ASM86 ASSEMBLY LANGUAGE REFERENCE MANUAL

Order Number: 122385-001

Intel Corporation, 3065 Bowers Avenue, Santa C ara, CA 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	6/85	C.W.

Preface

How to Use This Manual

This manual describes the assembly language for the 8086/8088 and the 8087. You should already be familiar with the 8086/8087/8088 before attempting to use this manual. If you are not familiar with the 8086/8087/8088 family of processors, you may wish to read the following manuals first:

- *iAPX 86/88/186/188 User's Manual-Programmer's Reference*, order number 210911, which is the basic reference document for iAPX 86/88, and iAPX 186/188 users. It includes a general description of the 8087 Numeric Processor.
- *An Introduction to ASM86*, order number 121689, which is an introduction to programming in assembly language for the 8086/8088.

The *ASM86 Assembly Language Reference Manual* is provided with the *ASM86 Macro Assembler Operating Instructions*, which describes how to assemble your 8086/8087/8088 program modules. It also contains a list of manuals for related Intel development tools, such as utilities and high-level language translators.

Before plunging into this manual you should read Chapter 1. It introduces some of the concepts, terminology, and conventions that are used throughout the manual. Sections labeled ""Overview" are introductions to material covered in a chapter. These sections are intended to give you an overall perspective of the material. In Chapter 3, there are two sections entitled "Introduction to...". These sections introduce two data structures unique to the assembly language. You should read these sections early in your use of the manual. The following is a brief description of the chapter contents:

- Chapter 1 discusses the important issues of the machine architecture (registers, segmentation) and introduces the assembly language.
- Chapter 2 discusses the assembler directives that control segmentation (defining program segments).
- Chapter 3 discusses the definition of variables and labels and the definition and initialization of data storage. It also describes the many data structures supplied by the assembly language.
- Chapter 4 describes the possible operand types that you can use with machine instructions. It also describes the assembly-time expressions that you can use.
- Chapter 5 describes the directives that allow you to develop modular programs, both in assembly language and assembly language programs that will link to modules written in other 8086/8088 languages.
- Chapter 6 fully describes the instruction sets for the 8086/8088 and the 8087.
- Chapter 7 describes the macro language supplied by the assembler.

Notational Conventions

typeface	This typeface indicates computer output or user input. The characters must be entered in the order shown.

UPPERCASE — Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.

italic — Italic indicates a metasymbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:

- *directory-name* — Is that portion of a *pathname* that acts as a file locator by identifying the directory containing the *filename*.
- *filename* — Is a valid name for the part of a *pathname* that names a file.
- *pathname* — Is a valid designation for a file; in its entirety, a pathname consists of a *directory-name* and a *filename*.
- *pathname1*, *pathname2*, ... — Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
- *variable-name* — Is a valid name for a variable.
- V*x.y* — Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.

[ ] — Brackets indicate optional arguments and parameters.

{ } — Braces indicate that one and only one of the enclosed items must be entered unless the field is also surrounded by brackets, in which case, the item is optional.

{ }... — At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case, the items are optional. The items may be used in any order unless otherwise noted.

| — In syntax descriptions, the vertical bar separates options within brackets [ ] or braces { }.

... — Ellipses indicate that the preceding argument or parameter may be repeated.

[,...] — The preceding item may be repeated, but each repetition must be separated by a comma.

punctuation — Punctuation other than ellipses, braces, and brackets must be entered as shown.

<cr> — Indicates a carriage return.

Table of Contents

Chapter 1
Overview of the ASM86 Assembly Language

Chapter 2
Segmentation

Chapter 3
Defining and Initializing Data

Chapter 4
Accessing Data—Operands and Expressions

Chapter 5
Program Linkage Directives

Chapter 6
The 8086/8087/8088 Instruction Set

Chapter 7
The MACRO Processing Language

Figures

Tables

Overview of the ASM86 Assembly Language 1

The 8086/8087/8088 Development Environment

This chapter presents an overview of ASM86, a macro assembly language for the 8086 and 8088 microprocessors, optionally in combination with the 8087 Numeric Data Processor. The ASM86 Macro Assembler generates object modules, which contain machine instructions and data, from programs written in ASM86. Programs may be written solely in assembly language or can be a modular combination of ASM86 and other high-level language modules. After the ASM86 modules have been assembled, they can be processed by the following iAPX 86,88 utilities:

- LINK86 combines object modules into load modules.
- LOC86 binds load modules to absolute memory addresses.
- LIB86 helps build and manage libraries of object modules.
- OH86 converts an 8086/8088 object module to Intel Hex Format.

Figure 1-1 describes the iAPX 86,88 development process.

This revision of the *ASM86 Language Reference Manual* includes information on the iAPX 186 instructions. These instructions can be used only if you use the iAPX 186 assembler. The 186-only instructions are indicated by having iAPX 186 in parentheses after the mnemonic. Clocks of iAPX 186 are given in Appendix H of this manual.

An Overview of the Assembly Language

The assembly language for the 8086/8088 is used to write and structure programs to be assembled, linked, located, and executed on an 8086 or 8088 microprocessor, optionally in combination with an 8087 Numeric Data Processor. There are directives to control program segmentation, the allocation of data, including structured data types, and to structure multi-module programs through relocation and linkage directives. The assembly language features a set of operators for assembly-time expressions, which allow the user to manipulate and control the data typing in a simple way and supply a means to perform assembly time arithmetic.

