an application that would normally run slowly because of the presence of square root calculations to achieve normal speed. ST must be in the range:

$-0 \leq ST \leq +\infty$

Note: The square root of -0 is defined to be -0.

Logic

ST <- $\sqrt{ST}$

Exception Flags

I,D,P

Encoding

10011011 11011001 11111010

9B D9 FA

Without FWAIT:

11011001 11111010
D9 FA

Example

```
FSQRT           ;SQUARE ROOT
```

FST (8087)
Store Real

Purpose

FST transfers the 8087 stack top ST to the location defined by the *destination* operand.

Format

FST *destination*

Remarks

FST transfers the top of the 8087 stack to the *destination*, which can be another 8087 stack element or a short or long real memory operand. If the contents of the 8087 stack top are longer than the *destination*, FST rounds the contents of the 8087 stack top according to the RC (rounding control) field in the control word. If the 8087 stack top is tagged a special value (it contains ∞, a NaN, or a denormal), FST does not round the 8087 stack top significand. In this case, the assembler deletes the least-significant bits of the 8087 stack top to fit the *destination*. It treats the exponent in the same way. This preserves the identification of the value as a special value so that you can properly load and tag it later in the program, if you want.

Exception Flags

I,O,U,P

FST ST to Register operand

Format

FST ST(*i*)

Remarks

Transfer ST to 8087 stack element ST(*i*).

Logic

ST(*i*) <- ST

Encoding

10011011 11011101 11010(*i*)

9B DD D0 + (*i*)

Without FWAIT:

11011101 11010(*i*)
DD D0 + (*i*)

Example

```
FST   ST(5)     ;STORE REAL
```

FST ST to Memory operand

Format

FST short_real
or
FST long_real

Logic

mem-op <- ST

Encoding

SHORT_REAL

10011011 11011001 mod010r/m disp-low disp-high
9B D9 mod010r/m disp-low disp-high

Without FWAIT:

11011001 mod010r/m disp-low disp-high
D9 mod010r/m disp-low disp-high

Example

```
FST   SHORT_REAL        ;STORE REAL
```

Encoding

LONG_REAL

10011011 11011101 mod010r/m disp-low disp-high
9B DD mod010r/m disp-low disp-high

Without FWAIT:

11011101 mod010r/m disp-low disp-high
DD mod010r/m disp-low disp-high

Example

```
FST   LONG_REAL         ;STORE REAL
```

FSTCW (8087)
Store Control Word

Purpose

FSTCW writes the current 8087 control word to the memory location defined by the *destination.*

Format

FSTCW *destination*

Remarks

An assembler-produced WAIT instruction precedes the FSTCW form of this instruction. This is the normal form of this instruction. FNSTCW does not cause the assembler to produce a WAIT.

Note: See FNSTCW and the 8087 control word for related information.

Logic

mem-op <- 8087 control word

Exception Flags

None

Encoding

10011011 11011001 mod111r/m disp-low disp-high

9B D9 mod111r/m disp-low disp-high

Example

```
FSTCW CONTROL_CONTROL_WORD   ;STORE CONTROL WORD
```

FSTENV (8087)
Store Environment

Purpose

FSTENV writes the status (control, status and tag words of the 8087, and exception pointers) to the memory location defined by the *destination* operand.

Format

FSTENV *destination*

Remarks

An assembler-generated WAIT instruction precedes the FSTENV form of this instruction. This ensures that any instruction currently active completes its execution before FSTENV stores the environment, and that the environment stored is current. After storing, FSTENV sets all exception masks in the processor; it does not affect the interrupt-enable mask. Use FNSTENV when you do not want an 8088 WAIT.

You must allow FSTENV/FNSTENV to finish before you decode any other 8087 instruction. When FSTENV is coded, an assembler-produced WAIT should precede the next 8087 instruction.

Note: See FNSTENV and the 8087 environment for related information.

Logic

mem-op <- 8087 environment

Exception Flags

None

Encoding

10011011 11011001 mod110r/m disp-low disp-high

9B D9 mod110r/m disp-low disp-high

Example

```
FSTENV STORE_ENVIRONMENT   ;STORE ENVIRONMENT
```

FSTP (8087)
Store Real and POP

Purpose

FSTP transfers the 8087 stack top element ST to the location defined by the *destination* operand. FSTP then pops the top element of the 8087 stack from the 8087 stack.

Format

FSTP *destination*

Remarks

The *destination* can be another 8087 stack element or memory operand (short-real, long-real, or temporary-real). If the *destination* is short or long real memory, FSTP rounds the significand to the width of the *destination* according to the RC field of the control word and converts the exponent to the width and bias of the *destination* format.

FSTP permits storing temporary real numbers in a memory operand while FST does not. FSTP ST(0) is the same as popping the 8087 stack with no data transfer.

Note: See FST for related information.

Exception Flags

I,O,U,P

FSTP ST to Register operand

Format

FSTP ST(*i*)

Remarks

This moves ST to the 8087 stack element ST(*i*) and pops the 8087 stack.

Logic

ST(*i*) <- ST
pop 8087 stack

Encoding

10011011 11011101 11011(*i*)

9B DD D8 + (*i*)

Without FWAIT:

11011101 11011(*i*)
DD D8 + (*i*)

Example

```
FSTP   ST(5)    ;STORE REAL AND POP
```

FSTP ST to Memory operand

Format

FSTP short_real
 or
FSTP long_real
 or
FSTP temp_real

Remarks

Transfer ST to memory operand and pop the 8087 stack.

Encoding

SHORT_REAL

10011011 11011001 mod011r/m disp-low disp-high
 9B D9 mod011r/m disp-low disp-high

Without FWAIT:

11011001 mod011r/m disp-low disp-high
 D9 mod011r/m disp-low disp-high

Example

```
FSTP   SHORT_REAL      ;STORE REAL AND POP
```

Encoding

LONG_REAL

10011011 11011101 mod011r/m disp-low disp-high
 9B DD mod011r/m disp-low disp-high

Without FWAIT:

11011101 mod011r/m disp-low disp-high
 DD mod011r/m disp-low disp-high

Example

```
FSTP   LONG_REAL       ;STORE REAL AND POP
```

Encoding

TEMPORARY_REAL

10011011 11011011 mod111r/m disp-low disp-high
9B DB mod111r/m disp-low disp-high

Without FWAIT:

11011011 mod111r/m disp-low disp-high
DB mod111r/m disp-low disp-high

Example

```
FSTP   TEMPORARY_REAL  ;STORE REAL AND POP
```

FSTSW (8087)
Store Status Word

Purpose

FSTSW writes the current value of the 8087 status word to the memory location defined by the *destination* operand.

Format

FSTSW *destination*

Remarks

An assembler-generated WAIT instruction precedes the FSTSW form of this instruction. In special cases where you do not want this WAIT, use the FNSTSW instruction.

This is a useful instruction to trigger various conditional branching routines by determining the state of the fields in the status word.

If you use the WAIT form (FSTSW) with an outstanding unmasked exception, a deadlock results.

Note: See FNSTSW and the 8087 status word for related information.

Logic

mem-op <- 8087 status word

Exception Flags

None

Encoding

10011011 11011101 mod111r/m disp-low disp-high
9B DD mod111r/m disp-low disp-high

Example

```
FSTSW STORE_STATUS_WORD   ;STORE STATUS WORD
```

FSTSW AX (80287)
Store Status Word

Purpose

FSTSW AX writes the current value of the 80287 status word directly to the AX register.

Format

FSTSW AX

Remarks

As a special 80287 instruction, FSTSW AX writes the current value of the 80287 status word directly into the 80286 register. This instruction optimizes conditional branching in numeric programs, where the 80286 processor must test the condition of various NPX status bits. This instruction checks for unmasked numeric exceptions. When the assembler executes this instruction, the 80286 AX register is updated with the NPX status word before the processor executes any further instructions. In this way, the 80286 can immediately test the NPX status word without requiring any WAIT or other synchronization instructions.

An assembler-generated WAIT instruction precedes the FSTSW form of this instruction. In special cases where you do not want this WAIT, use the FNSTSW instruction.

This is a useful instruction to start various conditional branching routines by determining the state of the status bits in the status word.

If you use the WAIT form (FSTSW AX) with an outstanding unmasked exception, a deadlock results.

You must use the .287 pseudo-op to use this instruction.

Note: See FNSTSW AX for related information.

Logic

AX <- 80287 status word

Exception Flags

None

Encoding

10011011 11011111 11100000

9B DF E0

Example

```
FSTSW AX
```

FSUB (8087)
Subtract Real

Purpose

FSUB subtracts the *source* operand from the *destination* operand and returns the difference to the *destination*.

Format

FSUB
or
FSUB *source*
or
FSUB *destination,source*

Remarks

FSUB stores the difference of the two operands in the *destination* (left-most) operand. You can write FSUB without operands, with only a *source*, or with a *destination* and a *source*.

Note: See FSUBP and FISUB for related information.

Exception Flags

I,D,O,U,P

FSUB (no operands) 8087 stack form

Format

FSUB [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks

FSUB picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FSUB (no operands) is similar to the FSUBP instruction with ST(*i*) being ST(1).

Logic

ST(1) <- ST(1) - ST
pop 8087 stack

Encoding

10011011 11011110 11101001

9B DE E9

Without FWAIT:

11011110 11101001
DE E9

Example

```
FSUB                ;SUBTRACT REAL
```

FSUB (source) Real memory form

Format

FSUB short_real
  or
FSUB long_real

Remarks

The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this form of the instruction.

Note: You can use any memory-addressing mode to define the *source* operand.

Logic

ST <- ST - mem-op

Encoding

SHORT_REAL

10011011 11011000 mod100r/m disp-low disp-high
  9B D8 mod100r/m disp-low disp-high

Without FWAIT:

11011000 mod100r/m disp-low disp-high
  D8 mod100r/m disp-low disp-high

Example

```
FSUB   SHORT_REAL      ;SUBTRACT REAL
```

Encoding

LONG_REAL

10011011 11011100 mod100r/m disp-low disp-high
9B DC mod100r/m disp-low disp-high

Without FWAIT:

11011100 mod100r/m disp-low disp-high
DC mod100r/m disp-low disp-high

Example

```
FSUB   LONG_REAL        ;SUBTRACT REAL
```

FSUB (destination,source) Register form

Remarks

Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format

FSUB ST,ST(*i*)

Logic

ST <- ST - ST(*i*)

Encoding

10011011 11011000 11100(*i*)

9B D8 E0 + (*i*)

Without FWAIT:

11011000 11100(*i*)
D8 E0 + (*i*)

Example

```
FSUB    ST,ST(1)        ;SUBTRACT REAL
FSUB    ST,ST(7)
```

Format

FSUB ST(*i*),ST

Logic

ST(*i*) <- ST(*i*) - ST

Encoding

10011011 11011100 11101(*i*)

9B DC E8 + (*i*)

Without FWAIT:

11011100 11101(*i*)
DC E8 + (*i*)

Example

```
FSUB    ST(1),ST        ;SUBTRACT REAL
FSUB    ST(7),ST
```

FSUBP (8087)
Subtract Real and POP

Purpose

FSUBP subtracts the *source* operand from the *destination* operand and returns the result to the *destination* operand. Then FSUBP pops the floating-point 8087 stack.

Format

FSUBP *destination,source*

Remarks

FSUBP stores the difference of the two operands in the *destination* (leftmost) operand. FSUBP picks the *source* operand from the ST register, or top of the 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. You can use FSUBP to get the 8087 stack top as the *source* operand and then discard it by popping the 8087 stack.

See FSUB and FISUB for related information.

Note: FSUB (no operands) is similar to the FSUBP instruction with ST(*i*) being ST(1).

Logic

ST(*i*) <- ST(*i*) - ST
pop 8087 stack

Exception Flags

I,D,O,U,P

Encoding

10011011 11011110 11101(*i*)

9B DE E8 + (*i*)

Without FWAIT:

11011110 11101(*i*)
DE E8 + (*i*)

Example

```
FSUBP  ST(7),ST      ;SUBTRACT REAL AND POP
```

FSUBR (8087)
Subtract Real Reversed

Purpose

FSUBR subtracts the *destination* operand from the *source* operand and returns the result to the *destination*.

Format

FSUBR
or
FSUBR *source*
or
FSUBR *destination,source*

Remarks

FSUBR reverses the normal order of operation. All other properties are the same as the FSUB instruction.

FSUBR stores the difference of the two operands in the *destination* (leftmost) operand. You can write FSUBR without operands, with only a *source*, or with a *destination* and a *source*.

Note: See FSUB, FSUBP, FSUBRP and FISUB for related information.

Exception Flags

I,D,O,U,P

FSUBR (no operands) 8087 stack form

Format

FSUBR [ST(1),ST]

Note: ST(1) and ST are the implied operands; they are not coded, and are shown here for information only.

Remarks

FSUBR picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FSUBR (no operands) is similar to the FSUBRP instruction with ST(*i*) being ST(1).

Logic

ST(1) <- ST - ST(1)
pop 8087 stack

Encoding

10011011 11011110 11100001

9B DE E1

Without FWAIT:

11011110 11100001
DE E1

Example

```
FSUBR                   ;SUBTRACT REAL
```

FSUBR (source) Real-memory form

Format

FSUBR short_real
or
FSUBR long_real

Remarks

The real-memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of the 8087 stack (register ST). It is implied in this form of the instruction.

Note: Any memory-addressing mode can be used to define the *source* operand.

Logic

ST <- mem-op - ST

Encoding

SHORT_REAL

10011011 11011000 mod101r/m disp-low disp-high
9B D8 mod101r/m disp-low disp-high

Without FWAIT:

11011000 mod101r/m disp-low disp-high
D8 mod101r/m disp-low disp-high

Example

```
FSUBR   SHORT_REAL
```

Encoding

LONG_REAL

10011011 11011100 mod101r/m disp-low disp-high
9B DC mod101r/m disp-low disp-high

Without FWAIT:

11011100 mod101r/m disp-low disp-high
  DC mod101r/m disp-low disp-high

Example

```
FSUB   LONG_REAL
```

FSUBR (destination,source) Register form

Remarks

Specify the top element of the 8087 stack as one operand and any register on the 8087 stack as the other operand.

Format

FSUBR ST,ST(*i*)

Logic

ST <- ST(*i*) - ST

Encoding

10011011 11011000 11101(*i*)

  9B D8 E8 + (*i*)

Without FWAIT:

11011000 11101(*i*)
  D8 E8 + (*i*)

Example

```
FSUBR   ST,ST(1)      ;SUBTRACT REAL
FSUBR   ST,ST(7)
```

Format

FSUBR ST(*i*),ST

Logic

ST(*i*) <- ST - ST(*i*)

Encoding

10011011 11011100 11100(*i*)

9B DC E0 + (*i*)

Without FWAIT:

11011100 11100(*i*)
DC E0 + (*i*)

Example

```
FSUBR   ST(1),ST
FSUBR   ST(7),ST
```

FSUBRP (8087)
Subtract Real Reversed and POP

Purpose

FSUBRP subtracts the *destination* operand from the *source* operand and returns the result to the *destination*. FSUBRP then pops the top element of the 8087 stack.

Format

FSUBRP *destination,source*

Remarks

FSUBRP reverses the normal order of operation. All other properties are the same as the FSUBP instruction.

FSUBRP stores the difference of the two operands into the *destination* (leftmost) operand. FSUBRP picks the *source* operand from the ST register or top of the 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. You can use FSUBRP to get the top element of the 8087 stack as the *source* operand, and then discard it by popping the 8087 stack.

See FSUB, FSUBR, FSUBP, and FISUBR for related information.

Logic

ST(*i*) <- ST - ST(*i*)
pop 8087 stack

Exception Flags

I,D,O,U,P

Encoding

10011011 11011110 11100(*i*)

9B DE E0 + (*i*)

Without FWAIT:

11011110 11100(*i*)
DE E0 + (*i*)

Example

```
FSUBRP  ST(7),ST      ;SUBTRACT REAL AND POP
```

FTST (8087)
Test

Purpose

FTST tests the floating-point 8087 stack top element by comparing it to zero. FTST posts the result to the condition code of the status word.

Format

FTST

Remarks

FTST posts the results to the condition code as follows:

C3	C0	Result
0	0	ST > 0
0	1	ST < 0
1	0	ST = 0
1	1	ST is not comparable (for example, it is a NaN or projective ∞)

Note: See FCOM, FCOMP, FCOMPP, FICOM, FICOMP, and FXAM for other comparison instructions.

Logic

ST <- ST - 0

Exception Flags

I,D

Encoding

10011011 11011001 11100100

9B D9 E4

Without FWAIT:

11011001 11100100
D9 E4

Example

```
FTST                    ;TEST
```

FWAIT (8087)
Wait (CPU Instruction)

Purpose

FWAIT causes the 8088 to wait until the current 8087 instruction is completed before the 8088 performs the next instruction.

Format

FWAIT

Remarks

FWAIT synchronizes the 8087 to the 8088. Use the FWAIT instruction for this purpose and not the 8088 WAIT instruction, which could cause an infinite wait.

FWAIT is an alternate mnemonic for the 8088 WAIT instruction. FWAIT and WAIT assemble with the same object code when you use the /R assembler option and with different object code when you use the /E assembler option.

Note: Use FWAIT instead of WAIT if you want 8087 emulator compatibility.

The routines that change an object program for 8087 emulation change any FWAITS to NOPS but do not change any WAITS. The program waits forever if it finds a WAIT during an emulated run because there is no active 8087 to drive the test pin of the 8088.

Logic

8088 wait

Exception Flags

None

Encoding

10011011

9B

Example

This example shows how the FWAIT instruction can be used to force the CPU to WAIT until the variable STATUS has been updated by the 8087 instruction FNSTSW to ensure that the information in STATUS is the most recently updated information.

```
FNSTSW      STATUS
FWAIT       ;Wait for FNSTSW
MOV         AX,STATUS
```

FXAM (8087)
Examine

Purpose

FXAM examines the contents of the 8087 stack top element (ST) and posts the result in the condition code field of the status word.

Format

FXAM

Remarks

The result of FXAM is posted to the condition code field of the status word as follows:

C3	C2	C1	C0	Interpretation
0	0	0	0	+ Unnormal
0	0	0	1	+ NaN
0	0	1	0	- Unnormal
0	0	1	1	- NaN
0	1	0	0	+ Normal
0	1	0	1	+ ∞
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

Note: Although four different internal representations can be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all internal representations. Bits C2 and C1 should be ignored when examining for empty.

Exception Flags

None

Encoding

10011011 11011001 11100101

9B D9 E5

Without FWAIT:

11011001 11100101
D9 E5

Example

```
FXAM            ;EXAMINE
```

FXCH (8087)
Exchange Registers

Purpose

FXCH swaps the contents of the *destination* and the 8087 stack top registers. If the *destination* is not coded explicitly, ST(1) is used.

Format

FXCH
or
FXCH *destination*

Remarks

Many 8087 instructions operate only on the 8087 stack top. FXCH provides a way of effectively using these instructions on the lower 8087 stack elements.

Exception Flags

I

FXCH (No Operands)

Format

FXCH

Logic

T_1 <- ST(1)
ST(1) <- ST
ST <- T_1

Encoding

10011011 11011001 11001000

9B D9 C8

Without FWAIT:

11011001 11001000
D9 C8

FXCH (destination)

Format

FXCH *destination*

Logic

T_1 <- ST(*i*)
ST(*i*) <- ST
ST <- T_1

Encoding

10011011 11011001 11001(*i*)

9B D9 C8 + (*i*)

Without FWAIT:

11011001 11001(*i*)
D9 C8 + (*i*)

Example

```
FXCH   ST(3)    ;EXCHANGE REGISTERS
FSQRT
FXCH   ST(3)
```

This example takes the square root of the the third register from the top of the 8087 stack.

FXTRACT (8087)
Extract Exponent and Significand

Purpose

FXTRACT factors the number in the 8087 stack top into a significand and an exponent expressed in real numbers. The exponent replaces the original operand in the 8087 stack top; then FXTRACT pushes the significand onto the 8087 stack.

Format

FXTRACT

Remarks

FXTRACT is useful with FBSTP for converting numbers in 8087 temporary real format to decimal representations (for example, for printing or displaying). It can also be useful for debugging, because it allows you to examine the exponent and significand parts of a real number.

Note: Numbers are stored internally in a biased exponent format. In this format a true zero exponent is expressed as 16383 or 3FFFH. FXTRACT converts this biased notation into true notation.

Logic

T_1 <- exponent(ST)
T_2 <- significand(ST)
ST <- T_1
push 8087 stack
ST <- T_2

Exception Flags

I

Encoding

10011011 11011001 11110100

9B D9 F4

Without FWAIT:

11011001 11110100
D9 F4

Example

```
FXTRACT   ;EXTRACT EXPONENT AND SIGNIFICAND
```

FYL2X (8087)
Y * Log_2X

Purpose

FYL2X calculates the $Z = Y*log_2X$ function.

Format

FYL2X

Remarks

X is taken from the top of the 8087 stack and Y from ST(1). FYL2X pops the 8087 stack and returns Z to the new 8087 stack top, replacing the Y operand.

The following function optimizes the calculation of log to any base other than 2 because a multiplication is required.

$log_nX = (1 \div (log_2n)) * log_2X$

Note: The operands must be in the ranges:

$0 < X < \infty$

$-\infty < Y < +\infty$

Logic

T_1 <- ST(1) * log_2(ST)
pop 8087 stack
ST <- T_1

Exception Flags

P (operands not checked)

Encoding

10011011 11011001 11110001

9B D9 F1

Without FWAIT:

11011001 11110001
D9 F1

Example

```
FYL2X           ;Y TIMES LOG BASE 2 OF X
```

FYL2XP1 (8087)
Y * Log

Purpose

FYL2XP1 calculates the $Z = Y*\log_2(X+1)$ function.

Format

FYL2XP1 (no operands)

Remarks

The value X is taken from the top of the 8087 stack and must be in the range $0 < |X| < (1-\sqrt{2}/2)$. The value Y is taken from ST(1) and must be in the range $-\infty < Y < +\infty$. FYL2XP1 pops the 8087 stack and returns **Z** at the new 8087 stack top, replacing Y.

This instruction provides improved accuracy over FYL2X when computing the log of a number very close to 1.

Note: See FYL2X for related information.

Logic

$T_1 <-$ ST + 1
$T_2 <-$ ST(1) * $\log_2 T_1$
pop 8087 stack
ST $<- T_2$.

Exception Flags

P (operands not checked)

Encoding

10011011 11011001 11111001

9B D9 F9

Without FWAIT:

11011001 11111001
D9 F9

Example

```
FYL2XP1         ;Y TIMES LOG BASE 2 OF (X+1)
```

HLT
Halt

Purpose

The HLT instruction causes the processor to enter its halt state.

Format

HLT

Remarks

The halt state is cleared by an enabled external interrupt or reset.

In protected mode, HLT is a privileged instruction which runs from privilege level 0 only.

Flags

None

Encoding

11110100

F4

Example

```
HLT
```

IDIV
Integer Division, Signed

Purpose

IDIV divides a dividend (NUMR) in the accumulator and its extension, by a divisor (DIVR) from the *source* operand. If the *source* operand is a Byte operand, AX is divided by the byte. IDIV returns the quotient (QUO) to AL and the remainder (REM) to AH. If the *source* operand is a Word operand, AX:DX is divided by the word. The quotient (QUO) is returned in AX and the remainder (REM) is returned in DX.

Format

IDIV *source*

Remarks

If the quotient is positive and greater than the maximum (MAX) or if the quotient is negative and less than 0-MAX-1 (as when division by zero is attempted), then QUO and REM are undefined, and the assembler produces type 0 interrupt. IDIV truncates non-integral quotients and returns a remainder with the same sign as the numerator.

If the division results in a value larger than the appropriate registers can hold, the assembler produces an interrupt of type 0. The flags are pushed onto the stack, IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack and IP is then filled with the word at 0. This sequence thus includes a FAR call to the interrupt handling procedure whose segment and offset are stored at locations 2 and 0.

If the division result can fit in the appropriate registers, the quotient is stored in AL or AX (for word operands) and the remainder in AH or DX, respectively.

Logic

```
(temp) <- NUMR
if (temp)/(DIVR) > 0
 and (temp)/(DIVR) > MAX
or (temp)/(DIVR) < 0
 and (temp)/(DIVR) < 0-MAX-1
  then
    (QUO),(REM) undefined
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- FLAGS
    (IF) <- 0
    (TF) <- 0
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- CS
    (CS) <- (2)
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- (IP)
    (IP) <- 0
else
    (QUO) <- (temp)/(DIVR)
    (REM) <- (temp)%(DIVR)
```

Flags

Affected— No valid flags result
Undefined— AF, CF, OF, PF, SF, ZF

Encoding

1111011w mod111r/m

F6 + w mod111r/m

If w=0, NUMR=AX, DIVR=EA, QUO=AL,
REM=AH, MAX=7FH.
If w=1, NUMR=DX:AX, DIVR=EA, QUO=AX,
REM=DX, MAX=7FFFH.

Example

To divide a word by a byte:

```
MOV   AX,NUMERATOR_WORD[BX]
IDIV  DIVISOR_BYTE[BX]
```

To divide a word by a word:

```
MOV   AX,NUMERATOR_WORD
CWD   ;CONVERTS WORD TO DOUBLEWORD
IDIV  DIVISOR_WORD
```

To divide a doubleword by a word:

```
MOV   DX,NUM_HI_WORD
MOV   AX,NUM_LO_WORD
IDIV  DIVISOR_WORD[SI]
```

(SEE DIV instruction)

IMUL
Integer Multiply

Purpose

IMUL does a signed multiplication of the accumulator (AL or AX) and the *source* operand.

Format

IMUL *source*

Remarks

If the source is a Byte operand, then *source* is multiplied by AL and the 16-bit signed result is left in AX. If *source* is a Word, then *source* is multiplied by AX and the 32-bit signed result is in DX and AX. (The low-order 16 bits go in AX and the high-order 16 bits go in DX.) If the high order half of the result is the sign extension of the low order half, the carry and overflow flags are set to zero; otherwise, they are set to 1.

Logic

```
(DEST) <- (LSRC)*(RSRC)
  where * is signed multiply
if (EXT) = sign-extension of (LOW) then
  (CF) <- 0
else (CF) <- 1
(OF) <- (CF)
```

Flags

Affected— CF, OF

Undefined—AF, PF, SF, ZF

Encoding

1111011w mod101r/m

F6 + w mod101r/m

If w = 0, LSRC = AL, RSRC = EA, DEST = AX,
EXT = AH, LOW = AL.
If w = 1, LSRC = AX, RSRC = EA, DEST = DX:AX

Example

Any of the memory operands in the following examples could be an indexed address-expression of the correct TYPE. LSRC_BYTE could be ARRAY[SI] if A were of type BYTE, and RSRC_WORD could be TABLE[BX][DI] if TABLE were of type WORD.

To multiply a byte by a byte:

```
MOV   AL,LSRC_BYTE
IMUL  RSRC_BYTE ;RESULT IN AX
```

To multiply a word by a word:

```
MOV   AX,LSRC_WORD
IMUL  RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

To multiply a byte by a word:

```
MOV   AL,MUL_BYTE
CBW   ;CONVERTS BYTE IN AL TO WORD IN AX
IMUL  RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

IMUL (80286)
Integer Immediate Multiply

Purpose

IMUL Immediate does a signed multiplication of a value by an *immediate* value allowing you to choose a *destination* register other than the combination of DX and AX.

Format

IMUL *destination, immediate*
or
IMUL *destination, source, immediate*

Remarks

IMUL Immediate requires three arguments:

- The *immediate* value
- The effective address of the second operand
- The register where the result is to be placed.

The two operands are the *immediate* value and the data at an effective address (which may be the same register where the result is placed, another register, or a memory location).

If the *immediate* operand is a byte, the processor sign extends it to 16 bits before multiplying. The result must be placed in one of the general-purpose registers. The result cannot exceed 16 bits without causing an overflow and only the lower 16 bits of the result are saved.

IMUL Immediate can also be used with unsigned operands because the low 16 bits of a signed or unsigned multiplication of two 16-bit values is the same.

If the high order half of the result is the sign extension of the low order half, the carry and overflow flags are reset; otherwise, they are set.

You must use the .286C pseudo-op to enable this instruction.

Logic

(DEST)<-(LSRC)*(RSRC)
  where * is signed multiply
if (EXT) = sign-extension of (LOW)
  then (CF)<-0
else (CF)<-1
(OF)<-(CF)

Flags

Affected— CF, OF

Undefined—AF, PF, SF, ZF

Encoding

011010s1 data [data if s = 0]

69 + s data [data if s = 0]

Example

In this example, IMUL replaces the contents of BX with the product of the contents of SI and an immediate value of 7.

```
IMUL BX,SI,7
```

In this example, IMUL replaces the contents of BX with the product of the contents of BX and an immediate value of 7.

```
IMUL BX,7
```

IN
Input Byte or Word

Purpose

The contents of the *accumulator* are replaced by the contents of the designated *port*.

Format

IN *accumulator,port*

Remarks

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.

The destination for input must be AX or AL and must be specified in order for the assembler to know the type of the input. The port names must be immediate values between 0 and 255, as used in the following example. The register name is DX, which must contain the requisite port location.

In protected mode, you can run IN only at a privilege level less than or equal to the value of IOPL in the flags register.

Logic

(DEST) <- (SRC)

Flags

None.

Fixed Port

Encoding

1110010w port

E4 + w port

If w = 0, SRC = port, DEST = AL.
If w = 1, SRC = port + 1, DEST = AX.

Example

```
IN AX,WORD_PORT  ;INPUT WORD TO AX
IN AL,BYTE_PORT  ;INPUT A BYTE TO AL
```

Variable Port

Encoding

1110110w

EC + w

If w = 0, SRC = (DX), DEST = AL.
If w = 1, SRC = (DX) + 1:(DX), DEST = AX.

Example

```
IN AX,DX ;INPUT A WORD TO AX
IN AL,DX ;INPUT A BYTE TO AL
```

INC
Increase Destination by One

Purpose

INC adds the operand and 1 and returns the result to the operand.

Format

INC *destination*

Remarks

The specified operand is increased by 1. There is no carry-out of the most significant bit.

Logic

(DEST) <- (DEST) + 1

Flags

Affected— AF, OF, PF, SF, ZF

Register Operand (Word)

Encoding

01000reg

40 + reg

DEST = REG

Example

```
INC AX
INC DI
```

Memory or Register Operand

Encoding

1111111w mod000r/m

FE + w mod000r/m

DEST = EA

Example

Increase register:

```
INC CX
INC BL
```

Increase memory:

```
INC   MEM_BYTE
INC   MEM_WORD[BX]
INC   BYTE PTR[BX]
;BYTE IN DATA SEGMENT AT OFFSET [BX]
INC   ALPHA[DI][BX]
INC   BYTE PTR [SI][BP]
;BYTE IN STACK SEGMENT AT OFF [SI + BP]
INC   WORD PTR [BX]
;INCREASES THE WORD IN DATA SEGMENT AT
; OFFSET [BX]
```

INS/INSB/INSW (80286)
Input from Port to String

Purpose

INS transfers a byte or word string element from an input *port* numbered by the DX register to the memory at ES:DI. The type of the first operand to INS determines whether a byte or a word is moved.

Format

INS *destination-string,port*
or
INSB
or
INSW

Remarks

The address of the destination is determined only by the contents of DI, not by the first operand to INS. You must load the correct index value into DI before running the INS, INSB, or the INSW instructions. The operand is used only to validate ES segment addressability and to determine the data type. The operand must be addressable from the ES register; no segment override is possible.

The *port* is addressed through the DX register; the *port* number cannot be specified as an immediate value.

INSB and INSW are synonyms for the byte and word INS instructions. They are simpler to use, requiring no operands; however, INSB and INSW do not check type or segment.

DI is advanced after the transfer is done. If the direction flag is 0, DI increases; if the direction flag is 1, DI decreases. DI is altered by 1 if a byte was moved, by 2 if a word was moved.

INS, INSB, and INSW can be preceded by the REP prefix for a block input of CX bytes or words. See the REP instruction for the details of this operation.

Note: Not all input port devices can handle the speed at which this instruction runs.

Logic

(DEST)<-(SRC)

Flags

None.

Encoding

0110110w

6C + w

If w = 0, DEST = byte
If w = 1, DEST = word

Example

```
INS BSTRING,DX  ;INPUT BYTE
INS WSTRING,DX  ;INPUT A WORD
INSB            ;INPUT BYTE
INSW            ;INPUT A WORD
```

INT
Interrupt

Purpose

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 1-byte form of this instruction produces a type-3 interrupt.

Format

INT *interrupt-type*

Remarks

INT decreases the Stack Pointer by 2 and pushes all flags into the stack. The interrupt and trap flags are then reset. SP is then decreased by 2 and the current contents of the CS register are pushed onto the stack. CS is then filled with the high order word of the doubleword interrupt vector (the segment base-address of the interrupt handling procedure for this *interrupt-type*).

SP is then decreased by 2 and the current contents of the Instruction Pointer are pushed onto the stack. IP is then filled with the low order word of the interrupt vector, located at absolute address TYPE*4. This completes an inter-segment FAR call to the procedure that is to process this *interrupt-type*.

See PUSHF, INTO, and IRET instructions.

Logic

```
(SP) <- (SP) - 2
((SP) + 1:(SP)) <- FLAGS
(IF) <- 0
(TF) <- 0
(SP) <- (SP) - 2
((SP) + 1:(SP)) <- (CS)
(CS) <- (TYPE*4 + 2)
(SP) <- (SP) - 2
```

((SP) + 1:(SP)) <- (IP)
(IP) <- (TYPE*4)

Flags

Affected— IF, TF

Encoding

1100110v type

CC + v type

If v = 0, type = 3.
If v = 1, type = TYPE.

Note: The operand must be immediate data, not a register or a memory reference.

Example

```
INT    3   ;ONE BYTE:(11001100)
INT    2   ;TWO BYTES:(11001101 00000010)
INT    67  ;TWO BYTES:(11001101 01000011)
IMM_44 EQU 44
INT    IMM_44 ;TWO BYTES:(11001101 00101100)
```

INT (80286P)
Interrupt

Purpose

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 1-byte form of this instruction produces a type-3 interrupt.

Format

INT *interrupt-type*

Remarks

There are four types of interrupts: non-maskable external, maskable external, type-implicit internal, and type-explicit internal. All interrupts use the Interrupt Descriptor Table, or IDT, implicitly, and the interrupt type selects a descriptor in the IDT in the same way as the index field of a VA selector selects the descriptor in GDT or LDT. For external interrupts, which include the protection faults (PF), either type is implied (NMI and PF), or it is given to the 80286 by external hardware in response to the interrupt acknowledge signal, INTA. Internal interrupts can also specify type in two ways: general interrupts (opcode = 0CDH) include the type in the instruction's second byte, while single-byte interrupts (opcode = 0CDH, type = 3), interrupt on overflow (opcode = 0CEH, type = 4), and array bounds-check interrupt (type = 5) all have implicit types.

Only three types of descriptors can appear in the IDT: fault gates, interrupt gates, and task gates. If the interrupt is fielded by a fault or interrupt gate, the 80286 responds identically except for the handling of IF in the service routine. The interrupt gates clear the flag, disabling further maskable external interrupts, while fault gates do not alter the IF value. Aside from the IF disposition, both cases cause the following sequence of events:

1. The new CS and IP are loaded from the gate and validated. Internal interrupts may only get access to visible gates to routines that are equally or more privileged than CPL.

2. If the interrupt results in a privilege level transition (that is, if the descriptor named by the selector in the int/fault gate has a numerically lower privilege level than CPL), then a new stack selector and stack pointer are loaded from the TSS save areas that correspond to the new privilege level, and the old SS and SP are pushed onto the new stack.
3. The flags of the interrupted task are pushed, followed by the VADW (CS:IP) of the interrupted instruction. Interrupts are considered to occur before or during the performance of an instruction. The TF and NC flags are both cleared.
4. If the interrupt was a TS, NP, SS, or GP protection fault, then the error code word determined by the fault type is pushed onto the stack.
5. The interrupt service routine begins as directed by the new values of CS:IP.

Fielding the interrupt with a task gate results in a task switch, and the outgoing task remains marked as busy. The incoming task is linked to the old task by storing the old contents of the task register (TR) in the link field of the new TSS and setting the nested-task flag (NT) of the new task. If an error code was defined by a protection fault, it is pushed onto the stack of the new task.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

```
If trap gate or interrupt gate then
  If CPL > DPL of segment named by gate selector then
    (SP) <- (SP) - 2
    ((SP)) <- (current TSS.SS)
    (SP) <- (SP) - 2
    ((SP)) <- (current TSS.SP)
  endif
  (SP) <- (SP) - 2
  ((SP)) <- FLAGS
  (SP) <- (SP) - 2
  ((SP)) <- (CS)
  (SP) <- (SP) - 2
```

```
  ((SP)) <- updated IP
  if type selects interrupt gate then FLAGS.IF <-0
  if error code defined then
    (SP) <- (SP) - 2
    ((SP)) <- error.code
  endif
  (FLAGS.NT) <- 0
  (FLAGS.TF) <- 0
endif
if task gate then
  switch tasks, setting FLAGS.NT in new task
  (new TSS.link) <- (old TR)
  if errorcode defined then
    (SP) <- (SP) - 2
    ((SP)) <- error.code
  endif
endif
```

Flags

Affected— N,PL,O,D,I,T,S,Z,A,P,C

Encoding

1100110v type

CC + v type

If v=0, type = 3.
If v=1, type = TYPE.

Note: The operand must be immediate data, not a register or a memory reference.

Example

```
INT    3   ;ONE BYTE:(11001100)
INT    2   ;TWO BYTES:(11001101 00000010)
INT    67  ;TWO BYTES:(11001101 01000011)
IMM_44 EQU 44
INT    IMM_44 ;TWO BYTES:(11001101 00101100)
```

INTO
Interrupt If Overflow

Purpose

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 not set, no operation takes place.

Format

INTO (no operands)

Remarks

If the OF is zero, no operation occurs. If OF is 1, INTO decreases the Stack Pointer by 2 and saves all flags onto the stack. The trap and interrupt flags are reset. SP is again decreased by 2 and the contents of CS are pushed into the stack. CS is then filled with the second word (segment) of the doubleword interrupt vector for a type 4 interrupt.

SP is again decreased by 2, and the current Instruction Pointer (pointing to the next instruction after INTO) is pushed onto the stack. IP is then filled with the first word of the type 4 doubleword interrupt vector, located at absolute location 16 (10H). This word is the offset of the procedure to handle type 4 interrupts. The segment base address must already be in CS. This completes a FAR call to the proper procedure.

Logic

```
if (OF) <- 1 then
  (SP) <- (SP) - 2
  ((SP) + 1:(SP)) <- FLAGS
  (IF) <- 0
  (TF) <- 0
  (SP) <- (SP) - 2
  ((SP) + 1:(SP)) <- (CS)
  (CS) <- (12H)
  (SP) <- (SP) - 2
  ((SP) + 1:(SP)) <- (IP)
  (IP) <- (10H)
```

Flags

IF = 0, TF = 0

Encoding

11001110

CE

Example

```
INTO
```

IRET
Interrupt Return

Purpose

IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

Format

IRET

Remarks

The Instruction Pointer is filled with the word at the top of the stack. The Stack Pointer is then increased by 2, and the CS register is filled with the word now at the top of the stack. This returns control to the point where the interrupt was found.

SP is again increased by 2, and the flags are restored from the appropriate bits of the word now at the top of the stack. (See the POPF instruction in this chapter.) SP is again increased by 2.

Logic

```
(IP) <- ((SP)+1:(SP))
(SP) <- (SP) + 2
(CS) <- ((SP)+1:(SP))
(SP) <- (SP) + 2
FLAGS <- ((SP)+1:(SP))
(SP) <- (SP)+2
```

For the 80286:

```
If FLAGS.NT=1 then
  new task = current TSS.link
else
  if return selector RPL = CPL then
    (IP) <- ((SP) + 1:(SP))
    (SP) <- (SP) + 2
    (CS) <- ((SP) + 1:(SP))
```

```
    (SP) <- (SP) + 2
    FLAGS <-((SP) + 1:(SP))
    (SP) <- (SP) + 2
  else return to outer privilege level:
    (IP) <- ((SP) + 1:(SP))
    (SP) <- (SP) + 2
    (CS) <- ((SP) + 1:(SP))
    (SP) <- (SP) + 2
    FLAGS <- ((SP) + 1:(SP))
    (SP) <- (SP) + 2
    TEMP <- ((SP) + 1:(SP))
    (SP) <- (SP) + 2
    (SS) <- ((SP) + 1:(SP))
    (SP) <- TEMP
    if DS or ES not valid at new CPL then
      (REG) <- 0
```

Flags

Affected— All

Protected mode:

Affected— All

Encoding

11001111

CF

Example

```
IRET
```

J(condition)
Jump Short If Condition Met

Purpose

J(condition) transfers control to the target operand. Conditional short jumps (except for JCXZ) test the flags. JCXZ tests the contents of the CX register for zero, rather than checking the flags.

Format

J(condition) *short-label*

Remarks

If the condition is true, then control takes a short jump to the label provided as the operand. The condition for each mnemonic is given in the following table along with the corresponding internal representation and description.

The target label must be within −128 to +127 bytes of the next instruction. This range is necessary for the assembler to construct a 1-byte signed displacement from the beginning of the next instruction. If the label is out of range, or if the label is a FAR label, then the assembler must perform a jump with the opposite condition around an unconditional jump to the non-short label.

Note: Above and below refer to the relation between two unsigned values. Greater and less refer to the relation between two signed values.

Logic

If Condition Met then
(IP) <- (IP) + disp (sign-extended to 16-bits)

Note: See the following table for JUMP SHORT instructions and their conditions to be tested.

Inst	Jump short if	Condition	Internal Representation
JA	above	CF = 0 and ZF = 0	77 disp
JAE	above or equal	CF = 0	73 disp
JB	below	CF = 1	72 disp
JBE	below or equal	CF = 1 or ZF = 1	76 disp
JC	carry	CF = 1	72 disp
JCXZ	CX register is zero	(CF or ZF) = 0	E3 disp
JE	equal	ZF = 1	74 disp
JG	greater	ZF = 0 and SF = OF	7F disp
JGE	greater or equal	SF = OF	7D disp
JL	less	(SF xor OF) = 1	7C disp
JLE	less or equal	((SF xor OF) or ZF) = 1	7E disp
JNA	not above	CF = 1 or ZF = 1	76 disp
JNAE	not above nor equal	CF = 1	72 disp
JNB	not below	CF = 0	73 disp
JNBE	not below nor equal	CF = 0 and ZF = 0	77 disp
JNC	not carry	CF = 0	73 disp
JNE	not equal	ZF = 0	75 disp
JNG	not greater	((SF xor OF) or ZF) = 1	7E disp
JNGE	not greater nor equal	(SF xor OF) = 1	7C disp
JNL	not less	SF = OF	7D disp
JNLE	not less nor equal	ZF = 0 and SF = OF	7F disp
JNO	not overflow	OF = 0	71 disp
JNP	not parity	PF = 0	7B disp
JNS	not sign	SF = 0	79 disp
JNZ	not zero	ZF = 0	75 disp
JO	overflow	OF = 1	70 disp
JP	parity	PF = 1	7A disp
JPE	parity even	PF = 1	7A disp
JPO	parity odd	PF = 0	7B disp

Inst	Jump short if	Condition	Internal Representation
JS	sign	SF = 1	78 disp
JZ	zero	ZF = 1	74 disp

Flags

None

Encoding

01110111 disp (for JA/JNBE)

77 disp (for JA/JNBE)

Note: See the previous table for other JUMP SHORT instructions and their internal representations.

Example

```
JA   TARGET_LABEL
JNBE TARGET_LABEL
```

JMP
Jump

Purpose

JMP transfers control to the *target* operand.

Format

JMP *target*

Remarks

The jump is relative to the segment base address in the CS register. A direct jump uses the offset (and segment, if FAR) byte that follows the instruction byte. Indirect jumps use the contents of the location addressed by the bytes that follow the instruction byte.

The Instruction Pointer is replaced by the offset of the *target* in all inter-segment jumps and in intra-segment (or intra-group) indirect jumps.

When the jump is a direct intra-segment or intra-group, the distance from the end of the instruction to the *target* label is added to the IP.

Inter-segment jumps first replace the contents of CS, using the second word following the instructions (direct) or using the second word following the indicated data address (indirect).

Logic

If inter-segment then (CS) <- SEG
(IP) <- DEST

Flags

None

Intra-Segment or Intra-Group Direct

Encoding

11101001 disp-low disp-high

E9 disp-low disp-high

DEST = (IP) + disp

Example

```
JMP   NEAR_LABEL
```

Intra-Segment Direct Short

Encoding

11101011 disp

EB disp

DEST = (IP) + disp (sign extended to 16 bits)

Example

```
JMP   TARGET_LABEL
JMP   SHORT NEAR_LABEL
```

Note: The target label must be within −128 to +127 bytes of the next instruction.

Inter-Segment Direct

Encoding

11101010 offset segment

EA offset segment

DEST = offset, SEG = seg

Example

```
JMP   LABEL_DECLARED_FAR
JMP   FAR PTR LABEL_NAME
JMP   FAR PTR NEAR_LABEL
```

Inter-Segment Indirect

Encoding

11111111 mod101r/m

FF mod101r/m

DEST = (EA), SEG = (EA + 2)

Example

```
JMP   VAR_DOUBLEWORD
JMP   DWORD PTR [BX][SI]
JMP   ALPHA [BP][DI]
```

Intra-Segment or Intra-Group Indirect

Encoding

11111111 mod100r/m

FF mod100r/m

DEST = (EA)

Example

```
JMP   TABLE[BX]
JMP   WORDPTR [BX][DI]
JMP   BETA_WORD
JMP   AX
JMP   SI
JMP   BP
```

Note: These last three replace the Instruction Pointer by the contents of the named register. This causes a jump directly to the byte with that offset past CS. This is different from the direct intra-segment jumps, which are self-relative, causing an add to the IP.

JMP (80286P)
Jump

Purpose

JMP transfers control to the *target* operand.

Format

JMP *target*

Remarks

The long jumps transfer control using a VADW, which may be either included in the instruction itself or found in a DWORD variable. The type of control transfer is determined by the selector part of the VADW as follows:

1. If the selector names a descriptor for an executable segment, then that selector replaces CS and the offset part of the VADW replaces IP, subject to protection mechanism addressability and visibility.
2. If the selector names a call-gate descriptor, then the offset part of the VADW is ignored, and the virtual address of the routine being entered is taken from the call-gate.
3. If the selector names a task gate descriptor, then context of the current task is saved in its Task State Segment (TSS), and the TSS named in the task-gate is used to load the new context. The outgoing task is marked not-busy, the new TSS marked busy, and running resumes at the point at which the new task was last suspended.
4. If the selector names a TSS, then the current task is suspended and the new task is initiated as in the list item above, except that there is no intervening gate.

In general, for the task-state to be considered valid, you must follow these limits to the contents of the segment registers:

- For CS, the selector must name an executable segment.
- SS must name a writable segment with privilege equal to CPL.
- DS and ES must either be zero (a value which represents an unloaded condition because it selects entry 0 in the GDT), or must select a readable segment which is visible at the CPL. A segment is visible to any CPL which has a numerically smaller privilege level. Also, conforming segments, which must be executable but which may be readable, are visible to any privilege level.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

Jump segment descriptor:
(CS) <- selector
(IP) <- offset
Jump call gate:
(CS) <- call-gate.selector
(IP) <- call-gate.offset
Jump task gate:
(TR) <- task.gate selector
Jump task state segment:
(TR) <- selector

Flags

Affected— N,PL,O,D,I,T,S,Z,A,P,C

Direct Virtual Address

Encoding

1110 1010 offset selector
E A offset selector

DEST = offset, SEG = selector

Example

```
JMP FAR_LABEL
JMP CALL_GATE
JMP TASK_GATE
JMP TASK_XXX
```

Indirect Virtual Address

Encoding

1111 1111 mod101r/m
F F mod101r/m

DEST = offset or gate.offset,
SEG = selector or gate.selector

Example

```
JMP CASE_TABLE[BX]
```

LAHF
Load AH from Flags

Purpose

LAHF transfers the flag registers SF, ZF, AF, PF, and CF into specific bits of the AH register. The bits shown as X are unspecified.

Format

LAHF (no operands)

Remarks

Specific bits of AH are filled from the following flags: The sign flag fills bit 7. The zero flag fills bit 6. The auxiliary carry flag fills bit 4. The parity flag fills bit 2. The carry flag fills bit 0. Bits 1, 3, and 5 of AH are unknown.

Logic

(AH) <- (SF):(ZF):X:(AF):X:(PF):X:(CF)

Flags

None

Encoding

10011111

9F

Example

```
LAHF
```

LAR (80286P)
Load Access Rights

Purpose

If the descriptor shown by the selector in the right operand is visible at the CPL, LAR loads a word consisting of the access rights byte of the descriptor in the high byte and a low byte of zero into the left operand.

Format

LAR *destination,source*

Remarks

The *source* operand (the right operand) can be a memory location or a register. The *destination* operand (the left operand) must be a register. ZF is set if the descriptor was visible, and it is cleared otherwise.

The .286P pseudo-op must be used to enable this instruction.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

If the descriptor named by the selector in right operand is visible at CPL then
 ZF <- 1
 left operand (reg) <- descriptor access rights
else ZF <- 0

Flags

Affected— ZF

Encoding

```
00001111  00000010  modregr/m
  0F        02      modregr/m
```

Example

```
LAR AX,SI
   or
LAR BX,SELECTOR
```

LDS
Load Data Segment Register

Purpose

LDS transfers a pointer-object (a 32-bit object containing an offset address and a segment address) from the doubleword *source* operand (which must be a memory operand) to a pair of *destination* registers. The segment address is transferred to the DS segment register. The offset address can be transferred to any 16-bit general, pointer, or index register you specify (not a segment register).

Format

LDS *destination,source*

Remarks

The contents of the specified register are replaced by the lower addressed word of the doubleword memory operand.

The contents of the DS register are replaced by the higher-addressed word of the doubleword memory operand.

In protected mode, the doubleword *source* operand is a virtual address. The second word is a selector instead of a segment address. The loading of DS initiates automatic loading of the descriptor information associated with the selector into the hidden part of the segment register and validation of both selector and descriptor information.

Logic

(REG) <- (EA)
(DS) <- (EA + 2)

Flags

None

Encoding

11000101 modregr/m

C5 modregr/m

For mod $\neq$ 11.
If mod = 11, undefined operation.

Example

```
LDS   BX,ADDR_TABLE[SI]
LDS   SI,DWORD PTR NEWSEG[BX]
```

LEA
Load Effective Address

Purpose

LEA transfers the offset address of the *source* operand to the *destination* register operand. The contents of the specified register are replaced by the offset of the indicated variable, label or address-expression.

Format

LEA *destination,source*

Remarks

The *source* operand must be a symbol defining a memory location and the *destination* operand can be any 16-bit general, pointer, or index register.