A very important feature of the assembly language is its simplified instruction mnemonics. Many assemblers require the programmer to remember a different mnemonic for each machine opcode. For example, a "move immediate" would require a different mnemonic than a "move memory". The 8086/8088 instruction set uses a single mnemonic for each generic instruction type. Thus, all "moves" use the mnemonic "MOV". The opcode generated is dependent on the operands supplied with the instruction. A move from memory could be written—

```
MJV AX, COUNT
```

where COUNT is a variable. An immediate move would be written as:

```
MOV DX, 0A123H
```

In each case the mnemonic is the same. This simplification allows the programmer to concentrate on the programming task and not on remembering a large set of mnemonics. In order to determine the correct instruction to generate, the assembler examines

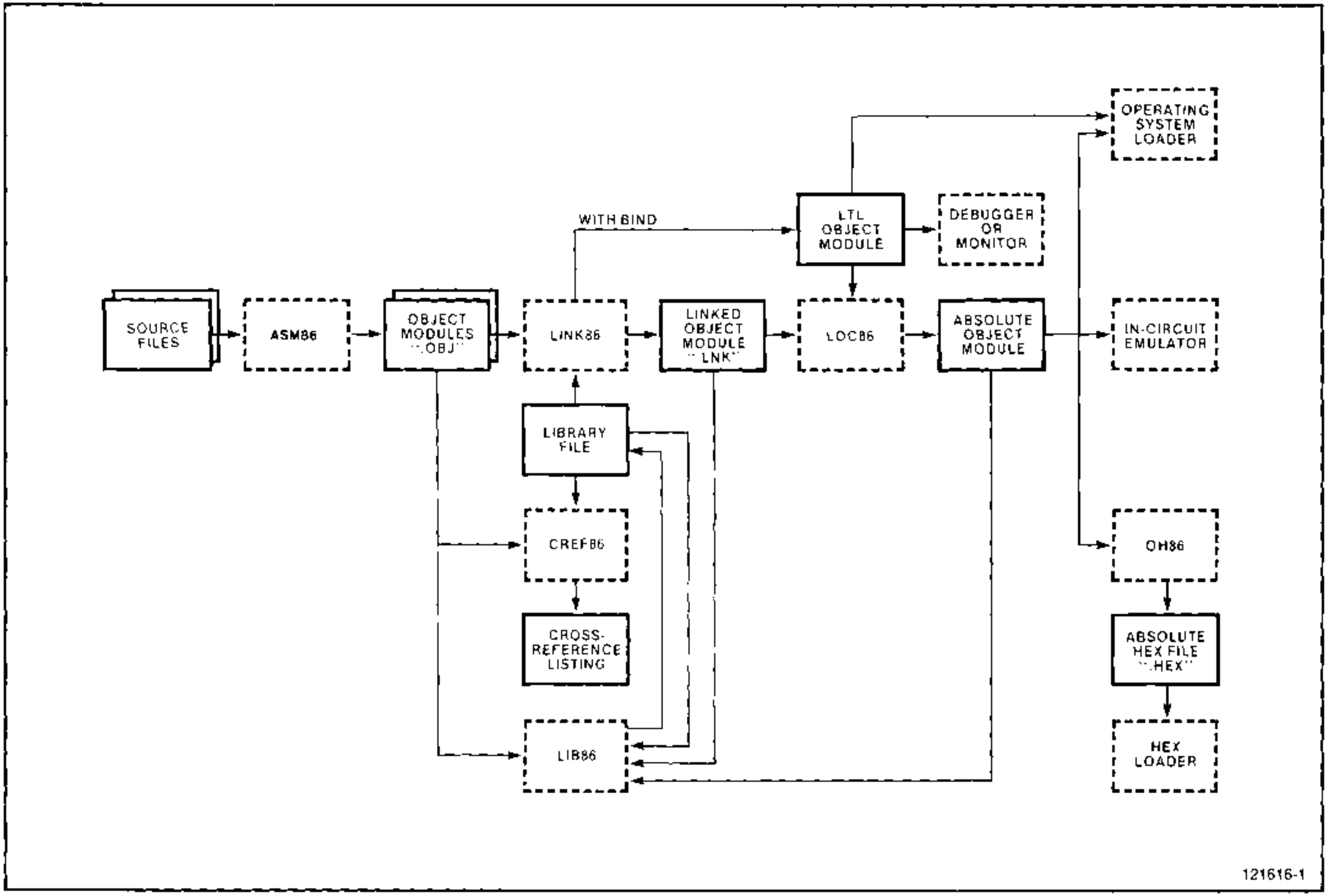


Figure 1-1. The iAPX 86,88 Family Development Process

the operands and determines their "type" (byte/word, variable/constant, etc.) and then uses this information to select the appropriate code.

The 8086 and 8088 have instructions to manipulate both 8 and 16-bit data. ASM86 is a "strongly-typed" language in that it checks that operands in an instruction are of the same "type". This prevents the programmer from inadvertantly moving a word variable into an 8-bit destination, for example. This would be an error that might not be detected until run-time. The assembler will catch this error at the time of assembly, saving the programmer the chore of debugging this error. However, one of the features of programming in assembly language is the ability to manipulate data in every possible way, including the above "illegal" operation. ASM86 has many directives and expression operators to override this typing mechanism so that these types of operations can be performed (see Chapter 4).

The assembler allows you to forward reference variables and labels in your program. A forward reference is a use of a variable or label prior to its definition. For example:

```
        MOV AX, COUNT      ; forward reference to COUNT
         .
         .
         .
COUNT DW  15               ; definition of COUNT
```

When you make a forward reference such as that shown above, the assembler must make a guess as to the nature of the thing referenced. In this case it will assume that it is a word variable because AX is a word register. However, it could be a constant if it was defined as:

```
COUNT EQU 15               ; definition of COUNT as a constant
```

It is possible for the assembler to make a poor or incorrect guess that could lead to an error message or inefficient code. It is recommended that you try to avoid forward references as much as possible in your program. A good practice is to define all your variables and data at the top of your program.

Basic Assembly Language Constituents

This section discusses the elements that constitute a source file in the ASM86 assembly language.

Character Set

The character set used in ASM86 is a subset of both ASCII and EBCDIC character sets. The valid characters consist of the alphanumerics:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

along with these special characters:

+ * / – ( ) [ ] < > ; ' . " , _ : ? @ $ &

and the non-printing characters:

space tab carriage-return line-feed

If an ASM86 program contains any character that is not in this set, the assembler will treat the character as a blank. The combination of a linefeed or carriage-return/linefeed immediately followed by an ampersand represents a continuation line and is treated as a blank (except within a character string or comment).