LEA allows the *source* to be subscripted. This is not allowed using the MOV instruction with the OFFSET operator. Also, the latter operation uses the offset of the variable in the segment where it was defined. LEA, however, takes into account a group offset if the group is the only possible access route via the latest ASSUME pseudo-op. (See Chapter 3, "Pseudo Operations," in this book.)

Logic

(REG) <- EA

Flags

None

Encoding

10001101 modregr/m

8D modregr/m

For mod $\neq$ 11.
If mod = 11, undefined operation.

Example

```
LEA   BX,VARIABLE_7
LEA   DX,BETA[BX][S1]
LEA   AX,[BP][DI]
```

LEAVE (80286)
High Level Procedure Exit

Purpose

LEAVE performs a procedure return for a high-level language.

Format

LEAVE

Remarks

LEAVE is the complementary operation to ENTER; it reverses the effects of that instruction. The LEAVE instruction frees all local variables and restores the SP and BP registers to their values immediately after the procedure's call.

LEAVE releases the stack space used by a procedure by copying BP to SP. The old frame pointer is now popped into BP, restoring the frame of the caller, and a subsequent RET *nn* instruction follows the back-link and removes any arguments pushed on the stack for the exiting procedure.

Do not confuse LEAVE with the $LEAVE SALUT structure statement. See the discussion of SALUT structure statements in the IBM *Macro Assembler/2 Assemble, Link, and Run* book.

The .286C pseudo-op is required.

Logic

(SP) <- (BP)
(BP) <- ((SP) + 1:(SP))
(SP) <- (SP) + 2

Flags

Affected— None
Undefined— None

Encoding

11001001

C9

Example

```
LEAVE
```

LES
Load Extra Segment Register

Purpose

LES transfers a pointer (an offset address and a segment address) from the doubleword *source* operand (which must be a memory operand) to a pair of *destination* registers. LES transfers the segment address to the ES segment register. LES transfers the offset address to a 16-bit general, pointer, or index register (not a segment register).

Format

LES *destination,source*

Remarks

LES replaces the contents of the specified register by the lower-addressed word of the doubleword memory operand. LES replaces the contents of the ES register by the higher-addressed word of the doubleword memory operand.

In protected mode, the doubleword *source* operand is a virtual address. The second word is a selector instead of a segment address. Loading ES starts automatic loading of the descriptor information associated with the selector into the hidden part of the segment register; loading also starts validation of both selector and descriptor information.

Logic

(REG) <- (EA)
(ES) <- (EA + 2)

Flags

None

Encoding

11000100 modregr/m

C4 modregr/m

For mod $\neq$ 11.
If mod = 11, undefined operation.

Example

```
LES   BX,ADDR_TABLE[SI]
LES   DI,DWORD PTR NEWSEG[BX]
```

LGDT (80286P)
Load Global Descriptor Table

Purpose

LGDT loads the Global Descriptor Table Register from the memory pointed to by *source*.

Format

LGDT *source*

Remarks

The operand addresses a 6-byte area in memory. The LIMIT field of GDTR loads from the first word at the EA; the next 3 bytes go to the BASE field of GDTR. LGDT is a privileged instruction that can only be run at level 0.

The SIDT and SGDT 80286 protected mode instructions operate on 6-byte quantities. Since the assembler has no way to define a 6-byte data item, it uses the first 6 bytes of the next largest type of data item, which is a QWORD. There are several ways of specifying the source operand for these instructions:

- a data item defined with DQ
- a symbol defined with EXTRN X:QWORD
- a symbol defined using LABEL QWORD
- use the QWORD PTR override.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL=0 then
  GDTR.BASE <- (EA + 4:EA + 2)
  GDTR.LIMIT <- (EA + 1:EA)
```

Flags

None

Encoding

```
00001111  00000001  mod010r/m
  0F        01      mod010r/m
```

Example

```
        .286P
        EXTRN       VAR1:QWORD
DATA    SEGMENT
        ASSUME      DS:DATA
VAR2    DQ          ?
VAR3    LABEL       QWORD
VAR4    DB          6 DUP(?)
DATA    ENDS
CODE    SEGMENT
        ASSUME      CS:CODE
        .
        .
        .
        LIDT        VAR1              ;external data item
        LGDT        VAR2              ;data field of type QWORD
        LIDT        VAR3              ;label of type QWORD
        LGDT        QWORD PTR VAR4    ;explicit override
```

LIDT (80286P)
Load Interrupt Descriptor Table

Purpose

LIDT loads the Interrupt Descriptor Table Register from the 6-byte *source*.

Format

LIDT *source*

Remarks

The operand addresses a 6-byte area in memory. The LIMIT field of the IDTR loads from the first word; the next 3 bytes go to the BASE field of the register. LIDT is a privileged instruction and can only be run at level 0.

The LIDT and LGDT 80286 protected mode instructions operate on 6-byte quantities. Since the assembler has no way to define a 6-byte data item, it uses the first 6 bytes of the next largest type of data item, which is a QWORD. There are several ways of specifying the *source* operand for these instructions:

- a data item defined with DQ
- a symbol defined with EXTRN X:QWORD
- a symbol defined using LABEL QWORD
- use the QWORD PTR override.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL = 0 then
  IDTR.BASE <- (EA + 4:EA + 2)
  IDTR.LIMIT <- (EA + 1:EA)
```

Flags

None

Encoding

```
00001111  00000001  mod011r/m
  0F        01      mod011r/m
```

Example

See the example under the LGDT description in this chapter.

LLDT (80286P)
Load Local Descriptor Table

Purpose

LLDT loads the Local Descriptor Table Register from the *source* operand.

Format

LLDT *source*

Remarks

The *source* operand can be either a register or a memory location. The operand should contain a selector pointing to a Global Descriptor Table Entry. This entry should be a Local Descriptor Table descriptor which will be loaded into the LDTR. The hidden descriptor fields for the segment registers are not affected and the LDT field of the TSS is not changed. If the *source* operand is 0, the LDTR is marked incorrect and all descriptor references cause GP faults with error codes of zero. LLDT works only at privilege level 0.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL=0 then
  LDTR <- OPERAND
```

Flags

None

Encoding

```
00001111  00000000  mod010r/m
  0F        00      mod010r/m
```

Example

```
LLDT   BX
   or
LLDT   MEMLOC
```

LMSW (80286P)
Load Machine Status Word

Purpose

LMSW loads the Machine Status Word (MSW) from the *source* operand and can be used to switch to protected mode.

Format

LMSW *source*

Remarks

The *source* operand can be either a register or a memory location. LMSW works in either real mode or in protected mode at level 0.

If LMSW is used to switch to protected mode, it must be followed by an intra-segment jump to clear the instruction queue. LMSW cannot be used to switch back to real mode.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL=0 then
  MSW <- OPERAND
```

Flags

None

Encoding

```
00001111  00000001  mod110r/m
  0F        01      mod110r/m
```

Example

```
LMSW   BX
    or
LMSW   MEMABC
```

LOCK
Lock Bus

Purpose

A special 1-byte lock prefix can precede any instruction. It causes the processor to assert a bus-lock signal during the operation caused by the instruction. In multiple processor systems with shared resources, you must provide mechanisms to enforce controlled access to those resources. Such mechanisms, while generally provided through operating systems, require hardware assistance. A mechanism for accomplishing this is a locked exchange (also called test-and-set-lock). See the XCH6 instruction description in this chapter.

Format

LOCK

Remarks

The instruction that is most useful in this context is an exchange register with memory. You can use a simple software lock with the following code sequence.

In protected mode, LOCK requires, at minimum, IOPL.

Flags

None

Encoding

11110000

F0

Example

```
CHECK: MOV AL,1 ;SET AL TO 1
;                       (IMPLIES LOCKED)
LOCK   XCHG  SEMO,AL ;TEST AND SET LOCK
        TEST  AL,AL  ;SET FLAGS BASED ON AL
        JNZ   CHECK  ;RETRY IF LOCK ALREADY SET
        .
        .
        .
        MOV  SEMO,0  ;CLEAR THE LOCK WHEN DONE
```

LODS/LODSB/LODSW
Load Byte or Word String

Purpose

LODS transfers a byte (or word) operand from the *source* operand addressed by SI to accumulator AL (or AX) and adjusts the SI register by delta. Delta is 1 if a byte was moved, 2 if a word was moved. This operation normally would not be repeated.

Format

LODS *source-string*
or
LODSB
or
LODSW

Remarks

The assembler determines from the *source* operand if the reference is a byte or a word. The *source* byte (or word) is loaded into AL (or AX). The Source Index is increased by 1 (or 2, for word strings) if the Direction Flag is reset; otherwise, SI is decreased by 1 (or 2).

The other two forms of this instruction, LODSB and LODSW, require no operand since the mnemonic itself specifies whether the operation is for a byte or a word string.

Logic

(DEST) <- (SRC)
If (DF) = 0, (SI) <- (SI) + DELTA.
Else (SI) <- (SI) - DELTA.

Flags

None

Encoding

1010110w

AC + w

If w=0,
SRC = (SI), DEST = AL, DELTA = 1.
If w=1,
SRC = (SI) + 1:(SI), DEST = AX, DELTA = 1.

Example

```
CLD     ;CLEARS DIRECTION FLAGS SO
;        SI WILL BE INCREASED
MOV    SI,OFFSET    BYTE_STRING
LODS   BYTE_STRING    ;SI=SI+1
.
.
.
STD     ;SETS (DF) SO SI
;        WILL BE DECREASED
MOV    SI,OFFSET     WORD_STRING
LODS   WORD_STRING    ;SI=SI-2
```

In the above example, (DF) = 1 implies that the variable *word_string* names the last or highest-addressed word in the string. The operand named in the LODS instruction is used only by the assembler to verify type and accessibility using correct segment register contents. LODS uses only SI to point to the location whose contents are to be loaded into the accumulator, without using the name given in the *source* instruction.

The string instructions are unusual in several aspects:

1. They load SI with the offset of the *source-string*.
2. They load DI with the offset of the *destination-string*.
3. You can code each with or without symbolic memory operands.
    - If you code symbolic operands, the assembler can check whether you can address them.
    - Code references that use hardware defaults using the operand-less forms (LODSB and LODSW), to avoid the additional pointer information.
    - Do not use [BX] or [BP] addressing modes with the string instructions.
4. If you code the instruction mnemonic without operands, SI defaults to an offset in the segment addressed by DS.

LOOP
Loop Until Count Complete

Purpose

LOOP decreases the CX (count) register by 1 and transfers control to the target operand (*short-label*) if CX is not zero.

Format

LOOP *short-label*

Remarks

LOOP decreases the Count register (CX) by 1. If the new CX is not zero, LOOP adds the distance from the end of this instruction to the target label, *short-label*, to the instruction pointer, causing the jump. If CX = 0, no jump occurs.

The target label must be within −128 to +127 byte of the next instruction.

Logic

(CX) <- (CX) - 1
If (CX) ≠ 0 then
  (IP) <- (IP) + disp (sign-extended to 16-bits)

Flags

None

Encoding

11100010

E2

Example

The following sequence computes the 16-bit check-sum of a non-null array:

```
      MOV   CX,LENGTH ARRAY
      MOV   AX,0
      MOV   SI,AX
;
NEXT: ADD   AX,ARRAY[SI]
      ADD   SI,TYPE ARRAY
      LOOP  NEXT
      MOV   CKS,AX
```

In the following example, the assembler runs the instructions from FIB to LL *N* times and stores into the FIBONACCI array the first *N* terms of the sequence. (1,1,2,3,5,8,13,21 ...)

```
      MOV   AX,0
      MOV   BX,1
      MOV   CX,N  ;NUMBER OP TERMS
      MOV   DI,AX
;
FIB:  MOV   SI,AX
      ADD   AX,BX
      MOV   BX,SI
      MOV   FIBONACCI[DI],AX
      ADD   DI,TYPE FIBONACCI
;
LL:   LOOP  FIB
```

LOOPE/LOOPZ
Loop If Equal/If Zero

Purpose

LOOPE or LOOPZ decreases the CX register by 1 and transfers control to *short-label* if CX is not zero and if the ZF flag is set to 1.

Format

LOOPE *short-label*
or
LOOPZ *short-label*

Remarks

The Count register (CX) is decreased by 1. If the zero flag is set and (CX) is not yet 0, the distance from the end of this instruction to the target label, *short-label*, is added to the instruction pointer, causing the jump. No jump occurs if (ZF) = 0 or if (CX) = 0.

The target label must be within −128 to +127 bytes of the next instruction.

Logic

(CX) <- (CX) - 1
If (ZF) = 1 and (CX) ≠ 0 then
  (IP) <- (IP) + disp (sign-extended to 16-bits)

Flags

None

Encoding

11100001 disp

E1 disp

Example

The following sequence finds the first non-zero entry in a byte array:

```
          MOV    CX,LENGTH ARRAY
          MOV    SI,-1
NEXT:     INC    SI
          CMP    ARRAY[SI],  0
          LOOPE  NEXT
          JNE    OKENTRY
                  ;ARRIVE HERE IF COMPLETE
                  ;  ARRAY IS ZERO

OKENTRY:         ;SI TELLS WHICH ENTRY
;                 IS NOT ZERO
```

LOOPNE/LOOPNZ
Loop If Not Equal/If Not Zero

Purpose

LOOPNE or LOOPNZ decreases the CX register by 1 and transfers control to *short-label* if CX is not 0 and the ZF flag is zero.

Format

LOOPNE *short-label*
or
LOOPNZ *short-label*

Remarks

LOOPNE or LOOPNZ decreases the Count register (CX) by 1. If the new (CX) is not 0 and the zero flag is reset, LOOPNE or LOOPNZ adds the distance from the end of this instruction to the target label, *short-label*, to the instruction pointer, causing the jump. If (CX) = 0 or if (ZF) = 1, no jump occurs.

The target label must be within −128 to +127 bytes of the next instruction.

Logic

(CX) <- (CX) - 1
If (ZF) = 0 and (CX) ≠ 0 then
  (IP) <- (IP) + disp (sign-extended to 16-bits)

Flags

None

Encoding

11100000 disp

E0 disp

Example

The following sequence computes the sum of 2 byte arrays, each of length *N*, only up to the point of finding zero entries in both arrays at the same time. At that point, the expression SI-1 gives the length of the nonzero sum arrays.

```
        MOV     AX,0
        MOV     SI-1
        MOV     CX,N
NOZERO: INC     SI
        MOV     AL,ARRAY1[SI]
        ADD     AX,ARRAY2[SI]
        MOV     SUM[SI],AX
        LOOPNZ  NOZERO
```

The following sequence searches down a linked list for the last element. This is the element with a zero in the word that normally contains the address of the next element. This word is located the same number of bytes past the beginning of each list element. LINK is the name for that absolute number of bytes, for example:

```
      LINK    EQU 7
      MOV     AX,OFFSET HEAD_OF_LIST
      MOV     CX,1000 ;SEARCH AT MOST 1000
;                     ENTRIES
NEXT: MOV     BX,AX
      MOV     AX,[BX] + LINK
      CMP     AX,0
      LOOPNE  NEXT
```

LSL (80286P)
Load Segment Limit

Purpose

LSL loads a word consisting of the limit field of the descriptor into the left operand, if the descriptor shown by the selector in the right operand is visible at the CPL.

Format

LSL *destination,source*

Remarks

The *destination* (left) operand must be a register. The *source* (right) operand can be a register or memory location. If the loading is performed (the descriptor was visible), the zero flag is set. Otherwise, the zero flag is cleared. LSL returns only the limit field of segments, TSSs and LDTs.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

If the descriptor named by the selector in the right operand is visible at CPL then
  ZF <- 1
  REG16 <- RSRC
else ZF <- 0

Flags

Affected—ZF

Encoding

```
00001111  00000011  modregr/m
  0F        03      modregr/m
```

Example

```
LSL   AX,SI
   or
LSL  BX,SELECTOR
```

LTR (80286P)
Load Task Register

Purpose

LTR loads the Task Register from the *source* operand.

Format

LTR *source*

Remarks

The *source* operand can be either a register or memory location. The processor marks the loaded TSS busy by setting bit 1 of the access byte to 1. LTR works only in protected mode at level 0.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

```
if CPL=0 then
  TR <- SRC
```

Flags

None

Encoding

```
00001111  00000000  mod011r/m
  0F        00      mod011r/m
```

Example

```
LTR BX
   or
LTR AREA
```

MOV
Move

Purpose

MOV copies the *source* operand to the *destination* operand. The *source* operand is not changed.

Format

MOV *destination,source*

Remarks

The types of move instructions are described below. Each type has multiple uses and internal representations, depending on the type of data being moved and the location of that data. The assembler produces the correct internal representation based on the type and location of data. If the *destination* is a register, the bit shown as "d" is a 1; otherwise, it is a 0. If the type is a word, the bit shown as "w" is a 1; otherwise, it is a 0.

Logic

(DEST) <- (SRC)

Flags

None

TO Memory FROM Accumulator

Encoding

1010001w addr-low addr-high

A2 + w addr-low addr-high

If w = 0, SRC = AL, DEST = addr.
If w = 1, SRC = AX, DEST = addr + 1:addr.

Example

```
MOV   ALPHA_MEM,AX
MOV   GAMMA_BYTE, AL
MOV   CS:DATUM_BYTE,AL
MOV   ES:ARRAY[BX][SI],AX
```

TO Accumulator FROM Memory

Encoding

1010000w addr-low addr-high

A0 + w addr-low addr-high

If w = 0, SRC = addr, DEST = AL.
If w = 1, SRC = addr + 1:addr, DEST = AX.

Example

```
MOV   AX,BETA_MEM
MOV   AL,GAMMA_BYTE
MOV   AX,ES:ARRAY[BX][SI]
MOV   AL,SS:OTHER_BYTE
```

TO Segment Register FROM Memory-or-Register Operand

Encoding

10001110 modregr/m

8E modregr/m

If reg $\neq$ 01 then SRC = EA, DEST = REG else undefined operation.

Example

```
MOV   ES,DX
MOV   DS,AX
MOV   SS,BX
MOV   ES,SS:NEW_WORD[DI]
```

Note: CS is incorrect as a destination here.

TO Memory-or-Register FROM Segment Register

Encoding

10001100 modregr/m

8C modregr/m

SRC = REG, DEST = EA, (DEST) = (SRC)

Example

```
MOV   DX,DS
MOV   BX,ES
MOV   ARRAY[BX][SI],SS
MOV   BETA_MEM_WORD,DS
MOV   GAMMA,CS
```

Note: CS *is* correct as a source here.

To Register From Register

See "To Memory-Or-Register Operand From Register" below.

To Register From Memory-or-Register Operand

See "To Memory-Or-Register Operand From Register" below.

To Memory-or-Register Operand From Register

Encoding

100010d1 modregr/m addr-low* addr-high*

89 + d modregr/m addr-low* addr-high*

If d = 1, SRC = EA, DEST = REG
else SRC = REG, DEST = EA
If d = 0, SRC = REG, DEST = EA.

These bytes are omitted in register to register moves, when mod = 1,

```
MOV   CX,DX
```

and also when the address-expression to memory is register-indirect with no variable-name-displacement, for example:

```
MOV   [BX][SI],DX
MOV   AX,[BP][DI]
```

Example

To register from register:

```
MOV   AX,BX
MOV   CL,DH
MOV   CX,DI
```

To register from memory or register:

```
MOV   AX,MEM_VALUE
MOV   DX,ARRAY[SI]
MOV   DI,MEM[BX][DI]
```

To memory or register from register:

```
MOV   ARRAY[DI],DX
MOV   MEM_VALUE,AX
MOV   [BX][SI],DI
```

TO Register FROM Immediate-data

Encoding

1011 w reg data data-high*

B + w + reg data data-high*

SRC = data, DEST = REG

*Present only if w = 1

Example

```
MOV   AX,77
MOV   BX,VALUE_14_IMM
MOV   SI,EQU_VAL_9
MOV   DI,618
```

TO Memory-or-Register Operand FROM Immediate-data

Encoding

1100011w mod000r/m data data-high*

C6 + w mod000r/m data data-high*

SRC = data, DEST = EA *Present only if w = 1

Example

```
MOV   ARRAY[BX][SI],DATA_4
MOV   MEM_BYTE,IMM_BYTE_3
MOV   BYTE PTR [DI],66
MOV   MEM_WORD,1999
MOV   BX,84
MOV   DS:MEM_WORD[BP],3989
```

MOVS/MOVSB/MOVSW
Move Byte or Word String

Purpose

MOVS transfers a byte (or word) operand from the *source* operand addressed by SI to the *destination* operand addressed by DI, and adjusts the SI and DI registers by delta (number of bytes specified by the operand TYPE). Delta is 1 if a byte was moved, 2 if a word was moved. As a repeated operation, this provides for moving a string from one location in memory to another.

Format

MOVS *dest-string,source-string*
or
MOVSB
or
MOVSW

Remarks

The *source* string whose offset is in the Source Index is moved into the location in the ES. The offset of the ES is in the DI. SI and DI are then both increased, if the direction flag is zero, or both decreased, if (DF) = 1. (See the CLD and STD instructions.) The increase or decrease is 1 for byte strings, 2 for word strings.

Logic

```
(DEST) <- (SRC)
If (DF) = 0 then
  (SI) <- (SI) + DELTA
  (DI) <- (DI) + DELTA
else
  (SI) <- (SI) - DELTA
  (DI) <- (DI) - DELTA
```

Flags

None

Encoding

1010010w

A4 + w

If w = 0, SRC = (SI), DEST = (DI), DELTA = 1.
If w = 1, SRC = (SI) + 1:(SI), DEST = (DI) + 1:(DI),
DELTA = 2

Example

For this example, assume type source = BYTE.

```
SOURCE  DB      17 DUP  ' '
DEST    DB      17 DUP  (?)
        .
        .
        .
        CLD  ;CLEAR DF TO AUTO-INCREMENT
        MOV     SI,OFFSET SOURCE
        MOV     DI,OFFSET DEST
        MOV     CX,LENGTH SOURCE ;PASS NUMBER OF
;                                  BYTES IN SOURCE
REP     MOVS    DEST,SOURCE
        or
REP     MOVS    ES:BYTE PTR[DI],DS:[SI]
        or
REP     MOVSB
```

The above sequence moves the complete *source* string (in any segment that can be reached by current segment registers) into the *destination* locations in the extra segment (the ES register is used for DI operands in string operations). See also "REP." The operands named in the string operation are used only by the assembler to verify type and accessibility using the contents of the current segment register. MOVS moves the byte pointed at by SI to the byte pointed at by DI in ES, without using the names given in the source MOVS instruction.

The string instructions are unusual in several aspects:

1. They load SI with the offset of the *source-string.*
2. They load DI with the offset of the *destination-string.*
3. They can be coded with or without symbolic memory operands.

- If you code symbolic operands, the assembler can check the addressability of them for you.
- Code references that use hardware defaults should be coded using the operand-less forms (MOVSB and MOVSW), to avoid the additional pointer information.
- Do not use [BX] or [BP] addressing modes with the string instructions.

4. If you code the instruction mnemonic without operands, the segment registers 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.

MUL
Multiply, Unsigned

Purpose

MUL does an unsigned multiplication of the accumulator and the *source* operand.

Format

MUL *source*

Remarks

If *source* is a Byte operand, then it is multiplied by AL and the 16-bit result is left in AX. If *source* is a Word, then it is multiplied by AX and the 32-bit result goes in DX:AX, with DX containing the high-order 16 bits of the product. If the high-order half of the result is zero, the carry and overflow flags are reset; otherwise, they are set.

Logic

(DEST) <- (LSRC) * (RSRC),
  where * is unsigned multiply
if (EXT) = 0 then (CF) <- 0
else (CF) <- 1;
(OF) <- (CF)

Flags

Affected— CF,OF
Undefined— AF,PF,SF,ZF

Encoding

1111011w mod100r/m

F6 + w mod100r/m

If w = 0,
 LSRC = AL, RSRC = EA, DEST = AX,

EXT = AH.
If w = 1,
LSRC = AX, RSRC = EA, DEST = DX:AX,
EXT = DX.

Example

Any of the following memory operands could be an indexed address-expression of the correct TYPE. LSRC_BYTE could be ARRAY[SI] if ARRAY were of type BYTE, and RSRC_WORD could be TABLE[BX][DI] if TABLE were of type WORD.

To multiply a byte by a byte:

```
MOV   AL,LSRC_BYTE
MUL   RSRC_BYTE      ;RESULT IN AX
```

To multiply a word by a word:

```
MOV   AX,LSRC_WORD
MUL   RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

To multiply a byte by a word:

```
MOV   AL,MUL_BYTE
CBW   ;CONVERTS BYTE IN AL TO WORD IN AX
MUL   RSRC_WORD
```

NEG
Negate, Form Two's Complement

Purpose

NEG does a subtraction of the operand from zero, adds 1, and returns the result to the operand. This forms the two's complement of the specified operand.

Format

NEG *destination*

Remarks

NEG subtracts the specified operand, *destination*, from all 1's (0FFH for bytes, 0FFFFH for words), adds 1, and stores the result back in *destination*.

Logic

(EA) <- (SRC) - (EA)
(EA) <- (EA) + 1 (affecting flags)

Flags

Affected— AF, CF, OF, PF, SF, ZF

Encoding

1111011w mod011r/m

F6 + w mod011r/m

If w=0, SRC = 0FFH.
If w=1, SRC = 0FFFFH.

Example

If AL contains 13H (00010011),

```
NEG   AL
```

causes AL to contain: -13H or 0EDH (11101101).

If MEM_BYTE contains 0AFH (10101111),

```
NEG   MEM_BYTE
```

causes MEM_BYTE to contain: -0AFH or 51H (01010001).

If SI contains 2FC3H,

```
NEG   SI
```

causes SI to contain: 0D03DH.

NOP
No Operation

Purpose

NOP causes no operation. The next sequential instruction is then run.

Format

NOP

Remarks

The next sequential instruction is run.

Flags

None

Encoding

10010000

90

Example

```
NOP
```

NOT
Logical Not

Purpose

NOT forms the one's complement of (inverts) the operand and returns the result to the operand. Flags are not affected.

Format

NOT *destination*

Remarks

NOT subtracts the specified operand, *destination*, from 0FFH (or 0FFFFH, if a word) and stores the result back into *destination.*

Logic

(EA) <- SRC - (EA)

Flags

None

Encoding

1111011w mod010r/m

F6 + w mod010r/m

If w = 0, SRC = 0FFH.
If w = 1, SRC = 0FFFFH.

Example

If AH contains 13H (00010011), then

```
NOT AH
```

causes AH to contain 0ECH (11101100).

If MEM_BYTE contains 0AFH (10101111), then

```
NOT MEM_BYTE
```

causes MEM_BYTE to contain 50H (01010000).

If DX contains 2FC3H, then

```
NOT DX
```

causes DX to contain 0D03CH.

OR
Logical Inclusive Or

Purpose

OR does the bit-logical inclusive disjunction of the two operands and returns the result to the first operand.

Format

OR *destination,source*

Remarks

Each bit position in the *destination* (leftmost) operand becomes 1, unless it and the corresponding bit position of the *source* (rightmost) operand were both 0. In other words, each bit position of the result has a 1 if either operand had a 1 in that position; if both had a 0, that position of the result has a zero. The carry and overflow flags are both reset.

Logic

(DEST) <- (LSRC) OR (RSRC)
(CF) <- 0
(OF) <- 0

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Memory or Register Operand with Register Operand

Encoding

000010dw modregr/m

08 + dw modregr/m

If d = 1,
LSRC = REG, RSRC = EA, DEST = REG.
If d = 0,
LSRC = EA, RSRC = REG, DEST = EA.

Example

OR register with register:

```
OR   AH,BL  ;RESULT IN AH,  BL UNCHANGED
OR   SI,DX  ;RESULT IN SI,  DX UNCHANGED
OR   CX,DI  ;RESULT IN CX,  DI UNCHANGED
```

OR memory with register:

```
OR   AX,MEM_WORD
OR   CL,MEM_BYTE[SI]
OR   SI,ALPHA[BX][DI]
```

OR register with memory:

```
OR   BETA[BX][DI],AX
OR   MEM_BYTE,DH
OR   GAMMA[DI],BX
```

Immediate Operand to Accumulator

Encoding

0000110w data

0C + w data

If w = 0,
LSRC = AL, RSRC = data, DEST = AL.
If w = 1,
LSRC = AX, RSRC = data, DEST = AX.

Example

OR immediate (byte):

```
OR   AL,11110110B
OR   AL,0F6H
```

OR immediate (word):

```
OR   AX,23F6H
OR   AX,75Q
OR   AX,23F6H
```

Immediate Operand to Memory or Register Operand

Encoding

1000000w mod001r/m data

80 + w mod001r/m data

LSRC = EA, RSRC = data, DEST = EA

Example

OR immediate with register:

```
OR    AH,0F6H
OR    CL,37
OR    DI,23F5H
```

OR immediate with memory:

```
OR    MEM_BYTE,3DH
OR    GAMMA[BX][DI],0FACEH
OR    ALPHA[DI],VAL_EQU_33H
```

Another example:

```
BITMASK  EQU  20H
FLAGS    DB   ?
         .
         .
         .
         OR   FLAGS,BITMASK ;TURN ON FLAG BIT
```

OUT
Output Byte or Word

Purpose

OUT replaces the contents of the designated *port* by the contents of the *accumulator*.

Format

OUT *port,accumulator*

Remarks

OUT transfers a byte (or word) from the AL register (or AX register) to an output *port*. The *port* is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number (0 to 65535) in the DX register, allowing variable access to 64K output ports.

In protected mode, the current privilege level must be less than or equal to the value of IOPL in the flags register.

Logic

(DEST) <- (SRC)

Flags

None

Fixed Port

Encoding

1110011w port

E6 + w port

If w = 0, SRC = AL, DEST = port.
If w = 1, SRC = AX, DEST = port + 1:port.
(0 < port < 255)

Example

```
OUT BYTE_PORT_VAL,AL
;             OUTPUTS A BYTE FROM AL
OUT WORD_PORT_VAL,AX
;             OUTPUTS A WORD FROM AX
OUT 44,AX     ;OUTPUTS A WORD FROM AX
;             THROUGH PORT 44
```

Variable Port

Encoding

1110111w

EE + w

If w = 0, SRC = AL, DEST = (DX).
If w = 1, SRC = AX, DEST = (DX) + 1:(DX).

Example

```
OUT   DX,AL  ;OUTPUTS A BYTE FROM AL
;             THROUGH VARIABLE PORT IN DX
OUT   DX,AX  ;OUTPUTS A WORD FROM AX
;             THROUGH VARIABLE PORT IN AX
```

OUTS/OUTSB/OUTSW (80286)
Output String to Port

Purpose

OUTS transfers a byte or word string element from memory at DS:SI to the *port* numbered by the DX register. The type of the second operand to OUTS determines whether a byte or a word is moved.

Format

OUTS *port,source-string*
or
OUTSB
or
OUTSW

Remarks

The address of the source data is determined only by the contents of SI, not by the second operand to OUTS. You must load the correct index value into SI before running the OUTS, OUTSB, or the OUTSW instructions. Use the operand only to verify segment addressability and to determine the data type. The segment addressability of the operand determines if a segment override byte is produced, or if the default segment register DS is used.

You must address the port through the DX register; the port number cannot be specified as an immediate value.

OUTSB and OUTSW are synonyms for the byte and word OUTS instructions. They are simpler to use, requiring no operands; however, the assembler does not check type or segment.

These instructions advance SI after the transfer is done. If the direction flag is 0 (CLD was run), SI increases; if the direction flag is 1 (STD was run), SI decreases. SI is altered by 1 if a byte was moved, by 2 if a word was moved.

OUTS, OUTSB, and OUTSW can be preceded by the REP prefix for a block input of CX bytes or words. See the REP instruction for the details of this operation.

Note: Not all output devices can handle this speed.

In protected mode, the CPL must be less than or equal to the value of IOPL in the flags register.

You must use the .286C pseudo-op to enable this instruction.

Logic

(DEST) <- (SRC)

Flags

None

Encoding

0110111w

6E + w

If w = 0, SRC = byte
If w = 1, SRC = word

Example

```
OUTS DX,BSTRING  ;OUTPUT BYTE
OUTS DX,WSTRING  ;OUTPUT A WORD
OUTSB            ;OUTPUT BYTE
OUTSW            ;OUTPUT A WORD
```

POP
Pop Word Off Stack to Destination

Purpose

POP replaces the contents of the *destination* by the word at the top of the stack. POP increases the stack pointer by 2.

Format

POP *destination*

Remarks

POP transfers a word operand from the stack element addressed by the SP register to the *destination* operand and then increases SP by 2.

There are three separate types of POP instructions, for different *destinations*. (register, segreg, or memory.)

In protected mode, if the *destination* operand is a segment register, the value popped is a selector. Loading the selector loads the descriptor information associated with that selector into the hidden part of the segment register, and validates both the selector and descriptor information. You may not use POP CS.

Logic

(DEST) <- ((SP) + 1:(SP))
(SP) <- (SP) + 2

Flags

None

Register Operand

Encoding

01011reg

58 + reg

DEST = REG

Example

```
POP   CX ;(01011001)
POP   DX ;(01011010)
```

Segment Register

Encoding

000reg111

07 + reg

If reg ≠ 01 then DEST = REG else undefined operation.

Note: POP CS is not valid.

Example

```
POP   SS;(00010111)
POP   DS;(00011111)
```

Memory or Register Operand

Encoding

10001111 mod000r/m

8F mod000r/m

DEST = EA

Example

```
POP   ALPHA  ;(10001111   00000110)
POP   ALPHA[BX] ;(10001111   10000111)
```

POPA (80286)
Pop All General Registers

Purpose

POPA restores the eight general-purpose registers DI, SI, BP, SP, BX, DX, CX, and AX saved on the stack by PUSHA, except that the SP value is discarded instead of loaded into SP.

Format

POPA

Remarks

POPA reverses a previous PUSHA, restoring the general purpose registers to their values before PUSHA was run.

The .286C pseudo-op is required.

Logic

Pop DI,SI,BP,SP,BX,DX,CX,AX

Flags

None

Encoding

01100001

61

Example

```
POPA
```

POPF
Pop Flags Off Stack

Purpose

POPF transfers specific bits of the stack element addressed by the SP register to the flag registers and then increases SP by 2.

Format

POPF

Remarks

POPF fills the flag registers from the appropriate bit positions of the word at the top of the stack:

overflow flag	= bit 11
direction flag	= bit 10
interrup flag	= bit 9
trap flag	= bit 8
sign flag	= bit 7
zero flag	= bit 6
auxiliary carry flag	= bit 4
parity flag	= bit 2
carry flag	= bit 0

Then the stack pointer is increased by 2.

On the 80286 processor, the entire flag register is popped from the stack. The flags are as follows:

undefined	= bit 15
nested task	= bit 14
I/O privilege level flag	= bits 12 & 13
overflow flag	= bit 11

direction flag	= bit 10
interrupts enabled flag	= bit 9
trap flag	= bit 8
sign flag	= bit 7
zero flag	= bit 6
undefined	= bit 5
auxiliary carry flag	= bit 4
undefined	= bit 3
parity flag	= bit 2
undefined	= bit 1
carry flag	= bit 0

The I/O privilege level will be altered only when executing at privilege level 0. The interrupt enable flag will be altered only when executing at a level at least as privileged as the I/O privilege level. (Real Address mode is equivalent to privilege level 0.) If you execute a POPF instruction with insufficient privilege, there will be no exception nor will the privileged bits be changed.

Logic

Flags <- ((SP) + 1:SP))
(SP)<-(SP) + 2

Flags

Affected— All

Encoding

10011101

9D

Example

POPF

PUSH
Push Word onto Stack

Purpose

PUSH decreases the stack pointer SP by 2 and then transfers a word from the *source* operand to the stack element currently addressed by SP.

Format

PUSH *source*

Remarks

The stack pointer (SP) is decreased by 2. The contents of the specified operand are placed on the top of the stack at the location pointed to by SP. The contents of SP are used as an offset to the base address of the stack in register SS.

The 80286 PUSH SP instruction pushes the value of SP as it existed prior to the instruction. This differs from the 8086/8088 instruction set, which pushes the new (decreased by 2) value.

There are three separate types of PUSH instructions depending on the type of operand supplied.

Logic

(SP) <- (SP) - 2
((SP + 1):(SP))<-(SRC)

Flags

None

Register Operand (Word)

Encoding

01010reg

50 + reg

Example

```
PUSH   AX ;(01010000)
          ;(   50   )
PUSH   SI ;(01010110)
          ;(   56   )
```

Segment Register

Encoding

See example.

Example

```
PUSH   SS ;(00010110)
          ;(   16   )
PUSH   ES ;(00000110)
          ;(   06   )
```

Note: PUSH CS *is* valid.

Memory-or-Register Operand

Encoding

11111111 mod110r/m

FF mod110r/m

Example

```
PUSH   BETA          ;(11111111  00110110)
                     ;(   FF        36   )
PUSH   BETA[BX]      ;(11111111  10110111)
                     ;(   FF        B7   )
PUSH   BETA[BX][DI]  ;(11111111  10110001)
                     ;(   FF        B1   )
```

PUSH (80286)
Push Immediate onto Stack

Purpose

The PUSH immediate instruction decreases the stack pointer SP by 2 and then transfers the *immediate* data to the stack element currently addressed by SP.

Format

PUSH *immediate*

Remarks

The data can be either *immediate* byte or *immediate* word. Byte data is sign-extended to a word before it is pushed onto the stack, because all stack operations are done on word data.

The stack pointer (SP) is decreased by 2. The contents of the specified operand are placed on the top of the stack at the location pointed to by SP. The contents of SP are used as an offset to the base address of the stack in register SS.

The .286C pseudo-op is required.

Logic

(SP) <- (SP)-2
((SP + 1):(SP))<-(SRC)

Flags

None

Encoding

011010s0 data [data if s = 0]

68 + s data [data if s = 0]

Example

```
PUSH  279DH ;(01101000 00100111 10011101)
            ;(   68       27       9D   )
PUSH    9DH ;(01101010 10011101)
            ;(   6A       9D   )
```

PUSHA (80286)
Push All General Registers

Purpose

PUSHA saves the the contents of the eight general-purpose registers on the stack: AX, CX, DX, BX, original SP, BP, SI, and DI.

Format

PUSHA

Remarks

PUSHA decreases the stack pointer SP by 16 to hold the eight word values. Since the registers are pushed onto the stack in the order above, they appear in the 16 new stack bytes in the reverse order.

Note: PUSHA does not save any of the segment registers, the instruction pointer, or the flag register.

The .286C pseudo-op is required.

Logic

Push AX,CX,DX,BX,original SP,BP,SI,DI

Flags

None

Encoding

01100000

60

Example

```
PUSHA
```

PUSHF
Push Flags onto Stack

Purpose

PUSHF decreases the SP register by 2 and transfers all the flag registers into specific bits of the word operand (stack element) addressed by SP.

Format

PUSHF

Remarks

PUSHF decreases the stack pointer by 2; then, the flags replace the appropriate bits of the word at the top of the stack (see POPF).

Logic

(SP) <- (SP) - 2
((SP) + 1:(SP)) <- Flags

Flags

Affected— All

Encoding

10011100

9C

Example

PUSHF

RCL
Rotate Left Through Carry

Purpose

RCL rotates the operand left through the CF flag register by *count* bits.

Format

RCL *destination*,1
or
RCL *destination*,CL

Remarks

The specified *destination* (leftmost) operand is rotated left through the carry flag a number of times (*count*). The second operand is either exactly 1, specified by an immediate value 1, or it is the number held in the CL register.

CF is rotated into bit 0 of the *destination*. The highest-order bit of the *destination* is rotated into CF. All other bits in the *destination* move left one position. The rotation continues until the count is exhausted.

OF is set if the operation changes the high-order (sign) bit of the destination operation on single-bit rotates. That is, if the second operand has a value of 1 and the two highest-order bits of the original *destination* value are unequal (one 0 and one 1), the OF is set to 0. If they are equal, OF is reset. If the second operand does not have a value of 1, OF is undefined and does not have a reliable value.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (tmpcf) <- (CF)
  (CF) <- high-order bit of (EA)
  (EA) <- (EA)*2 + (tmpcf)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
```

then (OF) <- 1
else (OF) <- 0
else (OF) undefined

Flags

Affected— CF, OF

Encoding

110100vw mod010r/m

D0 + vw mod010r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example

Rotate register, count immediate:

```
RCL       AH,1
RCL       BL,1
RCL       CX,1
VAL_ONE   EQU 1
RCL       DX,VAL_ONE
RCL       SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCL   MEM_BYTE,1
RCL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
RCL   DH,CL  ;ROTATES 3 BITS LEFT
RCL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL, 6
RCL   MEM_WORD,CL    ;ROTATES 6 TIMES
RCL   GAMMA_BYTE,CL
RCL   BETA[BX][DI],CL
```

RCL (80286)
Rotate Left Through Carry

Purpose

RCL rotates the operand left through the CF flag register by *count* bits.

Format

RCL *destination*,1
or
RCL *destination*,CL
or
RCL *destination*,*count*

Remarks

RCL rotates the specified *destination* (leftmost) operand left through the carry flag a number of times (1,CL, or *count*). The second operand can be an immediate number from 1 to 31, or it is the number held in the CL register.

CF is rotated into bit 0 of the *destination*. The highest-order bit of the *destination* is rotated into CF. All other bits in the *destination* move left one position. The rotation continues until the count is exhausted.

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the two highest-order bits of the original *destination* value are unequal (one 0 and one 1), the OF is set to 0. If they were equal, OF is reset. If the second operand does not have a value of 1, OF is undefined and does not have a reliable value.

Note: The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

The .286C pseudo-op is required to enable RCL using an immediate operand greater than 1.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (tmpcf) <- (CF)
  (CF) <- high-order bit of (EA)
  (EA) <- (EA)*2 + (tmpcf)
  (temp) <- (temp)-1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF, OF

Encoding

For an immediate value of 1, or CL:

```
110100vw  mod010r/m
D0 + vw   mod010r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

```
1100000w  mod010r/m
C0 +  w   mod010r/m
```

Example

Rotate register, count immediate:

```
RCL      AH,1
RCL      BL,1
RCL      CX,1
RCL      CX,5
VAL_ONE  EQU 1
RCL      DX,VAL_ONE
RCL      SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCL   MEM_BYTE,1
RCL   MEM_BYTE,5
RCL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
RCL   DH,CL  ;ROTATES 3 BITS LEFT
RCL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL, 6
RCL   MEM_WORD,CL   ;ROTATES 6 TIMES
RCL   GAMMA_BYTE,CL
RCL   BETA[BX][DI],CL
```

RCR
Rotate Right Through Carry

Purpose

RCR rotates the operand right through the CF flag register by *count* bits.

Format

RCR *destination*,1
or
RCR *destination*,CL

Remarks

RCR rotates the specified *destination* (leftmost) operand right through the carry flag by either exactly 1, specified by an immediate value of 1, or by the number held in the CL register.

CF is rotated into the high-order bit of the *destination*. The lowest-order bit of the *destination* is rotated into CF. All other bits in the *destination* move right one position. The rotation continues until the count is exhausted.

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the two highest-order bits of the original *destination* value are unequal (one 0 and one 1), the overflow flag is set. If they are equal, OF is reset. If the second operand does not have a value of 1, OF is undefined and has no reliable value.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (tmpcf) <- (CF)
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2
  high-order bit of (EA) <- (tmpcf)
```

```
  (temp) <- (temp) - 1
f COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF,OF

Encoding

```
110100vw  mod011r/m
D0 + vw   mod011r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example

Rotate register, count immediate:

```
RCR       AH,1
RCR       BL,1
RCR       CX,1
VAL_ONE   EQU 1
RCR       DX,VAL_ONE
RCR       SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCR     MEM_BYTE,1
RCR     ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV     CL,3
RCR     DH,CL  ;ROTATES 3 BITS RIGHT
RCR     AX,CL
```

Rotate memory, count in CL:

```
MOV     CL, 6
RCR     MEM_WORD,CL   ;ROTATES 6 TIMES
RCR     GAMMA_BYTE,CL
RCR     BETA[BX][DI],CL
```

RCR (80286)
Rotate Right Through Carry

Purpose

RCR rotates the operand right through the CF flag register by *count* bits.

Format

RCR *destination*,1
or
RCR *destination*,CL
or
RCR *destination,count*

Remarks

RCR rotates the specified *destination* (leftmost) operand right through the carry flag a number of times (1,CL,or *count*). The second operand can be an immediate value from 1 to 31, or it can be the number held in the CL register.

CF is rotated into the high-order bit of the *destination*. RCR rotates the lowest-order bit of the *destination* into CF. All other bits in the *destination* move right one position. The rotation continues until the *count* is exhausted.

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, the second operand has a value of 1 and the two highest-order bits of the original *destination* value are unequal (one 0 and one 1), the overflow flag is set. If they are equal, OF is reset. If the second operand does not have a value of 1, OF is undefined and has no reliable value.

Note: The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

Logic

The .286C pseudo-op is required to enable RCR using an immediate operand greater than 1.

```
(temp) <- COUNT
do while (temp) ≠ 0
  (tmpcf) <- (CF)
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2
  high-order bit of (EA) <- (tmpcf)
  (temp) <- (temp)-1
if COUNT = 1 then
  if high-order bit of (EA) ≠
  next-to-high-order bit of (EA)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF, OF

Encoding

For an immediate value of 1, or CL:

```
110100vw  mod011r/m
D0 + vw   mod011r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

```
1100000w  mod011r/m
C0 +  w   mod011r/m
```

Example

Rotate register, count immediate:

```
RCR       AH,1
RCR       BL,1
RCR       CX,1
RCR       CX,5
VAL_ONE   EQU 1
RCR       DX,VAL_ONE
RCR       SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCR     MEM_BYTE,1
RCR     MEM_BYTE,5
RCR     ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV     CL,3
RCR     DH,CL  ;ROTATES 3 BITS RIGHT
RCR     AX,CL
```

Rotate memory, count in CL:

```
MOV     CL, 6
RCR     MEM_WORD,CL   ;ROTATES 6 TIMES
RCR     GAMMA_BYTE,CL
RCR     BETA[BX][DI],CL
```

REP/REPZ/REPE/REPNE/REPNZ
Repeat String Operation

Purpose

REP causes the primitive string operation that follows to be done repeatedly while (CX) is not zero. For CMPS and SCAS, if after any repetition of the primitive operation the ZF flag differs from the "z" bit of the repeat prefix, the repetition is ended. This prefix can be combined with the segment override and/or LOCK prefixes. With multiple prefixes, interrupts must be disabled, because the return from an interrupt returns control to the interrupted instruction or to at most one prefix byte before that instruction.

Format

```
REP     ;Set z bit to 1
 or
REPZ    ;Set z bit to 1
 or
REPE    ;Set z bit to 1
 or
REPNE   ;Set z bit to 0
 or
REPNZ   ;Set z bit to 0
```

Remarks

The specified string operation is carried out until (CX) is decreased to 0. CX is decreased by 1 after each iteration.

The compare and scan string operations exit the loop when the zero flag is unequal to the value of bit 0 of this instruction byte.

- If the count in CX ran out, CX is zero, and the index register points one past the last byte of the string.
- If the ZF flag ended the repeat, the index register is one byte past the byte that caused the ZF condition, and CX is correspondingly smaller, requiring further adjustments; for example:

```
        REPE    SCASB
        JE      X       ;EXIT BECAUSE UNEQUAL?
        DEC     DI      ;YES, ADJUST INDEX
        INC     CX      ;AND COUNT REG
X:
```

Logic

Do while (CX) ≠ 0
  service pending interrupt (if any)
  run primitive string operation
    in the succeeding byte
  (CX) <- (CX) - 1
  if primitive operation is CMPS, or
    SCAS and (ZF) ≠ z bit of the
    repeat prefix, then exit from while loop

Flags

See individual string operations.

Encoding

1111001z

F2 + z

Example

```
REP   MOVS  DEST,SOURCE  ;SEE MOVS

REPE  CMPS  DEST,SOURCE
;LOOP WILL BE EXITED BEFORE (CX)=0 ONLY
;IF (ZF)=0, ONLY IF THE BYTE AT (DI) IS NOT
;EQUAL TO THE BYTE AT (SI). SEE CMPS.

REPNZ  SCAS  DEST  ;SEE SCAS
;ONLY IF (ZF)=1, (AL) = DEST, WILL
; THIS LOOP BE EXITED BEFORE (CX) = 0
```

Note:

REPNZ (nonzero) = REPNE (not equal)
REPZ (zero) = REPE (equal)

Code one of the forms of these instructions immediately preceding (but separated by at least one blank) the primitive string mnemonic (for example REPNZ SCASW). This specifies that the string operation is to be repeated the number of times determined by CX.

RET
Return from Procedure

Purpose

RET transfers control to the return address pushed by a previous CALL operation and optionally adds an immediate constant, *pop-value*, to the SP register to discard stack parameters. If this is an inter-segment RET (it was assembled under a procedure labeled FAR), RET replaces the IP and the CS using the two words at the top of the stack. Otherwise, RET replaces only the IP, using only one word from the top of the stack.

Format

RET [*pop-value*]

Remarks

RET replaces the instruction pointer by the word at the top of the stack (offset of top is in the stack pointer). SP is increased by 2. For inter-segment returns, RET replaces the CS register by the word now at the top of the stack, and SP is again increased by 2. If an immediate value *pop-value*, was specified on the RET statement, that value is now added to SP.

When using indirect CALLs, you must ensure that the type of CALL matches the type of RETurn in the procedure:

```
CALL WORD PTR [BX]
```

must not call a FAR procedure. And:

```
CALL DWORD PTR [BX]
```

must not call a NEAR procedure.

In protected mode, an inter-segment return may cause a privilege level change, but only to a lesser privileged level. If such a change is made, the lesser privileged stack VADW is assumed to be on the stack above the return CS by some fixed displacement. 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. If an

inter-segment return-and-add format is used, then the VADW for the old stack is found above the arguments on the new stack, and arg is removed from both the old and new stacks.

See Chapter 6, "80286/80386-Based Personal Computers," in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

```
(IP) <- ((SP)+1:(SP))
(SP) <- (SP) + 2
if inter-segment then
  (CS) <- ((SP)+1:(SP))
  (SP) <- (SP) + 2
if Add immediate to Stack Pointer
  then (SP) <- (SP) + data.
```

For protected mode:

```
if no immediate constant, Data = 0
if intra-segment then
 (IP) <- ((SP)+ 1:(SP))
 (SP) <- (SP) + DATA
else
 if inter-segment then
   if return selector RPL = CPL then
     (CS) <- ((SP) + 3:(SP) + 2)
     (IP) <- ((SP) + 1:SP)
     (SP) <- (SP) + DATA
   else
     (IP) <- ((SP) + 1:(SP))
     (CS) <- ((SP) + 3:(SP) + 2)
     (SP) <- ((SP) + 4:(SP) + 5) + DATA
     (SS) <- ((SP) + 6:(SP) + 7) + DATA
```

Flags

None.