Upper- and lowercase letters are not distinguished from each other (except in character strings). For example, xyz and XYz are interchangeable.

Blanks are not distinguished from each other and any unbroken sequence of blanks is considered to be the same as a single blank (except within a character string).

Special characters and combinations of special characters have particular meanings in a program, as described in the remainder of this manual.

Tokens and Separators

A token is the smallest meaningful unit of a source program, much as words are the smallest meaningful units of a book in English. Separators are used to separate two adjacent tokens so that they are not mistakenly thought to be one longer token. The most commonly used separator is the blank. Any unbroken sequence of blanks may be used wherever a single blank is allowed. Horizontal tabs are also used as separators and are interpreted by the assembler identically to blanks except that they may appear as multiple blanks in the list file (see operator's manual). Any illegal character, or character used in an illegal context, is also treated as a separator.

Delimiters

Delimiters are special characters that serve to mark the end of a token and also have a special meaning unto themselves (as opposed to separators, which merely mark the end of a token). Commas, plus-signs, square brackets, etc., all serve as delimiters. When a delimiter is present, separators need not be used; however, using separators often makes your programs easier to read and, therefore, easier to understand.

Identifiers

An identifier is used to name a user-defined entity in a program. This could be a segment, a group, a variable, a label, or a constant defined with an EQU (Equate) directive. The format for an identifier is as follows:

1. The identifier must begin with a letter or one of three special characters:
   a. A question mark (?), with hexadecimal value 3FH
   b. A commercial at-sign (@), with hexadecimal value 40H
   c. An underscore (_), with hexadecimal value 5FH
2. It may contain letters or digits and the three special characters.
3. The identifier name is considered unique only up to 31 characters, but it can be of any length (up to 255 characters).
4. Every identifier has global scope within your program module.

Statements

Just as tokens may be seen as the assembly language counterparts to the English concept of words, so may statements be viewed as analogous to sentences. A statement is a specification to the assembler as to what action to perform. In fact, one way of viewing a computer program is as a sequence of statements which, when taken as an aggregate, is intended to perform a particular function. Statements may be divided into two types: instructions and directives.

Instructions are translated by the assembler into machine instruction code which "instruct" the 8086/8087/8088 to perform certain operations. Directives are not translated into machine instruction code by the assembler but rather "direct" the assembler itself to perform certain clerical functions.

Usually a statement will occupy one line in your source file. A line is a sequence of characters ended by a terminator (line-feed or carriage-return/line-feed combination). However, ASM86 provides for continuation lines which allow a statement to occupy more than one physical line in your source file. Any statement may be continued if the first character following the terminator is an "&". (Symbols, however, may NOT be broken across continuation lines. Character strings may not be continued across continuation lines; the string must be closed with an apostrophe on one line and then reopened with an apostrophe on a subsequent continuation line, with an intervening ",". Comments are considered to be ended by a terminator; if a comment is continued then the first non-blank character following the "&" must be a ";".)

An Overview of the Macro Language

The assembler contains as its front-end a macro processor. The macro processor scans the source file for macro definitions and macro calls written in Macro Processor Language (MPL). Macro calls are expanded according to macro definitions, and the resulting source assembly language is assembled by the assembler. By using MPL, you can create macros specific to your application that can generate sequences of assembly language instructions or directives. The macro processor is a very powerful string replacement facility that can help to simplify a programming task. Repeatedly used code sequences can be replaced by a simple macro call. Also, frequently used assembler directive statements can be replaced by macro calls. Details for the use of MPL are in Chapter 7.

CPU Hardware Overview

The 8086 and 8088 execute exactly the same instructions. The instruction set includes arithmetic and logical, program transfer, and data transfer operations. It also includes the following new operations:

- Multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers
- Move, scan, and compare operations for strings up to 64K bytes in length
- Non-destructive bit testing
- Byte translation from one code to another
- Software-generated interrupts
- A group of instructions that can help coordinate the activities of multiprocessor systems

This section will give a broad overview of the machine architecture by presenting the register set for the 8086/8088. The 8087 is discussed in Chapter 6.

The General Register Set

The 8086/8088 has a set of eight 16-bit general registers which are shown in Figure 1-2. These general registers are subdivided into two sets of four registers. The first set is called the data registers. Each 16-bit data register is further divided into two 8-bit registers, allowing its upper (high) and lower halves to be separately addressed. This means that each data register can be used interchangeably as a 16-bit register, or as two 8-bit registers. Each of these 16-bit and 8-bit registers can participate in arithmetic and logical operations. The data register set is given below:

16-Bit Register	**8-Bit Registers**	
	High	**Low**
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

The second set of general registers consists of the pointer and index registers. These registers can participate in most of the same 16-bit arithmetic and logical operations as the data registers. In most cases, however, these registers are used as pointer or index registers for addressing data objects in memory. The addressing modes available on the 8086/8088 are discussed in Chapter 4. These registers are:

BP – base pointer
SP – stack pointer
SI – source index
DI – destination index

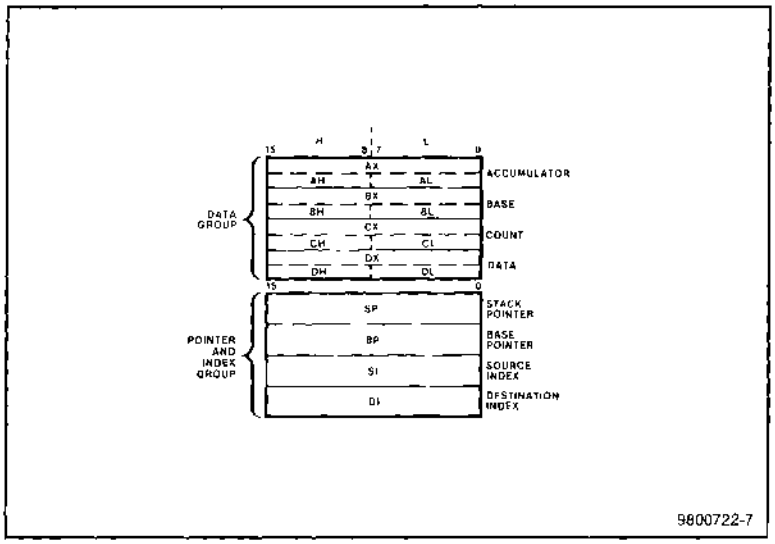


Figure 1-2. The General Register Set

Some of the 8086/8088 instructions make implicit use of general registers. Table 1-1 lists the general types of instructions which use these registers. You should refer to the complete description of each instruction given in Chapter 6 for a discussion of this implicit use.