Intra-Segment

Encoding

11000011

C3

Example

```
RET
```

Intra-Segment and Add Immediate to Stack Pointer

Encoding

11000010 data-low data-high

C2 data-low data-high

Example

```
RET   4
RET   12
```

Note: These values cause parameter words 2 and 6 earlier stored on the stack to be discarded. Since most stack operations are on words, these values are usually even numbers (2 bytes per word).

Inter-Segment and Add Immediate to Stack Pointer

Encoding

11001010 data-low data-high

CA data-low data-high

Example

```
RET   2  ;INTER-SEGMENT RETURNS
RET   8  ;RESTORE IP FIRST, THEN CS
```

Inter-Segment

Encoding

11001011 (CBH)

CB (CBH)

Example

```
RET
```

ROL
Rotate Left

Purpose

ROL rotates the operand left by *count* bits.

Format

ROL *destination*,1
or
ROL *destination*,CL

Remarks

ROL rotates the specified *destination* (leftmost) operand left by one or by the value held in the CL register.

The high-order bit of the *destination* operand replaces the carry flag, whose original value is lost. All other bits in the *destination* move left one position. The vacated bit-position 0 is filled by the new CF (the old high-order bit).

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does equal that high-order bit, OF becomes 0. However, if the second operand does not have a value of 1, OF is not defined and has no reliable value.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- high-order bit of (EA)
  (EA) <- (EA) * 2 + (CF)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) <- 1
```

else (OF) <- 0
else (OF) undefined

Flags

Affected— CF,OF

Encoding

110100vw mod000r/m

D0 + vw mod000r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example

Rotate register, count immediate:

```
ROL       AH,1
ROL       BL,1
ROL       CX,1
VAL_ONE   EQU  1
ROL       DX,VAL_ONE
ROL       DI,VAL_ONE
```

Rotate memory, count immediate:

```
ROL   MEM_BYTE,1
ROL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROL   DH,CL  ;ROTATES 3 BITS LEFT
ROL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROL   MEM_WORD,CL   ;ROTATES 6 TIMES
ROL   GAMMA_BYTE,CL
ROL   BETA[BX][DI],CL
```

ROL (80286)
Rotate Left

Purpose

ROL rotates the specified destination (leftmost) operand left *count* times.

Format

ROL *destination*,1
or
ROL *destination*,CL
or
ROL *destination,count*

Remarks

Count can be an immediate value from 1 to 31, or it is the number held in the CL register.

If the second operand is a 1, then the high-order bit of the destination replaces the carry flag, whose original value is lost. All other bits in the destination move left one position. The vacated bit-position 0 is filled by the new CF (the old high-order bit). as

OF is set if the operator changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does equal that high-order bit, OF becomes 0. However, if the second operand does not have a value of 1, OF is not defined and has no reliable value.

Note: The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- high-order bit of (EA)
  (EA) <- (EA) * 2 + (CF)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF,OF

Encoding

For an immediate value of 1, or CL:

```
110100vw  mod000r/m
D0 + vw   mod000r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

```
1100000w  mod000r/m
C0 +  w   mod000r/m
```

Example

Rotate register, count immediate:

```
ROL       AH,1
ROL       BL,1
ROL       CX,1
ROL       CX,5
VAL_ONE   EQU  1
ROL       DX,VAL_ONE
ROL       DI,VAL_ONE
```

Rotate memory, count immediate:

```
ROL   MEM_BYTE,1
ROL   MEM_BYTE,5
ROL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROL   DH,CL  ;ROTATES 3 BITS LEFT
ROL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROL   MEM_WORD,CL   ;ROTATES 6 TIMES
ROL   GAMMA_BYTE,CL
ROL   BETA[BX][DI],CL
```

ROR
Rotate Right

Purpose

ROR rotates the operand right by *count* bits.

Format

ROR *destination*,1
or
ROR *destination*,CL

Remarks

ROR rotates the specified *destination* (leftmost) operand right by 1 or CL times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move right one position. The vacated high-order position is filled by the new CF (the old value of position 0).

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the new high-order value is not equal to the old high-order value, the overflow flag is set; if they are equal, (OF) = 0. However, if the second operand does not have a value of 1, OF is undefined and has no reliable value.

Logic

```
(temp) <- COUNT
DO WHILE (temp) ≠ 0
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2
  high-order bit of (EA) <- (CF)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
```

else (OF) <- 0
else (OF) undefined

Flags

Affected— CF,OF

Encoding

110100vw mod001r/m

D0 + vw mod001r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example

Rotate register, count immediate:

```
ROR       AH,1
ROR       BL,1
ROR       CX,1
VAL_ONE   EQU  1
ROR       DX,VAL_ONE
ROR       SI,VAL_ONE
```

Rotate memory, count immediate:

```
ROR   MEM_BYTE,1
ROR   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROR   DH,CL  ;ROTATES 3 BITS RIGHT
ROR   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROR   MEM_WORD,CL   ;ROTATES 6 TIMES
ROR   GAMMA_BYTE,CL
ROR   BETA[BX][DI],CL
```

ROR (80286)
Rotate Right

Purpose

ROR rotates the specified *destination* (leftmost) operand right *count* times.

Format

ROR *destination*,1
or
ROR *destination*,CL
or
ROR *destination,count*

Remarks

Count can be an immediate value from 1 to 31, or it is the number held in the CL register.

If the second operand is a 1, then the low-order bit of the *destination* replaces the carry flag, whose original value is lost. All other bits in the *destination* move right one position. The vacated high-order position is filled by the new CF (the old value of position 0).

OF is set if the operation changes the high-order (sign) bit of the destination operand on single-bit rotates. That is, if the second operand has a value of 1 and the new high-order value is not equal to the old high-order value, the overflow flag is set; if they are equal, (OF) <- 0. However, if *count* was not 1, OF is undefined and has no reliable value.

Note: The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

The .286C pseudo-op is required to enable ROR using an immediate operand greater than 1.

Logic

```
(temp) <- COUNT
DO WHILE (temp) ≠ 0
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2
  high-order bit of (EA) <- (CF)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF, OF

Encoding

For an immediate value of 1 or CL:

```
110100vw  mod001r/m
D0 + vw   mod001r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

```
1100000w  mod001r/m
C0 + w    mod001r/m
```

Example

Rotate register, count immediate:

```
ROR       AH,1
ROR       BL,1
ROR       CX,1
ROR       CX,5
VAL_ONE   EQU  1
ROR       DX,VAL_ONE
ROR       SI,VAL_ONE
```

Rotate memory, count immediate:

```
ROR   MEM_BYTE,1
ROR   MEM_BYTE,5
ROR   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROR   DH,CL  ;ROTATES 3 BITS RIGHT
ROR   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROR   MEM_WORD,CL   ;ROTATES 6 TIMES
ROR   GAMMA_BYTE,CL
ROR   BETA[BX][DI],CL
```

SAHF
Store AH in Flags

Purpose

SAHF transfers specific bits of the AH register to the flag registers SF, ZF, AF, PF, and CF. The bits of AH indicated by "X" in the operation are ignored.

Format

SAHF

Remarks

SAHF replaces the five flags shown by specified bits from AH, the high-order byte of the accumulator:

(SF) = bit 7
(ZF) = bit 6
(AF) = bit 4
(PF) = bit 2
(CF) = bit 0.

Logic

(SF):(ZF):X:(AF):X:(PF):X:(CF)<-(AH)

Flags

Affected— AF, CF, PF, SF, ZF

Encoding

10011110

9E

Example

```
SAHF
```

SAL/SHL
Shift Arithmetic Left/Logical Left

Purpose

SAL and SHL shift the operand left by 1 or CL bits, shifting in low-order zero bits.

Format

SAL *destination*,1
or
SAL *destination*,CL

SHL *destination*,1
or
SHL *destination*,CL

Remarks

SAL/SHL shift the specified *destination* (leftmost) operand left by 1 or by the value contained in CL. The high-order bit of the *destination* operand replaces the carry flag, whose original value is lost. All other bits in the *destination* move left one position. The vacated low-order bit-position is filled by 0.

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts, however, the value of OF is always defined.

.ogic

```
temp) <- (COUNT)
lo while (temp) ≠ 0
 (CF) <- high-order bit of (EA)
 (EA) <- (EA)*2
 (temp) <- (temp) - 1
f COUNT = 1 then
 if high-order bit of (EA) ≠ (CF)
   then (OF) <- 1
 else (OF) <- 0
:lse (OF) undefined
```

Flags

```
Affected— CF,OF,PF,SF,ZF
Jndefined— AF
```

Encoding

```
110100vw  mod100r/m

D0 + vw   mod100r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example

Shift register, count immediate:

```
SHL       AH,1
SHL       BL,1
SHL       CX,1
VAL_ONE   EQU  1
SHL       SI,VAL_ONE
SHL       SI,VAL_ONE
```

Shift memory, count immediate:

```
SHL  MEM_BYTE,1
SHL  ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV  CL,3
SHL  DH,CL ;SHIFT 3 BITS LEFT
SHL  AX,CL
```

Shift register, count in CL:

```
MOV  CL,6
SHL  MEM_WORD,CL ;SHIFT 6 TIMES
SHL  GAMMA_BYTE,CL
SHL  BETA[BX][DI],CL
```

SAL/SHL (80286)
Shift Arithmetic Left/Shift Logical Left

Purpose

SAL and SHL shift the operand left by *count* bits, shifting in low-order zero bits.

Format

SAL *destination*,1
or
SAL *destination*,CL
or
SAL *destination*,*count*

SHL *destination*,1
or
SHL *destination*,CL
or
SHL *destination*,*count*

Remarks

SAL/SHL shift the specified *destination* (leftmost) operand left 1, CL, or *count*

The high-order bit of the *destination* replaces the carry flag, whose original value is lost. All other bits in the *destination* move left one position. The vacated low-order bit-position is filled by 0.

The second operand can be an immediate value from 1 to 31, or the value held in the CL register.

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts, however, the value of OF is always undefined.

Note: The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

The .286C pseudo-op is required to enable SAL and SHL using an immediate operand greater than 1.

Logic

```
(temp) <- (COUNT)
do while (temp) ≠ 0
  (CF) <- high-order bit of (EA)
  (EA) <- (EA)*2
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

For an immediate value of 1, or CL:
110100vw mod100r/m
D0 + vw mod100r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

1100000w mod100r/m
C0 + w mod100r/m

Example

Shift register, count immediate:

```
SHL       AH,1
SHL       BL,1
SHL       CX,1
SHL       CX,5
VAL_ONE   EQU  1
SHL       SI,VAL_ONE
SHL       SI,VAL_ONE
```

Shift memory, count immediate:

```
SHL  MEM_BYTE,1
SHL  MEM_BYTE,5
SHL  ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV  CL,3
SHL  DH,CL ;SHIFT 3 BITS LEFT
SHL  AX,CL
```

Shift register, count in CL:

```
MOV  CL,6
SHL  MEM_WORD,CL ;SHIFT 6 TIMES
SHL  GAMMA_BYTE,CL
SHL  BETA[BX][DI],CL
```

SAR
Shift Arithmetic Right

Purpose

SAR (shift arithmetic right) shifts the *destination* operand right by 1 or CL bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

Format

SAR *destination,1*
or
SAR *destination,CL*

Remarks

SAR shifts the specified *destination* (leftmost) operand right 1 or CL times. The low-order bit of the *destination* operand replaces the carry flag, whose original value is lost. All other bits in the *destination* move right one position. The vacated high-order position retains its old value. (If the original high-order bit value was 0, zeroes are shifted in). If that value was 1, ones are shifted in.

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts, however, the value of OF is always undefined.

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

110100vw mod111r/m

D0 + vw mod111r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2, where / is equivalent to signed
     division, rounding down
  (temp) <- (temp) - 1
if COUNT = 1 then
if high-order bit of (EA) ≠
     next-to-high-order bit of (EA)
   then (OF) <- 1
   else (OF) <- 0
else (OF) <- undefined
```

Example

Shift register, count immediate:

```
SAR      AH,1
SAR      BL,1
SAR      CX,1
VAL_ONE  EQU 1
SAR      DX,VAL_ONE
SAR      SI,VAL_ONE
```

Shift memory, count immediate:

```
SAR   MEM_BYTE,1
SAR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SAR   DH,CL  ;SHIFTS 3 BITS RIGHT
SAR   AX,CL
```

SAR (80286)
Shift Arithmetic Right

Purpose

SAR (shift arithmetic right) shifts the *destination* operand right by *count* bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

Format

SAR *destination*,1
or
SAR *destination*,CL
or
SAR *destination*,*count*

Remarks

SAR shifts the specified *destination* (leftmost) operand right 1, CL, or *count* times. The second operand can be an immediate value from 1 to 31, or it can be the number held in the CL register.

The low-order bit of the *destination* replaces the carry flag, whose original value is lost. All other bits in the *destination* move right one position. The vacated high-order position retains its old value. (If the original high-order bit value was 0, zeroes are shifted in; if that value was 1, ones are shifted in.)

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts, however, the value of OF is always undefined.

Note: The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

The .286C pseudo-op is required to enable SAR using an immediate operand greater than 1.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
    (CF) <- low-order bit of (EA)
    (EA) <- (EA)/2, where / is equivalent to signed
      division, rounding down
    (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
  else (OF) <- 0
else (OF) <- undefined
```

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

For an immediate value of 1 or CL:

```
110100vw  mod111r/m

D0 + vw   mod111r/m
```

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

```
1100000w  mod111r/m
C0 +  w   mod111r/m
```

Example

Shift register, count immediate:

```
SAR      AH,1
SAR      BL,1
SAR      CX,1
SAR      CX,5
VAL_ONE  EQU 1
SAR      DX,VAL_ONE
SAR      SI,VAL_ONE
```

Shift memory, count immediate:

```
SAR   MEM_BYTE,1
SAR   MEM_BYTE,5
SAR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SAR   DH,CL  ;SHIFTS 3 BITS RIGHT
SAR   AX,CL
```

SBB
Subtract with Borrow

Purpose

SBB does a subtraction of the two operands, subtracts one if the CF flag is set, and returns the result to one of the operands.

Format

SBB *destination,source*

Remarks

SBB subtracts the *source* (rightmost) operand from the *destination* (leftmost). If the carry flag was set, SBB subtracts 1 from the above result. The result replaces the original *destination* operand.

Logic

If (CF) = 1, (DEST) <- (LSRC) - (RSRC) - 1
Else (DEST) <- (LSRC) - (RSRC)

Flags

Affected—AF,CF,OF,PF,SF,ZF

Memory or Register Operand and Register Operand

Encoding

000110dw modregr/m

18 + dw modregr/m

If d = 1, LSRC = REG, RSRC = EA, DEST = REG.
If d = 0, LSRC = EA, RSRC = REG, DEST = EA.

Example

Subtract register from register:

```
SBB   AX,BX
SBB   CH,DL
```

Subtract memory from register:

```
SBB   DX,MEM_WORD
SBB   DI,ALPHA[SI]
SBB   DL,MEM_BYTE[DI]
```

Subtract register from memory:

```
SBB   MEM_WORD,AX
SBB   MEM_BYTE[DI],BX
SBB   GAMMA[BX][DI],SI
```

Immediate Operand from Accumulator

Encoding

0001110w data data*

1C + w data data*

If w = 0, then LSRC = data, DEST = AL.
If w = 1, then LSRC = data, DEST = AX.

*Present only if w=1.

Example

Subtract immediate (byte):

```
SBB         AL,4
VAL_SIXTY   EQU 60
SBB         AL,VAL_SIXTY
```

Subtract immediate (word):

```
SBB    AX,660
SBB    AX,VAL_SIXTY*6
```

Immediate Operand from Memory or Register Operand

Encoding

100000sw mod011r/m data

80 + sw mod011r/m data

LSRC = EA, RSRC = data, DEST = EA

If an immediate-data byte is being subtracted from a register-or-memory word, the byte is sign-extended to 16 bits before the subtraction. For this situation, the instruction byte is 83H (the s and w bits are both set).

Example

Subtract immediate from register:

```
SBB   BX,2001
SBB   CL,VAL_SIXTY
SBB   SI,VAL_SIXTY*9
```

Subtract immediate from memory:

```
SBB   MEM_BYTE,12
SBB   MEM_BYTE[DI],VAL_SIXTY
SBB   MEM_WORD[BX],79
SBB   GAMMA[DI][BX],1984
```

SCAS/SCASB/SCASW
Scan Byte or Word String

Purpose

SCAS subtracts the *destination* byte (or word) operand addressed by ES:DI 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 a string. See the REP instruction in this chapter.

Format

SCAS *dest-string*
or
SCASB
or
SCASW

Remarks

The *destination-string* element specified by offset DI in the extra segment is subtracted from the value in the accumulator, but the operation affects flags only. The destination index is then increased (if the direction flag is zero) or decreased (if (DF) = 1) by 1 for byte strings or 2 for words. (See the CLD and STD instructions.)

Logic

(LSRC) - (RSRC)
if (DF) = 0, then
  (DI) <- (DI) + DELTA
else (DI) <- (DI) - DELTA

Flags

Affected— AF,CF,OF,PF,SF,ZF

Encoding

1010111w

AE + w

If w=0, LSRC = AL, RSRC = (DI), DELTA = 1. If w=1, LRSC = AX, RSRC = (D) + 1:(DI), DELTA = 2.

Example

```
CLD   ;CLEARS DF, CAUSES DI INCREASING
MOV   DI,OFFSET DEST_BYTE_STRING
MOV   AL,'M'
SCAS  DEST_BYTE_STRING
    or
SCAS  ES:BYTE PTR[DI]
    or
SCASB

STD   ;SETS DF, CAUSES DI DECREASING
MOV   DI,OFFSET WORD_STRING
MOV   AX,'MD'
SCAS WORD_STRING
```

The operand named in the SCAS instruction is used only by the assembler to verify type and accessibility using current segment register contents. The operation of this instruction uses DI to point to the location to be scanned, without using the operand named in the source line.

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.
2. Load DI with the offset of the *destination-string*.
3. Each can be coded with or without symbolic memory operand.
    - If symbolic operand is coded, the assembler can check the addressability of it for you.
    - References that use hardware defaults should be coded using the operand-less forms (SCASB and SCASW), to avoid the additional pointer information.
    - Do not use [BX], [SI], or [BP] addressing modes with the string instructions.

4. If the instruction mnemonic is coded without operands, the segment registers 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.

SGDT (80286P)
Store Global Descriptor Table

Purpose

SGDT stores the Global Descriptor Table Register in 6-bytes of memory addressed by the destination operand.

Format

SGDT *destination*

Remarks

The effective address of the destination operand must be in memory. The LIMIT field of GDTR is the first word; the next 3 bytes are the BASE field of GDTR, and the sixth byte is undefined.

The SIDT and SGDT 80286 protected mode instructions operate on 6-byte quantities. Since the assembler has no way to define 6-byte data item, it uses the first 6 bytes of the next largest type of data item, which is a QWORD. There are several ways of specifying the source operand for these instructions:

- a data item defined with DQ
- a symbol defined with EXTRN X:QWORD
- a symbol defined using LABEL QWORD
- use the QWORD PTR override.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

((addr) + 5) <- undefined

((addr) + 4:(addr)) <- GDTR.base
((addr) + 1:(addr)) <- GDTR.limit

Flags

None

Encoding

00001111 00000001 mod000r/m
0F 01 mod000r/m

Example

```
      .286P
      EXTRN   VAR1:QWORD
DATA  SEGMENT
      ASSUME  DS:DATA
VAR2  DQ      ?
VAR3  LABEL   QWORD
VAR4  DB      6 DUP(?)
DATA  ENDS
CODE  SEGMENT
      ASSUME  CS:CODE
 .
 .
 .
      SIDT    VAR1           ;external data item
      SGDT    VAR2           ;data field of type QWORD
      SIDT    VAR3           ;label of type QWORD
      SGDT    QWORD PTR VAR4 ;explicit override
```

SHR
Shift Logical Right

Purpose

SHR shifts the operand right, shifting in high-order zero bits.

Format

SHR *destination*, 1
or
SHR *destination*, CL

Remarks

SHR shifts the specified *destination* (leftmost) operand right 1 or CL times. The low-order bit of the destination operand replaces the carry flag, whose original value is lost. All other bits in the *destination* move right one position. The vacated high-order position is filled by 0.

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts however, the value of OF is always undefined.

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

110100vw mod101r/m

D0 + vw mod101r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2, where / is equivalent
    to unsigned division
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
   else (OF) <- 0
else (OF) <- undefined
```

Example

Shift register, count immediate:

```
SHR       AH,1
SHR       BL,1
SHR       CX,1
VAL_ONE   EQU 1
SHR       DX,VAL_ONE
SHR       SI,VAL_ONE
```

Shift memory, count immediate:

```
SHR   MEM_BYTE,1
SHR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SHR   DH,CL  ;SHIFTS 3 BITS RIGHT
SHR   AX,CL
```

SHR (80286)
Shift Logical Right

Purpose

SHR shifts the operand right, shifting in high-order zero bits.

Format

SHR *destination*,1
or
SHR *destination*,CL
or
SHR *destination*,*count*

Remarks

SHR shifts the specified *destination* (leftmost) operand right 1, CL, or *count* times. The second operand can be an immediate value from 1 to 31, or it is the number held in the CL register. The low-order bit of the *destination* replaces the carry flag, whose original value is lost. All other bits in the *destination* move right one position. The vacated high-order position is filled by 0.

In a single-bit shift, OF is set if the value of the high-order (sign) bit of the destination operand was changed by the operation. If the sign bit did not change then OF is cleared. Following multiple bit shifts, however, the value of OF is always undefined.

Note: The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

The .286C pseudo-op is required to enable SHR using an immediate operand greater than 1.

Logic

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- low-order bit of (EA)
  (EA) <- (EA)/2, where / is equivalent to signed
    division, rounding down
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) <- 1
  else (OF) <- 0
else (OF) <- undefined
```

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

For an immediate value of 1 or CL:

110100vw mod101r/m

D0 + vw mod101r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding

For an immediate value of 2-31:

1100000w mod101r/m
C0 + w mod101r/m

Example

Shift register, count immediate:

```
SHR       AH,1
SHR       BL,1
SHR       CX,1
SHR       CX,5
VAL_ONE   EQU 1
SHR       DX,VAL_ONE
SHR       SI,VAL_ONE
```

Shift memory, count immediate:

```
SHR   MEM_BYTE,1
SHR   MEM_BYTE,5
SHR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SHR   DH,CL  ;SHIFTS 3 BITS RIGHT
SHR   AX,CL
```

SIDT (80286P)
Store Interrupt Descriptor Table

Purpose

SIDT stores the Interrupt Descriptor Table Register in 6-bytes of memory addressed by the *destination* operand.

Format

SIDT *destination*

Remarks

The effective address of the *destination* operand must be in memory. The LIMIT field of the IDTR is the first word; the next 3 bytes are the BASE field of the IDTR; and the sixth byte is not defined.

The LIDT and LGDT 80286 protected mode instructions operate on 6-byte quantities. Since the assembler has no way to define a 6-byte data item, it uses the first 6 bytes of the next largest type of data item, which is a QWORD. There are several ways of specifying the source operand for these instructions:

- a data item defined with DO
- a symbol defined with EXTRN X: QWORD
- a symbol defined using LABEL QWORD
- use the QWORD PTR override.

See Chapter 6, "80286/80386-based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

(EA + 5) <- UNDEFINED
(EA + 4:EA + 2) <- RIDT.BASE
(EA + 1:EA) <- RIDT.LIMIT

Flags

None

Encoding

00001111 00000001 mod001r/m
OF 01 mod001r/m

Example

See the example under the SGDT description in this chapter.

SLDT (80286P)
Store Local Descriptor Table

Purpose

SLDT stores the Local Descriptor Table Register in the 2-byte *destination* operand.

Format

SLDT *destination*

Remarks

The *destination* operand can be either a register or a memory location.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

OPERAND <- LDTR

Flags

None

Encoding

```
00001111  00000000  mod000r/m
  0F        00      mod000r/m
```

Example

```
SLDT BX
   or
SLDT REP
```

SMSW (80286P)
Store Machine Status Word

Purpose

SMSW stores the Machine Status Word (MSW) in the *destination* operand.

Format

SMSW *destination*

Remarks

The *destination* operand can be either a register or a memory location.

You must use the .286P pseudo-op to enable this instruction.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

Logic

OPERAND <- MSW

Flags

None

Encoding

```
00001111  00000001  mod100r/m
  0F        01      mod100r/m
```

Example

```
SMSW BX
   or
SMSW THING
```

STC
Set Carry Flag

Purpose

STC sets the CF flag.

Format

STC

Remarks

STC sets the carry flag to 1.

Note: See CLC for the opposite function.

Logic

(CF) <- 1

Flags

Affected— CF

Encoding

11111001

F9

Example

```
STC
```

STD
Set Direction Flag

Purpose

STD sets the DF flag causing the string operations to automatically decrease the operand indexes.

Format

STD

Remarks

STD sets the direction flag to 1.

Note: See CLD for the opposite function.

Logic

(DF) <- 1

Flags

Affected— DF

Encoding

11111101

FD

Example

```
STD   ;CAUSES DECREASING OF DI (AND SI)
;      IN STRING OPERATIONS
```

STI
Set Interrupt Flag (Enable)

Purpose

STI sets the IF flag, enabling maskable external interrupts after the performing of the next instruction.

Format

STI

Remarks

STI sets the interrupt flag to 1.

Note: See CLI for the opposite function.

In protected mode, STI must have a minimum of IOPL.