Table 1-1. Implicit Use of General Registers

Register	Operations
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The Segment Register Set

The 8086/8088 is capable of addressing a megabyte of memory. This megabyte can be accessed through four segments by the CPU. Each segment is 64K bytes in size. The four segment registers (CS, SS, DS, ES) indicate the base locations for these segments. They are shown in Figure 1-3 below. The four segments are functionally defined as containing code, data (two segments), and the hardware stack. The CS register points to the current code segment, from which instructions are fetched. The SS register points to the current stack segment. All hardware stack operations are performed on locations in this segment. The DS register points to the current data segment that generally contains program variables. The ES register points to the current extra segment; it is typically used for data storage. These registers are accessible to programs and can be manipulated by several instructions.

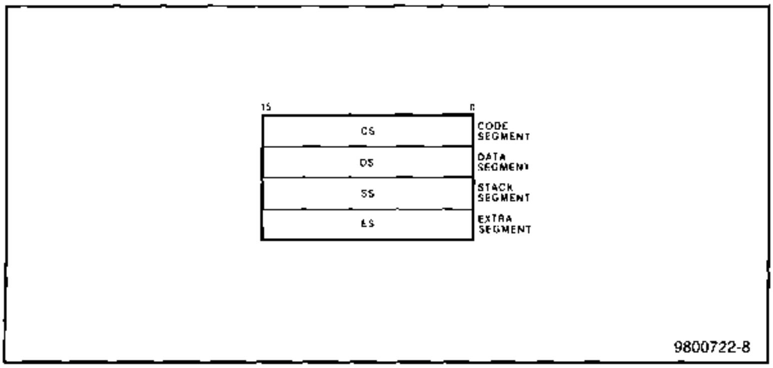


Figure 1-3. The Segment Register Set

The 8086/8088 Memory Segmentation Model

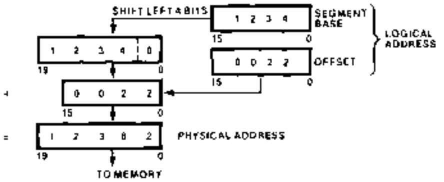
The 8086/8088 can address a megabyte of memory (1,048,576 bytes). This memory space is viewed by the CPU at run-time as four functional portions called physical segments. Each physical segment is dedicated for a particular use. One is dedicated to contain code, one for data, one for the hardware stack, and an extra segment that is usually used for data.

A segment register contains a 16-bit value, used to point to the start or base of a physical segment. The contents of a segment register determine the upper 16 bits of a 20-bit address. Thus, each physical segment must begin at an address whose low four bits are zero. Such a location is called a "paragraph boundary." The value in a segment register is called a "paragraph number." Thus the location 12340H is indicated by paragraph number 1234H. For a segment register to point to this location (denoting the start of a physical segment at that location) it would be loaded with the value 1234H defining a 64K segment starting at absolute address 12340H.

It requires 20 bits to address a megabyte of memory. The 20 bits are composed from two portions by the CPU. The first portion is the 16-bit paragraph number discussed above. It specifies where the physical segment begins in memory. Another quantity is required to specify the location of a particular object within that physical segment. This quantity is called the offset portion of the address. It defines a location at a specific offset from the start of the physical segment. Each offset is a 16-bit quantity, allowing you to address up to 64K bytes in a physical segment.

How then does the hardware generate a 20-bit address from these two values? First, the paragraph number in the appropriate segment register is multiplied by 16 (shifted left 4 bits). The result is then added to the offset yielding the 20-bit address (see Figure 1-4). The hardware automatically performs this operation. You, however must ensure: 1) that the correct paragraph number is loaded into the correct segment register and 2) that the instruction uses the correct offset value. The first is usually handled by some initialization code at the start of the program or by the loader. The second is handled by the assembler, as long as the instruction is correctly coded.

The assembler, while assembling the source file, is producing code/data for only one segment at a time. Within a segment, the assembler needs only to keep track of the offset of an object, whether it be code or data. The offset is referred to as the "location counter," which may be user programmable. This same situation is true during the execution of a program; only one segment is active at a given time for either code or data. Once a segment register is set with the base of a particular segment, objects within that segment can be referred to using only their offsets within that segment. Because of the segmentation model used, the programmer is usually only manipulating the offset part when coding an instruction.



9800722-18

Figure 1-4. Generating a Physical Address

An important concept to keep in mind while programming the 8086/8088 is "addressability." The object in memory which you are referencing must be addressable at run-time. This means that the appropriate segment register contains the base of the segment in which the object is located. You must insure that the proper value is loaded into a segment register before the object is referenced by an instruction. This is accomplished by using the appropriate sequence of instructions to initialize the segment register. There are assembler directives described in Chapter 2, which help to insure that you are aware of the addressability of data and code while writing a program. The following example shows the definitions of two segments, one for data and one for code. In the code segment, there is an illustration of the type of code that is used to initialize a segment register.

Example:

```
DATA SEGMENT                    ; define a segment for data

  ABYTE   DB    0               ; some data!

DATA ENDS                       ; end of segment definition

ASSUME CS:CODE, DS:DATA         ; defines the addressability
                                ; of the contents of these
                                ; segments

CODE SEGMENT
                                ; define a segment for code

         MOV   AX, DATA         ; AX = base address of DATA
         MOV   DS, AX           ;  segment. initialize DS,
data
                                ; now addressable through DS

                 .
                 .
                 .

CODE ENDS                       ; end of segment definition
```

A source module is a separately assembled or compiled source file. An executable program can be made up of one or more modules. A single module can define– ·

- A part of a physical segment
- A complete physical segment
- Parts of several physical segments
- Several complete physical segments

A physical segment is composed of one or more "logical segments." These are definitions of segments made in your program through the use of assembler directives designed for this purpose. You can specify a set of logical segments and their contents (code, data, etc.) and control how they are combined into physical segments. The mechanism for accomplishing this task is discussed in Chapter 2.

Format for Directive Specifications

The chapters that follow will discuss the form and use of the assembler directives. The following describes the format used to specify how each directive is written and the meaning and use of the different fields that can be part of a directive.

Syntax

The following notation is used to show the syntax of the directives.

1. Lower case strings represent fields that can be replaced either by user-supplied strings (such as names) or by assembler keywords. These items are referred to as Field Values. The actual replacement values are specified for each directive in the sections entitled *Field Values*.
2. Upper case strings represent assembler keywords (e.g., SEGMENT, DB, END, or NAME).
3. Optional items are shown in brackets, i.e., [item]. These items are optional in the syntax. Use of these fields is specified for each directive. In some cases the absence of this item ("none specified") denotes a default case, which is noted where appropriate.
4. The item [, ...] represents the optional repetition of a specific item. The syntax

   ```
   thing [, ... ]
   ```

   would expand to

   ```
   thing, thing, thing, ...
   ```

   The actual number of items that can appear in the list is typically limited by constraints internal to the assembler.

Description

This section is a paragraph which describes the meaning and use of the directive.

Field Values

This section describes the values used in specific fields shown in the syntax of the directive. Upper case strings denote assembler keywords.

Additional Notes

After the above sections, there may be sections dealing with important considerations, concepts, or applications of the particular directive. These sections should be read carefully.

Segmentation 2

Overview of Segmentation

The 8086/8088 directly addresses one megabyte of memory. This megabyte is viewed by the CPU through four segments, each containing up to 64K bytes. These four segments are called physical segments. The start of each segment is defined by the value placed in a segment register. This value is called a paragraph number. It defines a paragraph boundary in memory, an address divisible by 16 (least significant hexadecimal digit is equal to 0H).

The four segments are classified as code, data, stack, and extra. They are defined by the four segment registers:

CS for code ✓

DS for data ✓

SS for stack ✓

ES for extra ✓

Executable instructions will be in a physical segment defined by the value in CS. Any stack operations will occur within the segment defined by SS. Data is normally found in the segment defined by DS, but it can also be placed in the segments defined by the other segment registers. The contents of the physical segments in memory during the execution of a program are defined through the assembly language as logical segments. A physical segment can contain any number of logical segments that were specified in the program source code (either one or more modules). The SEGMENT directive is used to both define a logical segment and to control how the segments will be combined to form a physical segment. The GROUP directive is another way to combine logical segments with certain restrictions. Because all code and data must lie within some physical segment during program execution, a way to specify this addressability is required during assembly time to check for this condition. The ASSUME directive serves this purpose.

The SEGMENT/ENDS Directive

Syntax:

```
name   SEGMENT   [align-type]   [combine-type]   ['classname']
             .
             .
             .
name   ENDS
```

Description:

The SEGMENT/ENDS directive is used to define a logical segment. This segment may be combined with other segments in the same module and/or with segments defined in other modules. These segments will form the physical segments, located in memory, that are pointed to by the segment registers. The programmer will place within the SEGMENT/ENDS pair the code, data, or stack. Within a source module, each occurrence of an equivalent SEGMENT/ENDS pair (with the same name) is viewed as being one part of a single program segment.

Field Values:

name

The name for the segment, a unique ASM86 identifier.

[align-type]

This field specifies on what type of boundary in memory the segment will be located.

The values it may have are:

1. None specified—the default value of paragraph alignment. The segment will begin on an address divisible by 16 (i.e., an address whose least significant hexadecimal digit is equal to 0H).
2. PARA-- paragraph alignment (same as default).
3. BYTE--byte alignment; segment may start at any address.
4. WORD—word alignment; segment will start at an even address (i.e., least significant bit equal to 0B). (See EVEN directive, page 3-19.)
5. PAGE—page alignment; segment will start at an address whose two least significant hexadecimal digits are equal to 00H.
6. INPAGE—inpage alignment; the entire segment must fit within 256 bytes and, when located, must not overlap a page boundary (i.e., 00H, 100H, 200H, ... , 0FF00H).

[combine-type]

This field specifies how the segment will be combined with segments from other modules to form a physical segment in memory. The actual combination will occur during the LINK86 and LOC86 phase of development. The values for this field are:

1. None specified—the default value of non-combinable. The segment will not be combined with any other segment. (Note, however, that separate pieces of this segment in the same module will be combined.)
2. PUBLIC—all segments of the same name that are defined to be public will be combined (concatenated to form one physical segment). The order of combination is controlled during the use of LINK86. The length of the resulting physical segment will equal the sum of the lengths of the segments combined.
3. COMMON--all segments of the same name that are defined to be common will be overlapped to form one physical segment; all of the combined segments begin at the same physical address. The length of the physical segment will be equal to the length of the largest segment combined.
4. STACK—all segments of the same name that are defined to be stack will be combined into a physical segment so that each combined segment will *end* at the same address (overlaid against high memory) and will grow "downward." The length of the stack segment after combination will equal the sum of the lengths of the segments combined.
5. MEMORY—all segments of the same name that are defined to be memory will be combined so that the first memory segment encountered by LINK86 will be treated as the physical "memory" segment. In the list of modules linked together by LINK86, the first module that contains a "memory" segment will be used to define the physical "memory" segment. It will be located at an address above all other segments in the program. Any other segments of the type memory that are encountered by LINK86 will be combined as common with the first segment. The length of the memory segment will be equal to the length of the first memory segment encountered.