Logic

(IF) <- 1

Flags

Affected— IF

Encoding

11111011

FB

Example

```
STI    ;ENABLES INTERRUPTS
```

STOS/STOSB/STOSW
Store Byte or Word String

Purpose

STOS transfers a byte (or word) operand from AL (or AX) to the *destination* operand addressed by DI and adjusts the DI register by DELTA. As a repeated operation (see REP), this provides for filling a string with a given value. The operand named in the STOS instruction is used only by the assembler to verify type and accessibility using current segment register contents. The actual operation of the instruction uses only DI to point to the location being stored into.

Format

STOS *destination-string*
or
STOSB
or
STOSW

Remarks

The byte (or word) in AL (or AX) replaces the contents of the byte (or word) pointed to by DI in the extra segment. The instruction then increases DI if the direction flag is zero; the instruction decreases DI if DF = 1. The change is 1 for bytes, 2 for words.

Logic

```
(DEST) <- (SRC)
if (DF) = 0 then
  (DI) <- (DI) + DELTA
else (DI) <- (DI) - DELTA
```

Flags

None

Encoding

1010101w

AA + w

If w=0, SRC = AL, DEST = 1.
If w=1, SRC = AX,
DEST = (DI) + 1:(DI), DELTA = 2.

Example

Store byte:

```
MOV   DI,OFFSET  BYTE_DEST_STRING
STOS  BYTE_DEST_STRING
```

Store word:

```
MOV   DI,OFFSET WORD_DEST
STOS  WORD_DEST
```

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.
2. Load DI with the offset of the *destination-string*.
3. Each can be coded with or without symbolic memory operands.
   - If symbolic operands are coded, the assembler can check whether you can address them.
   - References that use hardware defaults should be coded using the operand-less forms (STOSB and STOSW) to avoid the additional pointer information.
   - Do not use [BX] or [BP] addressing modes with the string instructions.
4. If the instruction mnemonic is coded without operands, SI defaults to an offset in the segment addressed by DS.

STR (80286P)
Store Task Register

Purpose

STR stores the Task Register in the *destination* operand.

Format

STR *destination*

Remarks

The *destination* operand can be either a register or a memory location.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for more information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

OPERAND <- TR

Flags

None

Encoding

```
00001111  00000000  mod001r/m
  0F        00      mod001r/m
```

Example

```
STR BX
   or
STR PLACE
```

SUB
Subtract

Purpose

SUB does a subtraction of the *source* operand from the *destination*. The result goes to the *destination* operand.

Format

SUB *destination,source*

Remarks

SUB subtracts the *source* (rightmost) operand from the *destination* (leftmost) operand and stores the result in the *destination*.

Logic

(DEST) <- (LSRC) - (RSRC)

Flags

Affected— AF,CF,OF,PF,SF,ZF

Memory or Register Operand and Register Operand

Encoding

001010dw modregr/m

28 + dw modregr/m

If d = 1,
LSRC = REG, RSRC = EA, DEST = REG.
If d = 0,
LSRC = EA, RSRC = REG, DEST = EA.

Example

Subtract register from register:

```
SUB   AX,BX
SUB   CH,DL
```

Subtract memory from register:

```
SUB   DX,MEM_WORD
SUB   DI,ALPHA[SI]
SUB   BL,MEM_BYTE[DI]
```

Subtract register from memory:

```
SUB   MEM_WORD,AX
SUB   MEM_BYTE[DI],BL
SUB   GAMMA[BX][DI],SI
```

Immediate Operand from Accumulator

Encoding

0010110w data

2C + w data

If w = 0, LSRC = AL, RSRC = data, DEST = AL.
If w = 1, LSRC = AX, RSRC = data, DEST = AX.

Example

Subtract immediate from register (byte):

```
SUB         AL,4
VAL_SIXTY   EQU 60
SUB         AL,VAL_SIXTY
```

Subtract immediate from register (word):

```
SUB   AX,660
SUB   AX,VAL_SIXTY*6
SUB   AX,6606
```

If an immediate-data byte is being subtracted from a register-or-memory word, the byte is signed extended to 16 bits before the subtraction. For this situation, the instruction is 83H (the s and w bits are both set).

Immediate Operand from Memory or Register Operand

Encoding

100000sw mod101r/m data

80 + sw mod101r/m data

LSRC = EA, RSRC = data, DEST = EA

Example

Subtract immediate from register:

```
SUB   BX,2001
SUBM  CL,VAL_SIXTY
SUB   SI,VAL_SIXTY*9
```

Subtract immediate from memory:

```
SUB   MEM_BYTE,12
SUB   MEM_BYTE[DI],VAL_SIXTY
SUB   MEM_WORD[BX],79
SUB   GAMMA[DI][BX],1984
```

TEST
Test (Logical Compare)

Purpose

TEST does the bit-to-bit logical conjunction of the two operands, causing the flags to be affected, but does not return the result.

Format

TEST *destination,source*

Remarks

TEST performs the AND operation on the operands to affect the flags, but neither operand is changed. The carry and overflow flags are reset.

The *source* (rightmost) operand must be of the same type (byte or word) as the *destination* operand. The only exception for TEST is testing an immediate-data byte with a memory word.

Logic

(LSRC) & (RSRC)
(CF) <- 0
(OF) <- 0

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Memory or Register Operand with Register Operand

Encoding

1000010w modregr/m

84 + w modregr/m

LSRC = REG, RSRC = EA

Example

Register with register:

```
TEST   AX,DX
TEST   SI,BP
TEST   BH,CL
```

Register with memory:

```
TEST   MEM_WORD,SI
TEST   MEM_BYTE,CH
TEST   ALPHA[DI],DX
TEST   BETA[BX][SI],CX
```

Memory with register:

```
TEST   DI,MEM_WORD
TEST   CH,MEM_BYTE
TEST   AX,GAMMA[BP][SI]
```

Immediate Operand with Accumulator

Encoding

1010100w data

A8 + w data

If w = 0, LSRC = AL, RSRC = data.
If w = 1, LSRC = AX, RSRC = data.

Example

```
TEST   AL,6
TEST   AL,IMM_VALUE_DRIVE
TEST   AX,999
```

Immediate Operand with Memory or Register Operand

Encoding

1111011w mod000r/m data

F6 + w mod000r/m data

LSRC = EA, RSRC = data

Example

Immediate with register:

```
TEST   BH,7
TEST   CL,19_IMM_BYTE
TEST   DX,IMM_DATA_WORD
TEST   SI,798
```

Immediate with memory:

```
TEST   MEM_WORD,IMM_DATA_BYTE
TEST   GAMMA[BX],IMM_BYTE
TEST   [BP][DI],6ACEH
```

VERR (80286P)
Verify Read Access

Purpose

VERR lets you verify that the segment, marked by the selector contained in the operand, is readable and can be read from the current privilege level without your having to load the selector into a segment register and risk a fault.

Format

VERR *source*

Remarks

The operand can be either a register or a location of a word in memory. The verification is the same as if the selector were loaded into DS/ES, except that the zero flag receives the result of the verification instead of a possible fault occurring. If the verification passes, the zero flag is set to 1; if the verification fails, the zero flag is set to 0.

To set the zero flag, these conditions must be true:

- The selector must mark a descriptor within the bounds of the table (GDT or LDT).
- The selector must mark the descriptor of a code or data segment.
- The segment must be readable.
- If the segment is a readable and conforming code segment, its Descriptor Privilege Level can be any value; otherwise, the DPL must be greater than or equal to both the Current Privilege Level (CPL) and the selector's Requested Privilege Level (RPL).

The only faults that can occur are those produced by incorrectly addressing the memory operand that contains the selector. The selector is not loaded into a segment register and no faults attributed to the selector operand are produced.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

perform read-validity check on selector in register
or memory operand
if check succeeds, then ZF <- 1
else ZF <- 0

Flags

Affected— ZF

Encoding

00001111 00000000 mod100r/m
0F 00 mod100r/m

Example

```
VERR BX
   or
VERR MEMWORD
```

VERW (80286P)
Verify Write Access

Purpose

VERW lets you verify that the segment, shown by the selector contained in the operand, is writable and can be reached from the current privilege level without loading the selector into a segment register and risking a fault.

Format

VERW *source*

Remarks

Source can either be a register or a location of a word in memory. The verification that is performed is the same as if the selector was loaded into DS/ES, except that the zero flag receives the result of the verification instead of a possible fault occurring. If the verification passes, the zero flag is set to 1; if the verification fails, the zero flag is set to 0.

For you to set the zero flag, these conditions must be true:

- The selector must mark a descriptor within the bounds of the table (GDT or LDT).
- The selector must mark the descriptor of a code or data segment.
- The segment must be writable.
- The DPL of the segment must be greater than or equal to both CPL and the selector's RPL.

The only faults that can occur are those produced by incorrectly addressing the memory operand that contains the selector. The selector is not loaded into a segment register and no faults attributed to the selector operand are produced.

See Chapter 6, "80286/80386-Based Personal Computers" in the IBM *Macro Assembler/2 Fundamentals* book for information about the 80286 architecture.

You must use the .286P pseudo-op to enable this instruction.

Logic

perform write-validity check on selector in register
  or memory operand
if check succeeds, then ZF <- 1
else ZF <- 0

Flags

Affected— ZF

Encoding

00001111 00000000 mod101r/m
  0F   00   mod101r/m

Example

```
VERW BX
   or
VERW THING
```

WAIT
Wait

Purpose

This instruction allows the processor to synchronize itself with external hardware by placing the processor in a wait state until an external interrupt occurs.

Format

WAIT

Flags

None

Encoding

10011011

9B

Example

```
WAIT
```

XCHG
Exchange

Purpose

XCHG exchanges the byte or word *source* operand with the *destination* operand.

Format

XCHG *destination,source*

Remarks

A Bus Lock is asserted for the duration of the exchange.

The segment registers cannot be operands of XCHG.

There are two forms of the XCHG instruction, one for switching the contents of the accumulator with those of some other general word register, and one for switching a register and a memory-or-register operand.

The exchange is performed by the following operations:

1. The contents of the *destination* (leftmost operand) are temporarily stored in an internal word register:

   (Temp) <- (DEST)

2. The contents of the *destination* are replaced by the contents of the *source* (rightmost) operand:

   (DEST) <- (SRC)

3. The former contents of the *destination* are moved from the work register into the *source* operand:

   (SRC) <- (Temp)

Logic

(Temp) <- (DEST)
(DEST) <- (SRC)
(SRC) <- (Temp)

Flags

None

Register Operand with Accumulator

Encoding

10010reg

90 + reg

```
XCHG   AX,BX
XCHG   SI,AX
XCHG   CX,AX
```

Memory or Register Operand with Register Operand

Encoding

1000011w modregr/m

86 + w modregr/m

SRC = EA, DEST = REG

Example

```
XCHG   BETA_WORD,CX
XCHG   BX,DELTA_WORD
XCHG   DH,ALPHA_BYTE
XCHG   BL,AL
```

XLAT
Translate

Purpose

XLAT does a table lookup byte translation. The AL register is used as an index into a table (256 bytes at most) addressed by the DS:BX. The byte operand so addressed is transferred to AL.

Format

XLAT *source-table*

Remarks

The operand is the variable name whose address was moved to BX. The SEG component of this operand lets the assembler determine if the segment of the table is currently accessible. The TYPE attribute of the operand must be BYTE, or explicitly overridden with BYTE PTR. The content of AL is replaced by a byte from a table. The starting address of the table has been moved into register BX. The original contents of AL is the number of bytes past that starting address, where the desired translation byte is to be found. It replaces the contents of AL.

Logic

(AL) <- ((BX) + (AL))

Flags

None.

Encoding

11010111

D7

Example

```
MOV   AL,IMMED_BYTE
MOV   BX, OFFSET TABLE_NAME
XLAT  TABLE_NAME
```

XOR
Exclusive OR

Purpose

XOR does the bit-to-bit logical exclusive disjunction of the operands and returns the result to the *destination* operand.

Format

XOR *destination,source*

Remarks

XOR sets each bit in the *destination* (leftmost) operand to 0 if the corresponding bit positions in both operands are equal. If they are unequal, XOR sets the bits to 1.

Logic

(DEST) <- (LSRC) XOR (RSRC)
(CF) <- 0
(OF) <- 0

Flags

Affected— CF,OF,PF,SF,ZF
Undefined— AF

Encoding

001100dw modregr/m

30 + dw modregr/m

If d = 1, LSRC = REG, RSRC = EA, DEST = REG.
If d = 0, LSRC = EA, RSRC = REG, DEST = EA.

Example

Register with register:

```
XOR   AH,BL  ;RESULT IN AH, BL UNCHANGED
XOR   SI,DX  ;RESULT IN SI, DX UNCHANGED
XOR   CX,DI  ;RESULT IN CX, DI UNCHANGED
```

Memory with register:

```
XOR   AX,MEM_WORD
XOR   CL,MEM_BYTE[SI]
XOR   SI,ALPHA[BX][SI]
```

Register with memory:

```
XOR   BETA[BX][DI],AX
XOR   MEM_BYTE,DH
XOR   GAMMA[DI],BX
```

Immediate Operand to Accumulator

Encoding

0011010w data

34 + w data

If w = 0, LSRC = AL, RSRC = data, DEST = AL.
If w = 1, LSRC = AX, RSRC = data, DEST = AX

Example

Immediate (byte):

```
XOR   AL,11110110B
XOR   AL,0F6H
```

Immediate (word):

```
XOR   AX,23F6H
XOR   AX,75Q
XOR   AX,23F6H  ;AX DESTINATION
```

Immediate Operand to Memory or Register Operand

Encoding

1000000w mod110r/m data

80 + w mod110r/m data

LSRC = EA, RSRC = data, DEST = EA

Example

Immediate with register:

```
XOR   AH,0F6H
XOR   CL,37
XOR   DI,23F5H
```

Immediate with memory:

```
XOR   MEM_BYTE,3DH
XOR   GAMMA[BX][DI],0FACEH
XOR   ALPHA[DI],VAL_EQU_33H
```

Another example:

```
BITMASK EQU     20H
FLAGS   DB      ?
        .
        .
        .
        XOR     FLAGS,BITMASK ;TOGGLE STATE
                              ; OF FLAG BIT
```

Appendix A. Error Messages and Exit Codes

Error Messages

SALUT Error Messages

When SALUT detects an error, it inserts the word SALUTERR with the error message text line in both the formatted output and in the pre-processed output. When editing to fix the error, you do not have to delete this error line. If you assemble the .ASM file containing SALUT error messages, the Assembler will flag that line as containing an undefined op-code, thus adding it to its count of errors.

The text of the SALUTERR statement consists of one of the following:

- Structure mismatch
- Invalid continuation
- Invalid condition
- No structure active
- Invalid parameter
- Required parameter missing
- Extra characters on line
- Invalid structure indent
- Invalid opcode column
- Invalid operand column
- Invalid remark column

Macro Assembler Error Messages

If you have any errors in your assembler program, the assembler either:

- Inserts the error message into the listing file or
- Displays the error message on the screen.

After you check and correct your source program, assemble it again.

The Macro Assembler error messages are:

Code 0 - Block nesting error
Nested procedures, segments, structures, macros, IRPC, IRP, or REPT are not properly ended. For example, outer level of nesting is closed with inner level(s) still open.

Code 1 - Extra characters on line
This occurs when enough information to define the instruction directive is on a statement line, but extra characters beyond the end are also present.

Code 2 - Register already defined
This occurs only if the assembler has internal logic errors.

Code 3 - Unknown symbol type
The symbol statement has an unknown size type in a label or external declaration. Rewrite the declaration with a valid type such as BYTE, WORD, NEAR, etc.

Code 4 - Redefinition of symbol
If a symbol is defined in 2 places, this error occurs in pass 1 on the second declaration of the symbol. See errors 5 and 26.

Code 5 - Symbol is multi-defined
If a symbol is defined in 2 places, this error occurs in pass 2 on each declaration of the symbol. See errors 4 and 26.

Code 6 - Phase error between passes
The program has ambiguous instruction directives, so the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. For example, if a forward reference is coded without a segment override where one is required, an additional byte (the segment override) is generated in pass 2, causing the location of the next label to change. You can use the

/D option to produce a listing to aid in resolving phase errors between passes. (See the discussion of phase errors in the IBM *Macro Assembler/2 Assemble, Link, and Run* book.)

Code 7 - Already had ELSE clause
You attempted to define an ELSE clause within an existing ELSE clause.

Code 8 - Not in conditional block
You specified an ENDIF or ELSE without an active conditional assembly pseudo-op.

Code 9 - Symbol not defined
You used a symbol that has no definition.

Code 10 - Syntax error
The syntax of the statement is incorrect.

Code 11 - Type illegal in context
The type specified is the wrong size.

Code 12 - Should have been group name
The assembler expected a group name, but you entered something else.

Code 13 - Must be declared in pass 1
An item was referenced before it was defined in pass 1.

Code 14 - Symbol type usage illegal
The use of a PUBLIC symbol is incorrect.

Code 15 - Symbol already different kind
You attempted to define a symbol differently from its previous definition.

Code 16 - Symbol is reserved word
You attempted to use an assembler reserved word incorrectly (for example, to declare MOV as a variable).

Code 17 - Forward reference is illegal
You attempted to make a reference to something before it is defined in pass 1.

Code 18 - Must be register
The assembler expected a register as an operand, but you furnished a symbol.

Code 19 - Wrong type of register

The directive or instruction expected one type of register, and another was specified. For example, ASSUME AX was used instead of ASSUME *seg-reg*.

Code 20 - Must be segment or group

The assembler expected a segment or group and something else was specified.

Code 22 - Must be symbol type

You should have used WORD, DW, DQ, BYTE, or a similar designation but you specified something else.

Code 23 - Already defined locally

You tried to define a symbol as EXTRN that had already been defined locally.

Code 24 - Segment parameters are changed

The list of arguments to SEGMENT was not identical to the list the first time this segment was used.

Code 25 - Not proper align/combine type

The SEGMENT parameters were incorrect. Check the align and combine types.

Code 26 - Reference to multi-defined

The instruction refers to something that is defined more than once. For example, a symbol is defined in 2 places, or the location of a label has changed in value between pass 1 and pass 2. See errors 4 and 5.

Code 27 - Operand was expected

The assembler expected an operand, but received something else.

Code 28 - Operator was expected

The assembler expected an operator, but received something else.

Code 29 - Division by 0 or overflow

You gave an expression that results in division by 0 or a number larger than can be represented.

Code 30 - Shift count is negative

A shift expression is produced that results in a negative shift count.

Code 31 - Operand types must match

The assembler received different kinds or sizes of arguments in a case where they must match. For example, MOV AX, BH is illegal. Both operands must be WORD or both must be BYTE.

Code 32 - Illegal use of external

You used an external incorrectly. For example, DB M DUP(?) where M is declared external.

Code 34 - Must be record or field name

The assembler expected a record name or field name but did not receive it.

Code 35 - Operand must have size

The assembler expected the size of the operand but did not receive it. Often this error can be remedied by using the PTR operator to specify the size type.

Code 36 - Must be var, label or constant

The assembler expected a variable, label, or constant but received something else.

Code 38 - Left operand must have segment

You used something in the right operand that required a segment in the left operand; for example, ":symbol" is not correct; use "seg:symbol".

Code 39 - One operand must be constant

This is an incorrect use of the addition operator.

Code 40 - Operands must be same or 1 abs

This is an incorrect use of the subtraction operator.

Code 41 - Normal type operand expected

The assembler received STRUC, BYTE, WORD, or some other invalid operand when expecting a variable label.

Code 42 - Constant was expected

The assembler expected a constant but received an item that does not evaluabe to a constant. For example, a variable name or an external symbol.

Code 43 - Operand must have segment

This is an incorrect use of SEG directive.

Code 44 - Must be associated with data

You used a code-related item where a data-related item was expected. For example, using the DS override of a procedure would cause this message.

Code 45 - Must be associated with code

You used a data-related item where a code-related item was expected.

Code 46 - Already have base register

More than one base register was used in an operand.

Code 47 - Already have index register

More than one index register was used in an operand.

Code 48 - Must be index or base register

The instruction requires a base or index register, and you specified some other register within square brackets ([ ]).

Code 49 - Illegal use of register

You used a register with an instruction where there is no valid processor instruction possible.

Code 50 - Value is out of range

The value is too large for the expected use. For example, moving a DW to a byte register would cause this error.

Code 51 - Operand not in IP segment

Getting access to the operand is impossible because it is not in the currently executing segment.

Code 52 - Improper operand type

You used an operand such that the op code cannot be produced.

Code 53 - Relative jump out of range

Conditional jumps must be within the range from -128 through +127 bytes of the current instruction, and the specified jump is beyond this range. You can usually correct the problem by reversing the condition of the conditional jump and using an unconditional jump (JMP) to the out of range label.

Code 55 - Illegal register value

The register value specified does not fit into the reg field (the value field is greater than 7).

Code 56 - No immediate mode

You specified immediate data as an operand for an instruction that cannot accept the immediate. For example, MOV DS,DATA.

Code 57 - Illegal size for item

The size of the referenced item is incorrect. For example, shift of a double word.

Code 58 - Byte register is illegal

You used one of the byte registers in an incorrect context. For example, PUSH AL is not correct.

Code 59 - CS register illegal usage

You tried to use the CS register incorrectly. For example, XCHG CS, AX is not correct.

Code 60 - Must be AX or AL

You specified some register other than AX or AL where only these are acceptable. For example, the IN instruction requires AX or AL as an operand.

Code 61 - Improper use of segment register

You specified a segment register where this is incorrect. For example, an immediate move to a segment register is not correct.

Code 62 - Missing or unreachable CS

You tried to jump to an unreachable label.

Code 63 - Operand combination illegal

You specified a two-operand instruction where the combination specified is incorrect.

Code 64 - Near JMP/CALL to different CS

You attempted to do a NEAR jump or call to a location in a different code segment defined with a different ASSUME:CS.

Code 65 - Label cannot have segment override

This is an incorrect use of segment override.

Code 66 - Must have opcode after prefix

You used one of the prefix instructions, REPE, REPNE, REPZ, or REPNZ, without specifying the opcode after it.

Code 67 - Cannot override ES segment

You tried to override the ES segment in an instruction where this override is not legal. For example, STOS DS:TARGET is illegal.

Code 68 - Cannot address with segment register

There is no ASSUME that makes the variable reachable.

Code 69 - Must be in segment block

You attempted to produce code when it was not in a segment.

Code 70 - Cannot use EVEN on BYTE segment

A segment was declared to be a byte segment, and you attempted to use the EVEN pseudo-op.

Code 71 - Forward needs override or FAR

A FAR label is used in a CALL or JUMP before it is declared as being FAR.

Code 72 - Illegal value for DUP count

DUP counts must be a constant that evaluates to a positive integer greater than 0.

Code 73 - Symbol is already external

You attempted to define a symbol as local when it was already external.

Code 74 - DUP is too large for linker

Nesting of DUP operators created too large a record for the linker.

Code 75 - Usage of ? (indeterminate) bad

This is an incorrect use of the undefined operand (?). For example, ?+5 causes this error.

Code 76 - More values than defined with

Too many initial values were given when defining a variable using a REC or STRUC type.

Code 77 - Only initialized list legal

You used STRUC name without angle brackets (< >).

Code 78 - Pseudo-op illegal in STRUC

All statements incorrect within STRUC blocks must be either comments preceded by a semicolon (;) or one of the define pseudo-ops (DB, DW, DD, DQ, DT).

Code 79 - Override with DUP is illegal

In a STRUC initialization statement, you tried to use DUP in an override.

Code 80 - Field cannot be overridden

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Code 83 - Circular chain of EQU aliases

An EQU alias eventually points to itself.

Code 84 - Cannot emulate 8087 opcode

Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

Code 85 - End of file, no END pseudo-op

Either you forgot an END statement, or there is a nesting error. Possibly, an ENDM or ENDIF is missing. Also, you may have something between a LOCAL and MACRO statement.

Code 86 - Data emitted with no segment

Code that is not located within a segment attempted to produce data. An example is shown below:

```
code    SEGMENT
        .
        .
        .
code    ENDS
        push    ax
test    DW      ?
        END
```

Either of the two statements near the end of the example produces the error. Any statement that produces code or reserves data must be in a segment.

Code 87 - Forced error - pass1

You forced an error with the .ERR1 pseudo-op.

Code 88 - Forced error - pass2

You forced an error with the .ERR2 pseudo-op.

Code 89 - Forced error

You forced an error with the .ERR pseudo-op.

Code 90 - Forced error - expression equals 0

You forced an error with the .ERRE pseudo-op.

Code 91 - Forced error - expression not equal 0

You forced an error with the .ERRNZ pseudo-op.

Code 92 - Forced error - symbol not defined

You forced an error with the .ERRNDEF pseudo-op.

Code 93 - Forced error - symbol defined

You forced an error with the .ERRDEF pseudo-op.

Code 94 - Forced error - string blank

You forced an error with the .ERRB pseudo-op.

Code 95 - Forced error - string not blank
You forced an error with the .ERRNB pseudo-op.

Code 96 - Forced error - strings identical
You forced an error with the .ERRIDN pseudo-op.

Code 97 - Forced error - strings different
You forced an error with the .ERRDIF pseudo-op.

Code 98 - Override value is wrong length
The override value for a structure field is too large to fit in the field. An example is shown below:

```
x       STRUC
x1      DB      "A"
x       ENDS

y       x       <"AB">
```

In this example, the override value is a string consisting of two bytes, while the structure declaration only provided room for one.

Code 99 - Line too long expanding *symbol*
A symbol defined by an EQU or equal-sign (=) pseudo-op is so long that expanding it will cause the internal buffers of the assembler to overflow. This message may indicate a recursive text macro.