6. AT expression—this is an absolute physical segment to be located at the memory address defined by the expression. This expression will represent a paragraph number. For example, if the expression is 4444H, then the segment will be located at paragraph number 4444H or absolute memory address 44440H. The expression must evaluate to a constant (see Chapter 3). No forward references are allowed.

['classname']

The classname is used to indicate that segments are to be located (by LOC86) near each other in memory. This is not a means of combining segments so that they are addressable from the same segment register. The classname indicates that certain uncombined segments are to be put in the same general area in physical memory (for example, ROM).

Example:

The following two segments will be located adjacent to one another—

```
DATA1 SEGMENT BYTE 'ROM'      64K
          .
          .
          .
DATA1 ENDS

DATA2 SEGMENT BYTE 'ROM'
          .
          .
          .
DATA2 ENDS
```

Multiple Definitions for a Segment

You may "open" and "close" a segment (with SEGMENT and ENDS directives) within the module as many times as you wish. All "parts" of the segment which you define are treated together by the assembler as parts of one segment.

The following two occurrences of the segment DATA—

```
DATA SEGMENT PUBLIC

     ABYTE DB 0
     AWORD DW 0

DATA ENDS

DATA SEGMENT PUBLIC

     ANOTHERBYTE  DB 0
     ANOTHERWORD  DW 0

DATA ENDS
```

are equivalent to—

```
DATA SEGMENT PUBLIC

     ABYTE         DB 0
     AWORD         DW 0
     ANOTHERBYTE   DB 0
     ANOTHERWORD   DW 0

DATA ENDS
```

When you re-open a segment, you do not need to re-specify its attributes. However, you cannot change its attributes. The following is correct:

```
CODE SEGMENT BYTE PUBLIC
         .
         .
         .
CODE ENDS
  .
  .
  .
CODE SEGMENT
         .
         .
         .
CODE ENDS
```

The following will be flagged as an error:

```
DATA SEGMENT WORD 'ROM'
         .
         .
         .
DATA ENDS
  .
  .
  .
DATA SEGMENT BYTE 'ROM'
         .
         .
         .
DATA ENDS
```

"Nested" or "Embedded" Segments

Segments are never physically nested or embedded in memory. However, you may nest segment definitions in your program. This is only a lexical nesting and does not represent a physical nesting. For example, the following is a legal construct:

```
CODE SEGMENT          ;begin CODE
         .
         .
         .
    DATA SEGMENT      ;begin DATA, stop assembling CODE
             .
             .
             .
    DATA ENDS           ;end DATA, begin assembling CODE
                        ;again
          .
          .
          .
CODE ENDS               ;end CODE
```

The assembler will treat the CODE segment separate from the DATA segment. The contents of the DATA segment are not contained within the CODE segment. The following will be flagged as an error because SEGMENT/ENDS pairs must be matched as shown above:

```
CODE SEGMENT            ;begin CODE
          .
          .
          .
    DATA SEGMENT        ;begin DATA
                .
                .
                .
CODE ENDS               ;an error!!! Cannot close CODE before
                        ;closing DATA
          .
          .
          .
    DATA ENDS
```

The Default Segment—??SEG

All variables and instructions must lie within some segment at run-time. If you do not specify a segment to contain your code or data, the assembler will create a segment named ??SEG, in which this code or data will lie. This segment is non-combinable and paragraph aligned.

The ASSUME Directive

Syntax:

    ASSUME   segreg:segpart [, ...]

or

    ASSUME NOTHING

Description:

At run-time, every memory reference (a variable or label) requires two parts in order to be physically addressed: a paragraph number (segment part) and an offset (within the segment). The paragraph number will be in one of the segment registers, defining the physical segment in which the variable or label lies. (This value will have been placed in the segment register by the appropriate initialization code.) The offset value will be contained in the instruction which makes the reference. These two values are used to compute the absolute address of the object referenced. You use the ASSUME directive to define what the contents of the segment registers will be at run-time. This is done to help the assembler ensure that the code or data referenced will be addressable. The assembler will check each memory reference for addressability based on the contents of the ASSUME directive. The ASSUME directive does not initialize the segment registers; it is used by the assembler to help you to be aware of the addressability of the code and data. Unless the code or data is addressable (as defined either by an ASSUME or a segment override) the assembler will report an error. The ASSUME directive also helps the assembler decide when to automatically generate a segment override instruction prefix. (See Chapter 4 on the Segment Override Prefix.) The following example illustrates the use of ASSUME—

```
ASSUME  DS:DATA, CS:CODE     ;the DATA segment is
                             ;addressable through DS and
                             ;the CODE segment through CS
DATA  SEGMENT PUBLIC

ABYTE   DB  0
AWORD   DB  0

DATA ENDS

DATAX SEGMENT PUBLIC

WHERE   DB  0

DATAX ENDS

CODE  SEGMENT PUBLIC

        MOV   AX, DATA       ;AX = base address of DATA
        MOV   DS, AX         ;initialize DS

        MOV   AL, ABYTE      ;DS points to base of DATA
                             ;segment that contains ABYTE.
                             ;Instruction will use offset of
                             ;ABYTE to address value

ALAB:   MOV   AWORD, 15      ;CS points to base of CODE
        JMP   ALAB           ;CS initialized when program
                             ;loaded, instruction will use
                             ;offset of ALAB to calculate
                             ;jump

        MOV   AH, WHERE      ;AN ERROR!!!!! DS has not been
                             ;initialized with the base
                             ;address of the segment DATAX
                             ;and no ASSUME has been made!
                             ;The assembler does not know
                             ;where WHERE is.

        MOV   AX, DATAX
        MOV   ES, AX         ;initialize ES

ASSUME  ES:DATAX             ;DATAX now in ES

        MOV   AH, WHERE      ;assembler will automatically
                             ;assemble an ES instruction
                             ;prefix to address WHERE

CODE ENDS
```

Field Values:

segreg

One of the 8086/8088 segment register names: CS, DS, SS, or ES.

segpart

This field defines a paragraph number in one of the following ways:

1. A segment name, as in:

   ```
   ASSUME CS:CODE, DS:DATA
   ```

2. A previously defined group name (see page 2-8), as in:

   ```
   ASSUME CS:CODEGRP, DS:DATAGRP
   ```

3. An expression(see page 4-18) of the form:
   SEG variable-name or SEG label-name or SEG external-name,
   as in:

   ```
   ASSUME CS:SEG START, DS:SEG COUNT
   ```

4. The keyword NOTHING, which states that nothing is defined to be in that segment register at that time. If a segment register is assumed to contain NOTHING, the assembler will not generate instructions that use this segment register for memory addressing.

   Example:

   ```
   ASSUME ES:NOTHING
   ```

   The form ASSUME NOTHING is equivalent to:

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

   This is the default, which remains in effect until the first ASSUME directive is seen.

Forward Referenced Names in an ASSUME Directive

You may forward reference a name (i.e., refer to name not yet defined) in an ASSUME directive only if that name is the name of a segment. This is in the form:

```
ASSUME CS:CODE    ;The name CODE is a forward reference

CODE SEGMENT      ;CODE defined here
         .
         .
         .
CODE ENDS
```

If the name is not the name of a segment, an error will be reported.

Multiple ASSUME Directives

An ASSUME directive will stay in effect until it is changed by another ASSUME. That is, if you assume some contents in CS, that assumption will hold until you assume some new contents or NOTHING in CS.

The GROUP Directive

Syntax:

name GROUP segpart [,...]

Description:

The GROUP directive is used to combine several logical segments together, so that they will form one physical segment (i.e., they will all be addressable from the same base) after the program has been located. The size of the group is equal to the sum of the sizes of all the segments specified in the GROUP directive. The total size must be less than or equal to 64K bytes. The assembler will make no checks to see if the size of the group will be correct. This check is made by LOC86. The group name can be used as if it were a segment name (except in another GROUP directive). The order of the segments in the group directive will not necessarily be the order of the segments in memory after the program is located.

The GROUP directive serves as a "shorthand" way of referring to a combination of segments. Its utility is in specifying a collection of segments that are to be grouped at link-time to form one physical segment. However, the assembler views the program content in terms of segments. When you define a variable or label (see Chapter 3), the assembler assigns that variable or label to the segment in which it was defined. The offset associated with the variable or label is from the base of its segment and not from the base of the group.

One use of the group name is in the ASSUME directive. If, for example, you have defined many different data segments that you intend to form into one physical segment when the program is located in memory, you could combine these segments with the GROUP directive. Since the contents of all these data segments will be addressable through DS during the execution of the program, you may use the group name in the ASSUME and to initialize DS. For example,

```
DATAGRP GROUP DATA1, DATA2

DATA1   SEGMENT

ABYTE   DB  0

DATA1   ENDS

DATA2   SEGMENT

AWORD   DW  0

DATA2   ENDS

ASSUME  DS:DATAGRP, DS:CODE     ;use group name in ASSUME

CODE    SEGMENT
```

```
          MOV    AX, DATAGRP        ;AX = base address of group
          MOV    DS, AX             ;initialize DS

          MOV    AX, AWORD          ;addressable through DS

          .
          .
          .
CODE      ENDS
```

Field Values:

name

A unique ASM86 identifier that is used as the name for the group.

segpart

The field defines a paragraph number in one of the following ways:

1. A segment name, as in:

   ```
   CODEGRP GROUP CODE1, CODE2
   ```

2. An expression (see page 4-18) of the form:
   SEG variable-name or SEG label-name or SEG external-name,
   as in:

   ```
   DATAGRP GROUP SEG START, SEG COUNT
   ```

Use of the OFFSET Operator With Groups

When using the OFFSET operator (see page 4-18) with a variable or label whose segment is in a group, you must use the group name as a segment override (see page 4-14) in the expression, as in:

```
MOV BX, OFFSET DATAGRP:COUNT
```

Also, if you wish to store the paragraph number of a variable or label, you must use this construct:

```
DW DATAGRP:COUNT
DD DATAGRP:COUNT
```

Defining and Initializing Data 3

Overview of Variables and Labels

The two most referenced objects (other than registers) in a program are variables and labels. You define these objects in your program. Variables refer to data items, areas of memory where values are stored. Labels refer to sections of code that may be jumped to or CALLed. Each variable and label has a unique name in your program.

A variable is defined through a data definition statement or the LABEL directive. Each variable has three attributes:

1. Segment—The segment in which the variable was defined. It is a value that represents the paragraph number of the segment.
2. Offset—The offset (current location counter) of the variable defined. It is a 16-bit value which represents the distance in bytes from the base (or start) of the segment to the start of the variable in memory.
3. Type—The size of the data item in bytes. In most cases this type is specified through a keyword in the definition. The type of a variable determines how it may be used in an instruction and also, in some cases, how data will be stored within that variable. The possible types are:

    1. BYTE—one byte—8086/8088 data types.
    2. WORD—one word (two bytes)—8086/8088 data types.
    3. DWORD—one double-word (four bytes)—8086/8088 or 8087 data types.
    4. QWORD—one quad-word (eight bytes)—8087 data types.
    5. TBYTE—one ten-byte (ten bytes)—8087 data types.
    6. A structure—a multi-byte, "structured" 8086/8088 data type.
    7. A record—an 8 or 16 bit, "bit-encoded" 8086/8088 data type.

When you define a variable, the assembler will store its definition, which includes the above attributes. In Chapter 4, there is a discussion of expression operators that allow you to obtain or to override these attributes.