Code 100 - Impure memory reference
The code contains an attempt to store data in the code segment when the **.826p** pseudo-op and the /P option are in effect. An example of storing code to the code segment is shown below:

```
code    SEGMENT
        ASSUME cs:code
c_word  DW      ?
        .
        .
        .
        mov     cs:c_word,data
code    ENDS
```

The /P option checks for such statements that are acceptable in nonprotected mode but that can cause problems in protected mode.

Code 101 - Missing data; zero assumed

An operand is missing from a statement. For example:

```
mov     ax,
```

The code is assembled as if it were:

```
mov     ax,0
```

This is a warning error. The assembler does not delete the object file as it does with severe errors.

Unnumbered Error Messages

In addition to the above messages, the assembler can display the following unnumbered messages.

file(*linenumber*)**: Out of memory error**

Either the source is too big or too many symbols are in the symbol table. If an **Out of Memory** condition occurs with the assembler, there is insufficient work space to continue. To recover from such a condition, try rerunning the assembler but specifying only the oject file. On one run, get the listing; on another, the OBJ file. Some items that use up work space are: open INCLUDE files, many symbol names, long symbol names, open OUTPUT files, defined STRUCs, and MACROS. You can regain space by a PURGE of a MACRO after calling the MACRO for the last time before defining another MACRO. In MACRO definitions, separate fields with TAB characters, not a series of blanks, to conserve space. Avoid single semicolon comments in macros; use double semicolon comments instead.

Fatal assembler error

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

Error in expression analyzer

You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

Include file *filename* **not found**

Error defining symbol *symbol* **from command line**

You defined a symbol using the /D*symbol* option, but you used a character that the assembler does not allow in a symbol name. For example, if you specified */foo#*, you would get an error because the character # is not allowed in a symbol name.

End of file encountered on input file

You get this error when an end-of-file condition occurs before the END statement is processed.

Unable to open CREF file *filename*

Write error on object file

The file is read-only or the disk is full.

Write error on listing file
The file is read-only or the disk is full.

Write error on cross-reference file
The file is read-only or the disk is full.

Unable to open input file *filename*
The file could not be opened for reading.

Unable to access input file *filename*
The file could not be repositioned at the beginning for second pass.

Unable to open listing file *filename*
The file could not be opened for writing.

Unable to open object file *filename*
The file could not be opened for writing.

Extra filename ignored
The assembler displays this message when you give more than four file names.

Line invalid, start again
This error occurs when you give a directory/drive for source file, but no file name.

Buffer size expected after /b
You gave the /b option without the buffer size.

Path expected after /I
You gave the /I option without a path.

Unknown case switch c
You gave an -M**c** but did not specify **c** as u, x, or l.

Unknown switch c
You gave a **-c** but did not specify **c** as a valid option character.

Read error on stdin
The assembler encountered end-of-file on STDIN while prompting for a file name (STDIN redirected to a null or empty file, or CTL-Z entered).

Out of heap space
The assembler ran out of memory while it was storing the file names.

Linker Error Messages and Limits

This section lists error messages produced by the IBM linker, LINK. Limits imposed by the linker are described at the end of this section.

Fatal errors cause the linker to stop running. Fatal error messages have the following format:

location: **fatal error L1***xxx*:
message text

Non-fatal errors indicate problems in the executable file. LINK produces the executable file (and sets the error bit in the header if for protected mode). Non-fatal error messages have the following format:

location: **error L2***xxx*:
message text

Warnings indicate possible problems in the executable file. LINK produces the executable file (it does not set the error bit in the header if for protected mode). Warnings have the following format:

location: **error L4***xxx*:
message text

In these messages, *location* is the input file associated with the error, or LINK if there is no input file. If the input file is a module definitions file, the line number will be included, as shown below:

foo.def(3): fatal error L1030: missing internal name

If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

SLIBC.LIB(_file) MAIN.OBJ(main.asm)

The following error messages may appear when you link object files with LINK.

L1001 *option* **: option name ambiguous**
A unique option name does not appear after the option indicator (/). For example, the command

LINK /N main;

produces this error, since LINK cannot tell which of the three options beginning with the letter **N** is intended.

L1002 *option* **: unrecognized option name**
An unrecognized character followed the option indicator (/), as in the following example:

LINK /ABCDEF main;

L1003 *option* **: MAP symbol limit too high**
The specified symbol value limit following the MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.

L1004 *option* **: invalid numeric value**
An incorrect value appeared for one of the linker options. For example, a character string is entered for an option that requires a numeric value.

L1005 *option* **: packing limit exceeds 65536 bytes**
The number following the /PACKCODE option is greater than 65536.

L1006 *option* **: stack size exceeds 65534 bytes**
The size you specified for the /STACK option is more than 65534 bytes.

L1007 *option* **: interrupt number exceeds 255**
You gave a number greater than 255 as a value for the /OVERLAYINTERRUPT option.

L1008 *option* **: segment limit set too high**
The specified limit on the /SEGMENTS option is greater than 1024.

L1009 *option* **: CPARMAXALLOC : illegal value**
The number you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.

L1020 no object modules specified
You did not specify any object-file names to the linker.

L1021 cannot nest response files
A response file occurs within a response file.

L1022 response line too long
A line in a response file is longer than 127 characters.

L1023 terminated by user
You entered **Ctrl + C**.

L1024 nested right parentheses
You typed the contents of an overlay incorrectly on the command line.

L1025 nested left parentheses
You typed the contents of an overlay incorrectly on the command line.

L1026 unmatched right parenthesis
A right parenthesis is missing from the contents specification of an overlay on the command line.

L1027 unmatched left parenthesis
A left parenthesis is missing from the contents specification of an overlay on the command line.

L1030 missing internal name
In the module definitions file, when you specify an import by entry number, you must give an internal name, so the linker can identify references to the import.

L1031 module description redefined
In the module definitions file, a module description specified with the DESCRIPTION keyword is given more than once.

L1032 module name redefined
In the module definitions file, the module name is defined more than once with the NAME or LIBRARY keyword.

L1040 too many exported entries
An attempt is made to export more than 3072 names.

L1041 resident-name table overflow
The total length of all resident names, plus three bytes per name, is greater than 65534.

L1042 nonresident-name table overflow

The total length of all nonresident names, plus three bytes per name, is greater than 65534.

L1043 relocation table overflow

There are more than 65536 load-time relocations for a single segment.

L1044 imported-name table overflow

The total length of all the imported names, plus one byte per name, is greater than 65534 bytes.

L1045 too many TYPDEF records

An object module contains more than 255 TYPDEF records. These records describe communal variables. This error can only appear with programs produced by compilers that support communal variables.

L1046 too many external symbols in one module

An object module specifies more than the limit of 1023 external symbols. Break the module into smaller parts.

L1047 too many group, segment, and class names in one module

The program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

L1048 too many segments in one module

An object module has more than 255 segments. Split the module or combine segments.

L1049 too many segments

The program has more than the maximum number of segments. The SEGMENTS option specifies the maximum allowed number; the default is 128. Relink using the /SEGMENTS option with an appropriate number of segments.

L1050 too many groups in one module

The linker found more than 21 group definitions (GRPDEF) in a single module. Reduce the number of group definitions or split the module.

L1051 too many groups

The program defines more than 20 groups, not counting DGROUP. Reduce the number of groups.

L1052 too many libraries

An attempt is made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.

L1053 symbol table overflow

The program has more than 256K bytes of symbolic information, such as public, external, segment, group, class, and file names). Combine modules or segments and recreate the object files. Eliminate as many public symbols as possible.

L1054 requested segment limit too high

The linker does not have enough memory to allocate tables describing the number of segments requested (the default is 128 or the value specified with the /SEGMENTS option). Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 too many overlays

The program defines more than 63 overlays.

L1057 data record too large

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator (compiler or assembler) error. Note which translator produced the incorrect object module and the circumstances, and contact your authorized IBM dealer or representative.

L1070 segment size exceeds 64K

A single segment contains more than 64K bytes of code or data. Try compiling, or assembling, and linking using the large model.

L1071 segment _TEXT larger than 65520 bytes

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this is increased to 16 for alignment purposes.

L1072 common area longer than 65536 bytes

The program has more than 64K bytes of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler/2. It occurs only with programs produced by IBM C/2 or other compilers that support communal variables.

L1073 file-segment limit exceeded
There are more than 255 physical or file segments.

L1074 *name* **: group larger than 64K bytes**
A group contained segments which total more than 65536 bytes.

L1075 entry table larger than 65535 bytes
There is a limit of 65535 bytes imposed by OS/2 on the Entry Table in an OS/2 executable (program or library). An Entry Table entry is generated for each exported name in a dynlink library, so reducing the number of exported names may solve the problem.

L1080 cannot open list file
The disk or the root directory is full. Delete or move files to make space.

L1081 out of space for run file
The disk on which the .EXE file is being written is full. Free more space on the disk and restart the linker.

L1082 stub .EXE file not found
The stub file specified in the module definitions file is not found.

L1083 cannot open run file
The disk or the root directory is full. Delete or move files to make space.

L1084 cannot create temporary file
The disk or root directory is full. Free more space in the directory and restart the linker.

L1085 cannot open temporary file
The disk or the root directory is full. Delete or move files to make space.

L1086 scratch file missing
Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

L1087 unexpected end-of-file on scratch file
The disk with the temporary linker-output file is removed.

L1088 out of space for list file
The disk on which the listing file is being written is full. Free more space on the disk and restart the linker.

L1089 *filename* **: cannot open response file**
The linker could not find the specified response file. This usually indicates a typing error.

L1090 cannot reopen list file
The original disk is not replaced at the prompt. Restart the linker.

L1091 unexpected end-of-file on library
The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.

L1092 cannot open module definitions file
The specified module definitions file cannot be opened.

L1100 stub .EXE file invalid
The stub file specified in the definitions file is not a valid .EXE file.

L1101 invalid object module
One of the object modules is non-valid. If the error persists after recompiling, contact your authorized IBM dealer or representative.

L1102 unexpected end-of-file
A non-valid format for a library was found.

L1103 attempt to access data outside segment bounds
A data record in an object module specified data extending beyond the end of a segment. This is a translator (compiler or assembler) error. Note which translator produced the incorrect object module and the circumstances, and contact your authorized IBM dealer or representative.

L1104 *filename* **: not valid library**
The specified file is not a valid library file. This error causes the linker to stop running.

L1110 DOSALLOCHUGE failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

L1111 DOSREALLOCHUGE failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer or representavie.

L1112 DOSGETHUGESHIFT failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

L1113 unresolved COMDEF; internal error

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

L1114 file not suitable for /EXEPACK; relink without

For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image. Relink without the EXEPACK option.

L2000 imported entry point

A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.

L2001 fixup(s) without data

A fix-up record occurred without a data record immediately preceding it. This is probably an assembler error. See the *IBM Operating System/2 Programmer's Guide* for more information on fix-up.

L2002 fixup overflow near *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*

The following conditions can cause this error:

- A group is larger than 64K bytes.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link.
- An EXTRN declaration in an assembler-language source file appeared inside the body of a segment. For example:

```
code    SEGMENT public  'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public  'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and recreate the object file.

L2003 intersegment self-relative fixup

An intersegment self-relative fixup is not allowed.

L2004 LOBYTE-type fixup overflow

A LOBYTE fixup produced an address overflow.

L2005 fixup type unsupported

A fixup type occurred that is not supported by the linker. This is probably a compiler error. You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

L2010 too many fixups in LIDATA record

There are more fixups applying to a LIDATA record than will fit in the linker's 1024-byte buffer. The buffer is divided between the data in the LIDATA record itself and run-time relocation items, which are 8 bytes apiece, so the maximum varies form 0 to 128. This is probably a translator (compiler or assembler) error.

L2011 *'name'* : NEAR/HUGE conflict

Conflicting NEAR and HUGE attributes are given for a communal variable. This error can occur only with programs produced by compilers that support communal variables.

L2012 *'name'* : array-element size mismatch

A far communal array is declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler/2. It occurs only with IBM C/2 and any other compiler that supports far communal arrays.

L2013 LIDATA record too large

A LIDATA record in an object module contains more than 512 bytes of data. Most likely, an assembly module contains a very complex structure definition or a series of deeply-nested DUP operators. For example, the following structure definition causes this error:

```
alpha   DB     10 DUP(11 DUP(12 DUP(13 DUP(...))))
```

Simplify the structure definition and reassemble. (LIDATA is a DOS term).

L2020 no automatic data segment

No group named DGROUP is declared.

L2021 library instance data not supported in real mode

The library module is directed to have instance data. This works in protected mode only.

L2022 *name* **alias** *internalname.*: **export undefined**

A name is directed to be exported but is not defined anywhere.

L2023 *name* **alias** *internalname.*: **export imported**

An imported name is directed to be exported.

L2024 *name* **: symbol already defined**

One of the special overlay symbols required for overlay support is defined by an object.

L2025 *'name'* **: symbol defined more than once**

Remove the extra symbol definition from the object file.

L2026 multiple definitions for entry ordinal *number*

More than one entry point name is assigned to the same ordinal.

L2027 *name* **: ordinal too large for export**

You tried to export more than 3072 names.

L2028 automatic data segment plus heap exceeds 64K

The size of DGROUP near data plus requested heap size is greater than 64K.

L2029 unresolved externals

One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:

```
_exit in file(s)
 main.obj (main.c)
_fopen in files(s)
 fileio.obj(fileio.c) main.obj(main.c)
```

The name that comes before **in file(s)** is the unresolved external symbol. On the next line is a list of object modules which have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 starting address not code (use class 'CODE')

The program or dynamic link library has defined a starting address, or initial CS:IP, whose segment is not a code segment. Change the definition of the starting segment so that it has a class name ending in 'CODE'.

L4000 seq disp. included

This message is generated by the /WARNFIXUP switch (q.v.).

L4001 frame-relative fixup, frame ignored

A fixup occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fixup is meaningless in protected mode, so the target segment is assumed for the frame segment.

L4002 frame-relative absolute fixup

A fixup occurred with a frame segment different from the target segment where both frame and target segments were absolute. This fixup is processed using base-offset arithmetic, but the warning is issued because the fixup may not be valid in the OS/2 environment.

L4010 invalid alignment specification

The number following the **/ALIGNMENT** option is not a power of 2 or is not in numerical form.

L4011 PACKCODE value exceeding 65500 unreliable

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4012 load-high disables EXEPACK

The options **/HIGH** and **/EXEPACK** are mutually exclusive.

L4013 invalid option for new-format executable file ignored

If an OS/2 environment format program is being produced, then the options **/CPARMAXALLOC**, **/DSALLOCATE**, **/EXEPACK**, **/NOGROUPASSOCIATION**, and **/OVERLAYINTERRUPT** are meaningless, and the linker ignores them.

L4014 invalid option for old-format executable file ignored

If a DOS format program is produced, the options **/ALIGNMENT**, **/NOFARCALLTRANSLATION**, and **/PACKCODE** are meaningless, and the linker ignores them.

L4020 *name* **: code-segment size exceeds 65500**

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4021 no stack segment

The program does not contain a stack segment defined with STACK combine type. This message should not appear for modules compiled with IBM C/2, but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

L4022 *name1, name2* **: groups overlap**

Two groups are defined such that one starts in the middle of another. This may occur if you defined segments in a module definitions file or assembly file and did not correctly order the segments by class.

L4023 *exportname* **: export internal-name conflict**

An exported name, or its associated internal name, conflict with an already-defined public symbol.

L4024 *name* **: multiple definitions for export name**

The name *name* is exported more than once with different internal names. All internal names except the first are ignored.

L4025 *name* **: import internal-name conflict**

An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored. The conflict may come from a definition in either the module definitions file or an object file.

L4026 *modulename* **: self-imported**

The module definitions file directed that a name be imported from the module being produced.

L4027 *name* **: multiple definitions for import internal-name**

An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.

L4028 *name* **: segment already defined**

A segment is defined more than once with the same name in the module definitions file. Segments must have unique names for the linker. All definitions with the same name after the first are ignored.

L4029 *name* **: DGROUP segment converted to type data**

A segment which is a member of DGROUP is defined as type CODE in a module definition file or object file. This probably happened because a CLASS keyword in a SEGMENTS statement is not given.

L4030 *name* **: segment attributes changed to conform with automatic data segment**

The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute. For example, the *shared* attribute is NONSHARED and the *instance* is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.

L4031 *name* **: segment declared in more than one group**

A segment is declared to be a member of two different groups. Correct the source file and recreate the object files.

L4032 *name* **: code-group size exceeds 65500 bytes**

Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4034 more than 239 overlay segments; extra put in root

The IBM C/2 overlay manager has a limit of 238 code segments which can go in overlays (data segments go in the root). Any code segments encountered after the limit is reached are assigned to the root. This error will not be applicable to the IBM Macro Assembler/2.

L4036 no automatic data segment

The program or dynamic link did not define a group named "DGROUP", which is recognized by the linker as the automatic data segment.

L4040 NON-CONFORMING : obsolete

In the module definitions file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.

L4041 HUGE segments not yet supported

This feature is not yet implemented in the linker.

L4042 cannot open old version

An old version of the EXE file, specified with the OLD keyword in the module definitions file, could not be opened.

L4043 old version not segmented-executable format

The old version of the .EXE file, specified with the OLD keyword in the module definitions file, does not conform to segmented-executable format.

L4045 <name> : is name of output file

The user created a dynlink library file without specifying an extension. In such cases the linker supplies an extension of ".DLL". This is to warn the user in case he expected an ".EXE" file to be generated.

L4046 module name different from output file name

The user specified a module name via the NAME or LIBRARY statement in the definitions file which is different from the output file (.EXE or .DLL) name. This will likely cause problems in BINDing the file or in using it in protected mode so the user should rename the file to match the module name before it is executed.

L4050 too many public symbols

The **/MAP** option is used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (the default is 2048 symbols). The linker produces an unsorted listing of the public symbols. Relink using **/MAP**:*number* where *number* > 2048.

L4051 *filename* **: cannot find library**

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

L4053 VM.TMP : illegal file name; ignored

VM.TMP appears as an object-file name. Rename the file and rerun the linker.

L4054 *filename* **: cannot find file**

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

Linker Limits

The table below summarizes the limits imposed by the linker. If you find one of these limits, you may adjust your program so that the linker can accommodate it.

Item	Limit
Symbol table	256K
Load-time relocations (for DOS programs)	Default is 32K. If **/EXEPACK** is used, the maximum is 512K.
Public symbols	The range 7700-8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defined DGROUP so the effective maximum is 20.
Overlays	63
Segments	128 by default; however, this maximum can be set as high as 1024 by using the **/SEGMENTS** option of the LINK command.
Libraries	32
Group definitions per module	21

Item	Limit
Segments per module	255
Stack	64K

CodeView Error Messages

CodeView® [1] displays an error message whenever it detects a command it cannot run. You may see any of the following error messages. Most errors (start-up errors are the exception) end the CodeView command under which the error occurred but do not end CodeView.

Bad address
You specified the address in an invalid form. For example, you may have entered an address containing hexadecimal characters when the radix is decimal.

Bad breakpoint command
You typed an invalid breakpoint number with the BREAKPOINT CLEAR, BREAKPOINT DISABLE, or BREAKPOINT ENABLE command. The number must be in the range of 0 through 19.

Bad flag
You specified an invalid flag mnemonic with the REGISTER dialog command (**R**). Use one of the mnemonics displayed when you enter the command **RF**.

Bad format string
You specified an invalid type specifier following an expression. The valid type specifiers are **d**, **i**, **u**, **o**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, and **s**. Some type specifiers can be preceded by the prefix **h** or **l**.

Bad radix (use 8, 10, or 16)
CodeView only uses octal, decimal, and hexadecimal radixes.

Bad register
You typed the REGISTER command (**R**) with an invalid register name. Use **AX, BX, CX, DX, SP, BP, SI, DI, DS, ES, SS, CS, IP,** or **F**.

Bad type cast
This applies to C or BASIC programs only. Refer to the appropriate IBM *Language Reference* book.

---

[1] CodeView® is a trademark of Microsoft Corporation

Badly formed type

The type information in the symbol table of the file you are debugging is incorrect. If this message occurs, note the circumstances of the error and contact your authorized IBM Personal Computer dealer.

Breakpoint # or '' expected

You entered the BREAKPOINT CLEAR (BC), BREAKPOINT DISABLE (**BD**), or BREAKPOINT ENABLE (**BE**) commands with no argument. These commands require that you specify the number of the breakpoint to be acted on, or that you specify an asterisk (*), indicating that all breakpoints are to be acted on.

Cannot use struct or union as scalar

Applies to C or BASIC programs only. A **struct** or **union** variable cannot be used as a scalar value. Such variables must be followed by a file specifier or preceded with the address-of operator.

Cannot find file name

CodeView could not find the executable file you specified when you started. You probably misspelled the file name, or the file is in a different directory.

Constant too big

CodeView cannot accept a constant number larger than 4294967295 (0xFFFFFFFF).

Divide by zero

An expression in an argument of a dialog command attempts to divide by zero.

Expression too complex

An expression given as a dialog command argument is too complex. Simplify the expression.

Extra input ignored

You specified too many arguments to a command. CodeView evaluates the valid arguments and ignores the rest. Often in this situation, CodeView may not evaluate the arguments the way you intended.

Floating point error

This message should not occur, but if it does, note the circumstances of the error and contact your authorized IBM dealer or representative.

Internal debugger error

If this message occurs, note the circumstances of the error and contact your authorized IBM dealer or representative.

Invalid argument

One of the arguments you specified is not a valid CodeView expression.

Missing "

You specified a string as an argument to a dialog command, but you did not supply a closing double quote mark.

Missing '('

An argument to a dialog command was specified as an expression containing a right parenthesis but no left parenthesis.

Missing ')'

An argument to a dialog command was specified as an expression containing a left parenthesis but no right parenthesis.

Missing ']'

An argument to a dialog command was specified as an expression containing a left bracket but no right bracket. This error can also occur if a regular expression is specified with a right bracket but no left bracket.

No closing single quote

You specified a character in an expression used as a dialog command argument, but the closing single quote is missing.

No code at this line number

Applies only to C or BASIC programs. You tried to set a breakpoint on a source line that does not correspond to code. For example, the line may be a data declaration or a comment.

No match of regular expression

Applies only to C or BASIC programs. No match was found for the regular expression you specified with the SEARCH command or with the **Find** selection from the **Search** menu.

No previous regular expression

You selected **Previous** from the **Search** menu, but there was no previous match for the last regular expression specified.

No program to debug

You have reached the end of the program you are debugging. You must restart the program (using the RESTART command) before using any command that runs code.

No source lines at this address

Applies only to C or BASIC programs. The address you specified as an argument for the View command (**V**) does not have any source lines. For example, it could be an address in a library routine or an assembler-language module.

No such file/directory

A file you specified in a command argument or in response to a prompt does not exist. For example, the message appears when you select **Load** from the FILE menu, and then enter the name of a nonexistent file.

No symbolic information

Applies only to C or BASIC programs. The program file you specified is not in the CodeView format. You cannot debug in source mode, but you can use assembly mode.

Not a text file

You attempted to load a file using the Load selection from the FILE menu or using the VIEW command, but the file is not a text file. CodeView determines if a file is a text file by checking the first 128 bytes for characters that are not in the ASCII ranges of 9 through 13 and of 20 through 126.

Not an executable file

Applies only to C or BASIC programs. The file you specified to be debugged when you started CodeView is not an executable file having the extension .EXE or.COM.

Not enough space

You typed the SHELL ESCAPE command (**!**) or selected **Shell** from the **File** menu, but there is not enough free memory to run the command processor. The message also occurs with assembler-language programs that do not specifically release memory.

Object too big

You entered a TRACEPOINT command with a data object (such as an array) that is larger than 128 bytes. You can watch data objects larger than 128 bytes by using the memory version of the TRACEPOINT command.

Operand types incorrect for this operation

An operand had a type incompatible with the operation applied to it. For example, if **p** is declared as **char ***, then **? p*p** produces this error, because a pointer cannot be multiplied by a pointer.

Operator must have a struct/union type

You used the one of the member selection operators (-> or .) in an expression that does not refer to an element of a structure or a union.

Operator needs lvalue

You specified an expression that does not evaluate to an lvalue as in an operation that requires an lvalue. For example, **? 3 = 100** is nonvalid.

Program terminated normally (number)

You ran your program to the end. The number displayed in parentheses is the exit code returned to DOS or OS/2 by your program. You must use the RESTART command (or the START menu selection) to start the program before running more code.

Register variable out of scope

You tried to specify a register variable using the period (.) operator and a function name. For example, if you are in a third-level function, you can display the value of a local variable called **local** in a second-level function called **parent** with the following command:

```
? parent.local
```

However, this command does not work if **local** is declared as register variable. Instead, the message above is displayed.

Regular expression too complex

The regular expression specified is too complex for CodeView to evaluate.

Regular expression too long

The regular expression specified is too long for CodeView to evaluate.

Syntax error

You specified an invalid command line for a dialog command. Check for an invalid command letter. This message also appears if you enter an invalid assembler-language instruction using the Assemble command. The error will be preceded by a caret that points to the first character CodeView could not interpret.

Too many breakpoints

You tried to specify more than 20 breakpoints, which is the maximum allowed by CodeView.

Too many open files

You do not have enough files handles for CodeView to operate correctly. You must specify more files in your CONFIG.SYS file. See your IBM *Disk Operating System Reference* or IBM *Operating System/2 User's Reference* book for information about using the CONFIG.SYS file.

Type conversion too complex

Applies only to C or BASIC programs. You tried to type cast an element of an expression in a type other than the simple types or with more than one level of indirection. An example of a complex type would be type casting to a struct or union type. An example of two levels of indirection would be **char ****.

Unable to open file

A file you specified in a command argument or in response to a prompt cannot be opened. For example, the message appears when you select **Load** from the FILE menu, and then enter the name of a file that is corrupted or has its file attributes set so that it cannot be opened.

Unknown symbol

You specified an identifier that is not in the symbol table of CodeView. Check for a misspelling. A symbol name spelled with letters of the wrong case will not be recognized unless the **Case Sense** selection on the OPTIONS menu has been turned off. Another potential cause for this message is if you are trying to use a local variable in an argument when you are not in the function where the variable is defined.

Unrecognized option *option*

Valid options: /B /C*command* /F /S /T /W. You entered an invalid option when starting CodeView. Retype the command line.

Usage: cv ffloptions" file fflarguments"

You failed to specify an executable file when you started CodeView. Try again with the syntax shown in the message.

Video mode changed without /S option

The program changed video modes (from or to one of the graphics modes) when screen swapping was not specified. You must use the /S option to specify screen swapping when debugging graphics programs. You can continue debugging when you get this message, but the output screen of the debugged program may be damaged.

Warning: packed file

You started CodeView with a packed file as the executable file. You can attempt to debug the program in assembly mode, but the packing routines at the start of the program may make this difficult.

CREF Error Messages

The Cross-Reference Utility, CREF, ends operation and displays one of the following messages when it finds an error. The following error messages are in alphabetical order:

can't open cross-reference file for reading
The .CRF file is not found. Make sure the file is on the specified disk and that the name is spelled correctly in the command line.

can't open listing file for writing
May indicate that the disk is full or write protected, that a file with the specified name already exists, or that the specified device is not available.

cref has no switches
You specified an option in the command line with the slash (/) or dash (-) character, but CREF has no options.

extra file name ignore
You specified more than two files in the file name. CREF creates the reference file using only the first two files given.

line invalid, start again
You provided no .CRF file in the command line or at the prompt. CREF will display this message followed by a prompt asking for a .CRF file.

out of heap space
CREF cannot find enough memory to process the files. Try again with no resident programs or shells, or add more memory.

premature eof
You specified a file that is not a valid .CRF file, or the file is damaged.

read error on stdin
This error occurs if the program receives a Ctrl+Z from the keyboard or from a redirected file.

Library Manager Error Messages

Error messages produced by the IBM Library Manager, LIB, have one of the following formats:

- fflf*ilename*|LIB" : fatal error U1*xxx* : *messagetext*
- fflf*ilename*|LIB" : warning U4*xxx* : *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. LIB may display the following error messages:

U1150 page size too small
The page size of an input library is too small, which indicates a non-valid input .LIB file.

U1151 syntax error : illegal file specification
You gave a command operator, such as a minus sign (-), without a module name following it.

U1152 syntax error : option name missing
You gave a forward slash (/) with a value following it.

U1153 syntax error : option value missing
You gave the **/PAGESIZE** option without a value following it.

U1154 option unknown
An unknown option is given. Currently, LIB recognizes the **/PAGESIZE** option only.

U1155 syntax error : illegal input
The given command did not follow correct LIB syntax.

U1156 syntax error
The given command did not follow correct LIB syntax.

U1157 comma or new line missing
A comma or carriage return is expected in the command line, but did not appear. This may indicate an inappropriately placed comma, as in the following line:

LIB math.lib,-mod1 + mod2;

The line should have been entered as follows:

LIB math.lib -mod1 + mod2;

U1158 terminator missing

Either the response to the **Output library:** prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 cannot rename old library

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection. Change the protection of the old .BAK version.

U1162 cannot reopen library

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 error writing to cross-reference file

The disk or root directory is full. Delete or move files to make space.

U1170 too many symbols

More than 4609 symbols appeared in the library file.

U1171 insufficient memory

LIB did not have enough memory to run. Remove any shells or resident programs and try again, or add more memory.

U1172 no more virtual memory

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1173 internal failure

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1174 mark : not allocated

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1175 free : not allocated

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1180 write to extract file failed

The disk or root directory is full. Delete or move files to make space.

U1181 write to library file failed

The disk or root directory is full. Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory is full, or the specified extract file already exists with read-only protection. Make space on the disk or change the protection of the extract file.

U1183 cannot open response file

The response file was not found.

U1184 unexpected end-of-file on command input

An end-of-file character is received prematurely in response to a prompt.

U1185 cannot create new library

The disk or root directory is full, or the library file already exists with read-only protection. Make space on the disk or change the protection of the library file.

U1186 error writing to new library

The disk or root directory is full. Delete or move files to make space.

U1187 cannot open VM.TMP

The disk or root directory is full. Delete or move files to make space.

U1188 cannot write to VM

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1189 cannot read from VM

You should note the conditions when the error occurs and contact your authorized IBM deaker or representative.

U1190 DOSALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1191 DOSREALLOCHUGE failed

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1192 DOSGETHUGESHIFT failed

You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1200 *name* **: invalid library header**
The input library file has a non-valid format. It is either not a library file, or it has been corrupted.

U1203 *name* **: invalid object module near** *location*
The module specified by *name* is not a valid object module.

U4150 *modulename* **: module redefinition ignored**
A module is specified to be added to a library, but a module with the same name is already in the library. Or, a module with the same name is found more than once in the library.

U4151 *symbol(modulename)* **: symbol redefinition ignored**
The specified symbol is defined in more than one module.

U4152 *filename* **: cannot create listing**
The directory or disk is full, or the cross-reference listing file already exists with read-only protection. Make space on the disk or change the protection of the cross-reference listing file.

U4153 *number* **: page size too small; ignored**
The value specified in the **/PAGESIZE** option is less than 16.

U4155 *modulename* **: module not in library; ignored**
The specified module is not found in the input library.

U4156 *libraryname* **: output-library specification ignored**
An output library is specified in addition to a new library name. For example, specifying

LIB new.lib + one.obj,new.lst,new.lib

where **new.lib** does not already exist causes this error.

U4157 *filename* **: cannot access file**
LIB is unable to open the specified file.

U4158 *libraryname* **: invalid library header; file ignored**
The input library has an incorrect format.

U4159 *filename* **: invalid format** *hexnumber*; **file ignored**
The signature byte or word, *hexnumber*, of an input file is not one of the recognized types.

MAKE Error Messages

Error messages displayed by the IBM Program Maintenance Utility, MAKE, have one of the following formats:

- fff*filename*|MAKE" : fatal error U1*xxx* : *messagetext*
- fff*filename*|MAKE" : warning U4*xxx* : *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. MAKE produces the following error messages:

U1001 macro definition larger than *number*
A single macro is defined to have a value string longer than the number stated. Try rewriting the MAKE description file to split the macro into two or more smaller ones.

U1002 infinitely recursive macro
A circular chain of macros is defined, as in the following example:

- A = $(B)
- B = $(C)
- C = $(A)

U1003 out of memory
MAKE ran out of memory for processing the MAKE description file. Try to reduce the size of the MAKE description file by reorganizing or splitting it.

U1004 syntax error : macro name missing
The MAKE description file contained a macro definition with no left side (that is, a line beginning with =).

U1005 syntax error : colon missing
A line that should be an outfile/infile line lacked a colon indicating the separation between outfile and infile. MAKE expects any line following a blank line to be an outfile/infile line.

U1006 *targetname* **: macro expansion larger than** *number*
A single macro expansion, plus the length of any string it may be concatenated to, is longer than the number stated. Try rewriting the MAKE description file to split the macro into two or more smaller ones.

U1007 multiple sources
An inference rule is defined more than once.

U1008 *name* **: cannot find file or directory**
The file or directory specified by *name* could not be found.

U1009 *command* **: argument list too long**
A command line in the MAKE description file is longer than 128 bytes, which is the maximum that DOS allows. Rewrite the commands to use shorter argument lists.

U1010 *filename* **: permission denied**
The file specified by *filename* is a read-only file.

U1011 *filename* **: not enough memory**
Not enough memory is available for MAKE to run a program.

U1012 *filename* **: unknown error**
You should note the conditions when the error occurs and contact your authorized IBM dealer or representative.

U1013 *command* **: error** *errcode* **(ignored)**
One of the programs or commands called in the MAKE description file returned with a nonzero error code. If MAKE is run with the **/I** option, MAKE displays **(ignored)** and continues. Otherwise, MAKE stops running.

U4000 *filename* **: target does not exist**
This usually does not indicate an error. It warns you that the target file does not exist. MAKE runs any commands given in the block description since in many cases the outfile file will be created by a later command in the MAKE description file.

U4001 dependent *filename* **does not exist; target** *filename* **not built**
MAKE could not continue because a required infile file did not exist. Make sure that all named files are present and that they are spelled correctly in the MAKE description file.

U4014 usage : make ffl/n" ffl/d" ffl/i" ffl/s" ffname=value..." file
MAKE has not been called correctly. Try entering the command line again with the syntax shown in the message.

EXEMOD Error Messages

Error messages from the IBM EXE File Header Utility, EXEMOD, have one of the following formats:

- [*filename*|EXEMOD] : fatal error U1*xxx* : *messagetext*
- [*filename*|EXEMOD] : warning U4*xxx* : *messagetext*

The message begins with the input file name (*filename*), if one exists, or with the name of the utility. EXEMOD produces the following error messages:

U1050 usage : exemod file ffl-/h" ffl-/stack n" ffl-/max n" ffl-/min n"
You did not specify the EXEMOD command line properly. Try again using the syntax shown. Note that the option indicator can be either a slash (/) or a dash (-). The single brackets (ffl") in the error message show your optional choices.

U1051 invalid .EXE file : bad header
The specified input file is not an executable file or has an incorrect file header.

U1052 invalid .EXE file : actual length less than reported
The second and third fields in the input-file header indicate a file size greater than the actual size.

U1053 cannot change load-high program
When the minimum allocation value and the maximum allocation value are both zero, you cannot change the file.

U1054 file not .EXE
EXEMOD adds the .EXE extension to any file name without an extension. In this case, no file with the given name and an .EXE extension was found.

U1055 *filename* **: cannot find file**
The file specified by *filename* was not found.

U1056 *filename* **: permission denied**
The file specified by *filename* is a read-only file.

U4050 packed file
The given file is a packed file. This is a warning only.

U4051 minimum allocation less than stack; correcting minimum

If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the changed request), the minimum allocation value is adjusted. This is a warning message only; the change is still performed.

U4052 minimum allocation greater than maximum; correcting maximum

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the change is still performed.

Exit Codes

All the executable programs in the IBM Macro Assembler/2 package return a code (sometimes called an error-level code) that batch files or other programs such as MAKE use. If the program finishes without errors, it returns a code of 0. The code returned varies if the program finds an error. This appendix lists the return codes for the Macro Assembler. The sections are divided into the type of exit code: Macro Assembler, CREF, LIB, MAKE, and EXEMOD.

How Batch Files Use Exit Codes

If you prefer to use batch files, you can test the code returned with the IF ERRORLEVEL command. The sample batch file below, ASMBL.BAT or ASMBL.CMD (for OS/2 mode), illustrates how to do this:

```
MASM %1;
IF NOT ERRORLEVEL 1 LINK %1;
IF NOT ERRORLEVEL 1 %1
```

If you run this sample batch file with a command,

```
ASMBL test
```

DOS first runs the command,

```
MASM test;
```

and returns a code of 0, if the assembler finds no fatal errors, or a higher code if the assembler finds an error. In the second line, DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), DOS performs the command:

```
LINK test;
```

and again returns a code that is tested by the third line, which executes the program if the LINK step returned a code of 0.

Exit Codes for Programs in the IBM Macro Assembler/2 Package

When a program in the assembler package runs with no fatal errors, it returns an exit code of 0. When there are warning errors, the program also returns an exit code of 0. Some programs can return various codes to distinguish different kinds of errors. Other programs always return an exit code of 1 to indicate an error. The exit codes for each program in the assembler package are listed below:

Macro Assembler Exit Codes

No error (Code 0)
Argument error (Code 1)
Unable to open input file (Code 2)
Unable to open listing file (Code 3)
Unable to open object file (Code 4)
Unable to open cross-reference file (Code 5)
Unable to open include file (Code 6)
Assembly error (Code 7)

> **Note:** If the exit code is 7, the assembler erases the incorrect object file.

Memory allocation error (Code 8)
Error defining symbol from command line (Code 10)
User interrupted (Code 11)

CREF Exit Codes

No error (Code 0)
Any CREF fatal error (Code 1)

LIB Exit Codes

No error (Code 0)
All LIB fatal errors not listed below (Code 1)
Internal error (Code 4)
Too many symbols (Code 13)
4609 symbols is the maximum.

Page size too small (Code 16)
Page size must be 16 or more.

Mark: not allocated, 4 (internal error)
Free: not allocated, 4 (internal error)
Internal error, contact your authorized IBM dealer or representative.

An exit code of 0 means no errors.

How MAKE Uses Exit Codes

When an error is found, MAKE stops the program execution and displays the exit code as part of the error message.

For example, assume the MAKE description file TEST contains the following lines:

```
test.obj :     test.asm
     MASM test;
```

If the source code in TEST contains an assembly error, you would see this message the first time you use MAKE with the file TEST:

```
make: MASM test; - error 7
```

This error message shows that the command

```
MASM test;
```

in the MAKE description file returned code 7.

MAKE Exit Codes

No error (Code 0)
Any MAKE fatal error (Code 1)

If a program called by a command in the MAKE description file produces an error, the MAKE error message displays the exit code.

EXEMOD Exit Codes

No error (Code 0)
Any EXEMOD fatal error (Code 1)

Appendix B. Instructions and Pseudo-Ops Listed by Task

8088 Instructions

Moving Data

Instruction	*Meaning*
IN	Input Byte or Word
LDS	Load Data Segment Register
LEA	Load Effective Address
LES	Load Extra Segment Register
MOV	Move
OUT	Output Byte or Word
XCHG	Exchange
XLAT	Translate

Moving Data - Related to Flags

Instruction	*Meaning*
LAHF	Load AH from Flags
SAHF	Store AH in Flags

Moving Data - Related to Stacks

Instruction	*Meaning*
POP	Pop Word Off Stack to Destination
POPF	Pop Flags Off Stack
PUSH	Push Word onto Stack
PUSHF	Push Flags onto Stack

Doing Arithmetic

Doing Addition

Instruction	*Meaning*
AAA	ASCII Adjust for Addition
ADC	Add with Carry
ADD	Addition
DAA	Decimal Adjust for Addition
INC	Increase Destination by One

Doing Subtraction

Instruction	*Meaning*
AAS	ASCII Adjust for Subtraction
DAS	Decimal Adjust for Subtraction
DEC	Decrease Destination by One
SBB	Subtract with Borrow
SUB	Subtract

Doing Multiplication

Instruction	*Meaning*
AAM	ASCII Adjust for Multiply
IMUL	Integer Multiply
MUL	Multiply, Unsigned

Doing Division

Instruction	*Meaning*
AAD	ASCII Adjust for Division
DIV	Division, Unsigned
IDIV	Integer Division, Signed

Comparing, Negating, Converting

Instruction	*Meaning*
CBW	Convert Byte to Word
CMP	Compare Two Operands
CWD	Convert Word to Doubleword
NEG	Negate, Form Two's Complement

Processing Logic

Using Logical Operators

Instruction	*Meaning*
AND	Logical AND
NOT	Logical NOT
OR	Logical Inclusive OR
TEST	Test (Logical Compare)
XOR	Exclusive OR

Rotating

Instruction	*Meaning*
RCL	Rotate Left Through Carry
RCR	Rotate Right Through Carry
ROL	Rotate Left
ROR	Rotate Right

Shifting

Instruction	*Meaning*
SAL/SHL	Shift Arithmetic Left/Logical Left
SAR	Shift Arithmetic Right
SHR	Shift Logical Right

Manipulating Strings

Instruction	*Meaning*
CMPS/CMPSB/CMPSW	Compare Byte or Word String
LODS/LODSB/LODSW	Load Byte or Word String
MOVS/MOVSB/MOVSW	Move Byte or Word String
REP/REPZ/REPE/REPNE/REPNZ	Repeat String Operation
SCAS/SCASB/SCASW	Scan Byte or Word String
STOS/STOSB/STOSW	Store Byte or Word String

Changing Control

Calling Procedures

Instruction	*Meaning*
CALL	Call a Procedure
RET	Return from Procedure

Interrupting

Instruction	*Meaning*
INT	Interrupt
INTO	Interrupt If Overflow
IRET	Interrupt Return

Jumping

Instruction	*Meaning*
J(condition)	Jump Short If Condition Met
JMP	Jump

Looping

Instruction	*Meaning*
LOOP	Loop Until Count Complete
LOOPE/LOOPZ	Loop If Equal/If Zero
LOOPNE/LOOPNZ	Loop If Not Equal/If Not Zero

Controlling the Processor

Changing Flags

Instruction	*Meaning*
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CLI	Clear Interrupt Flag (Disable)
CMC	Complement Carry Flag
STC	Set Carry Flag
STD	Set Direction Flag
STI	Set Interrupt Flag (Enable)

Other

Instruction	*Meaning*
ESC	Escape
HLT	Halt
LOCK	Lock Bus
NOP	No Operation
WAIT	Wait

8087 Instructions

Moving Data

Moving Packed Decimal Data

Instruction	*Meaning*
FBLD	Packed Decimal (BCD) Load
FBSTP	Packed Decimal (BCD) Store and Pop

Moving Integer Data

Instruction	*Meaning*
FILD	Integer Load
FIST	Integer Store
FISTP	Integer Store and Pop

Moving Real Data

Instruction	*Meaning*
FLD	Load Real
FST	Store Real
FSTP	Store Real and Pop

Moving Registers

Instruction	*Meaning*
FXCH	Exchange Registers

Making Comparisons

Instruction	*Meaning*
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

Doing Arithmetic

Doing Addition

Instruction	*Meaning*
FADD	Add Real
FADDP	Add Real and Pop
FIADD	Integer Add

Doing Subtraction

Instruction	*Meaning*
FISUB	Integer Subtract
FISUBR	Integer Subtract Reversed
FSUB	Subtract Real
FSUBP	Subtract Real and Pop
FSUBR	Subtract Real Reversed
FSUBRP	Subtract Real Reversed and Pop

Doing Multiplication

Instruction	*Meaning*
FIMUL	Integer Multiply
FMUL	Multiply Real
FMULP	Multiply Real and Pop

Doing Division

Instruction	*Meaning*
FDIV	Divide Real
FDIVP	Divide Real and Pop
FDIVR	Divide Real Reversed
FDIVRP	Divide Real Reversed and Pop
FIDIV	Integer Divide
FIDIVR	Integer Divide Reversed

Other

Instruction	*Meaning*
FABS	Absolute Value
FCHS	Change Sign
FPREM	Partial Remainder
FRNDINT	Round to Integer
FSCALE	Scale
FSQRT	Square Root
FXTRACT	Extract Exponent and Significant

Calculating Functions

Instruction	*Meaning*
FPATAN	Partial Arc Tangent
FPTAN	Partial Tangent
FYL2X	Y * Log_2X
FYL2XP1	Y * $Log_2(X + 1)$
F2XM1	2 to the X power-1

Loading Constants

Instruction	*Meaning*
FLDLG2	Load $Log_{10}2$
FLDLN2	Load Log base e of 2
FLDL2E	Load Log_2e
FLDL2T	Load Log_210
FLDPI	Load PI
FLDZ	Load Zero
FLD1	Load +1.0

Controlling the Processor

Storing/Restoring

Instruction	*Meaning*
FNSTCW	Store Control Word
FNSTENV	Store Environment
FNSTSW	Store Status Word
FRSTOR	Restore State
FSTCW	Store Control Word
FSTENV	Store Environment
FSTSW	Store Status Word

Other

Instruction	*Meaning*
FCLEX	Clear Exceptions
FDECSTP	Decrease 8087 Stack Pointer
FDISI	Disable Interrupts
FENI	Enable Interrupts
FFREE	Free Register
FINCSTP	Increase 8087 Stack Pointer
FINIT	Initialize Processor
FLDCW	Load Control Word
FLDENV	Load Environment
FNCLEX	Clear Exceptions
FNDISI	Disable Interrupts
FNENI	Enable Interrupts
FNINIT	Initialize Processor
FNOP	No Operation
FNSAVE	Save State
FSAVE	Save State
FWAIT	Wait (CPU Instruction)

80286 Instructions

Note: The **(P)** indicates a protected mode instruction.

Moving Data

Instruction	*Meaning*
ARPL (P)	Adjust Requested Privilege Level
INS/INSB/INSW	Input from Port to String
LAR (P)	Load Access Rights
LGDT (P)	Load Global Descriptor Table
LIDT (P)	Load Interrupt Descriptor Table
LLDT (P)	Load Local Descriptor Table
LMSW (P)	Load Machine Status Word
LSL (P)	Load Segment Limit
LTR (P)	Load Task Register
OUTS/OUTSB/OUTSW	Output String to Port
POPA	Pop All General Registers
PUSH	Push Immediate onto Stack
PUSHA	Push All General Registers

Controlling the Processor

Instruction	*Meaning*
CLTS (P)	Clear Task Switched Flag
SGDT (P)	Store Global Descriptor Table
SIDT (P)	Store Interrupt Descriptor Table
SLDT (P)	Store Local Descriptor Table
SMSW (P)	Store Machine Status Word
STR (P)	Store Task Register

Verifying Fields

Instruction	*Meaning*
BOUND	Check Array Index Against Bounds
VERR (P)	Verify Read Access
VERW (P)	Verify Write Access

Preparing for High Level Language

Instruction	*Meaning*
ENTER	Make Stack Frame for Procedure Parameters
LEAVE	High Level Procedure Exit

Doing Arithmetic

Instruction	*Meaning*
IMUL	Integer Immediate Multiply

Processing Logic

Instruction	*Meaning*
RCL	Rotate Left Through Carry
RCR	Rotate Right Through Carry
ROL	Rotate Left
ROR	Rotate Right
SAL/SHL	Shift Arithmetic Left/Shift Logical Left
SAR	Shift Arithmetic Right
SHR	Shift Logical Right

80287 Instructions

Setting Mode

Instruction	*Meaning*
FSETPM	Set Protected Mode

Controlling the Processor

Instruction	*Meaning*
FSTSW AX	Store Status Word in AX (Wait)
FNSTSW AX	Store Status Word in AX (No Wait)

Using Conditionals

Pseudo-op	*Meaning*
ELSE	Else
ENDIF	End If
IF	If
IFB	If Blank
IFDEF	If Defined
IFDIF	If Different
IFE	If Zero
IFIDN	If Identical
IFNB	If Not Blank
IFNDEF	If Not Defined
IF1	If Pass 1
IF2	If Pass 2

Using Conditional Errors

Pseudo-op	*Meaning*
.ERR	Error
.ERRB	Error String Blank
.ERRDEF	Error Symbol Defined
.ERRDIF	Error Strings Different
.ERRE	Error Expression Equals 0
.ERRIDN	Error Strings Identical
.ERRNB	Error String Not Blank
.ERRNDEF	Error Symbol Not Defined
.ERRNZ	Error Expression Not Equal 0
.ERR1	Error Pass 1
.ERR2	Error Pass 2

Ordering Segments

Pseudo-op	*Meaning*
.ALPHA	Alphabetic Order
.SEQ	Source File Order

Manipulating Data

Pseudo-op	*Meaning*
ASSUME	Identify Segment Register for Segment
COMMENT	Enter Program Comments
DB	Define Byte
DD	Define Doubleword
DQ	Define Quadword
DT	Define Tenbytes
DW	Define Word
END	End
ENDP	End PROC
ENDS	End SEGMENT and STRUC
EQU	Equal
=	Equal Sign
EVEN	Even Boundary
EXTRN	External Assembly Module
GROUP	Segments Under One Name
INCLUDE	Statements From Alternate Source File
LABEL	Create a Variable or Label
NAME	Set Module Name
ORG	Set Location Counter
PROC	Block of Code and Linkage
PUBLIC	Public Use Symbols
.RADIX	Change Default RADIX (Decimal)
RECORD	Define Record Type for 8 or 16 Bit Record
SEGMENT	Define a Segment
STRUC	Define Type Definition for Structure

Controlling Listings

Pseudo-op	*Meaning*
.CREF	Control Cross Reference Information
.LALL	List Complete Macro Text
.LFCOND	List False Conditions
.LIST	Control Output to Listing File
%OUT	Display Assembly Progress
PAGE	Control Listing Page Size
.SALL	Suppress Listing
.SFCOND	Suppress False Conditional Listing
SUBTTL	Specify Subtitle
.TFCOND	Control False Conditional Listing
TITLE	Specify Title

XALL	List Source Line
XCREF	Control Cross Reference Listing
XLIST	Control Output to Listing File

Using Macros

Pseudo-op	*Meaning*
ENDM	End MACRO, REPT, IRP, and IRPC
EXITM	End When Expansion Not Required
IRP	Repeat Block of Statements
IRPC	Repeat Block of Statements
LOCAL	Create Symbols for Labels
MACRO	Produce Sequence of Statements
PURGE	Delete MACRO Definition
REPT	Repeat Block of Statements

Changing Modes

Pseudo-op	*Meaning*
.186	Enable Assembly of 80186 Instructions
.286C	Enable Assembly of 80286 Real Mode Instructions
.286P	Enable Assembly of 80286 Protected Mode Instructions
.287	Enable Assembly of Floating Point Instructions
.8086	Enable Assembly of 8088 and 8086 Instructions
.8087	Enable Assembly of 8087 Instructions

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Numerics

Notes:

Notes:

Notes:

Notes:

Notes:

Notes:

Notes:

Notes:

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

00F8619