Labels define addresses for executable instructions. They represent a "name" for a location in the code. This "name" or label is a location that can be jumped to or CALLed. The label is an operand of the CALL, JMP, and conditional jump instructions. A label can be defined three ways: 1) a name followed by a ":" associated with an instruction statement, 2) a PROC directive, or 3) with a LABEL directive. Like a variable, a label has three attributes, two of which are the same as those for a variable:

1. Segment—same as variable.
2. Offset—same as variable.
3. Type—for a label, the type specifies the type of jump or CALL that must be made to that location. There are two types:

    1. NEAR—this type represents a label that will be accessed by a jump or CALL that lies within the same physical segment. This type of access is referred to as an intra-segment jump or CALL. In this case, only the offset part of the label is used in the jump or CALL instruction.

2. FAR—this type represents a label that will be accessed from another segment. In this case, because control is transferred from one physical segment to another, the contents of the CS register must be changed by the jump or CALL. A far label will be represented in the jump or CALL instruction by its offset and its segment part (to be loaded into CS).

A special form for defining a label is the PROC directive. This form specifies a sequence of code that will be CALLed just as a subroutine in a high-level language. The PROC directive defines a label with a type, either NEAR or FAR. It also defines a context for the RET instruction so that the assembler can determine the type of RET to code (either a near RET or a far RET). This construct can help to structure your programs into clearly defined subroutines. But, unlike high-level language procedures, there is no scoping of names and you can "fall into" an imbedded "procedure."

Constants

A constant is a pure number without any attributes. In general, a constant can be binary, octal, decimal, hexadecimal, ASCII, decimal real, or hexadecimal real. A constant can evaluate to one of three types: 8-bit, 16-bit, or real. These types cannot necessarily be used in the same context. You should verify the correct use of constants. The assembler will report an error if a constant is used incorrectly. The proper contexts for a particular type are noted throughout this manual. Table 3-1 gives the rules for forming each type of constant. A constant that can be represented in 8 or 16 bits has a special internal representation in the assembler. These constants are referred to as '17-bit numbers.' The maximum range of values for these numbers is −0FFFFH to 0FFFFH. All assembly time expressions use two's complement arithmetic on 17-bit numbers. Real constants (or floating point numbers) are restricted to DD, DQ, DT, and EQU directives. (For further information on the use of reals and the 8087 see *The 8086 Family User's Manual Numeric Supplement.*)

There is a special set of constants that are used in programming for the 8087. In general, these constants are referred to as "reals." The actual types are:

1. Short integer –four bytes.
2. Short real--four bytes.
3. Long integer—eight bytes.
4. Long real—eight bytes.
5. Packed decimal number—ten bytes.
6. Temp-real—ten bytes.

A short, long, or temp-real can be expressed in three ways:

1. Decimal real—without exponent.

```
1.234
3.14159
98.6
1234.4321
1.
```

2. Decimal real—with exponent.

```
6.8E27
1.23E-3
1E6
```

3. Hexadecimal real.

```
40490FDBR
0C0000000R
```

Integers (includes packed decimal) can be expressed in either binary, octal, decimal, or hexadecimal notation. The type of data allocation (the directive) you choose will affect the range of values that can be used in the initialization. These ranges are noted below under the appropriate directive.

Table 3-1. Constants

Constant Type	Rules for Formation	Examples
Binary (Base 2)	A sequence of 0's and 1's followed by the letter 'B'	11B 10001111B
Octal (Base 8)	A sequence of digits 0 through 7 followed by either the letter 'O' or the letter 'Q'	7777O 4567Q 77777O
Decimal (Base 10)	A sequence of digits 0 though 9, optionally followed by the letter 'D'	3309 3309D
Hexadecimal (Base 16)	A sequence of digits 0 through 9 and/or letters A through F followed by the letter 'H'. (Sequence must begin with 0-9)	55H 2EH 0BEACH 0FEH
ASCII	Any ASCII string enclosed in quotes (More than 2 chars. valid for DB only.)	'A', 'BC' 'UPDATE.EXT'
Decimal Real (Base 10)	A decimal fraction, optionally followed by an exponent. The fraction is a sequence of digits 0 through 9. A decimal point is required if no exponent is present and is optional otherwise The exponent starts with an E, followed by an optional sign and digits from 0–9.	3.1415927 .002E7 1E-32 1.
Hexadecimal Real (Base 16)	A sequence of digits 0-9 and/or letters A through F followed by the letter R. The sequence must begin with 0-9. Total number of digits must be (8, 16, 20) or (9, 17, 21). If odd numbered, the first digit must be 0.	40490FDBR 0C0000000R

Defining and Initializing Variables (DB, DW, DD, DQ, DT Directives)

Syntax:

1 byte initialization:

[name] DB init [, ...]

2 byte initialization:

[name] DW init [, ...]

4 byte initialization:

[name] DD init [, ...]

8 byte initialization:

[name] DQ init [, ...]

10 byte initialization:

[name] DT init [, ...]

Description:

The DB, DW, DD, DQ, and DT directives are used to define variables and/or initialize memory. When the directive is used with a name, it specifies a named variable whose segment part is the current segment and whose offset is the current location counter. Its type depends on the type of data initialization statement used. The variable can be initialized to a value, as in:

```
COUNT   DB   10       ;a variable initialized to 10
```

or it can simply reserve space with no specific initial value:

```
FLAGS   DW   ?        ;reserve a word
```

You may also use these directives to define the contents of memory when the program is loaded. To specify 10 bytes of 0, you might code

```
        DB   0,0,0,0,0,0,0,0,0,0
```

or

```
        DB   10 DUP (0)  ;a DUP is a repeated initialization
```

There are many types of values that can be used to initialize data. The following is a list of the possible types of initialization:

1. Constant expressions—a numeric value.

```
   TEN   DB 10
```

2. Indeterminate initialization.

```
   RESERVE   DW   ?
```

3. An address expression—the offset or base part of a variable or label.

```
   POINTER   DW   COUNT       ;store offset of COUNT

   SEGBASE   DW   DATA        ;store base address of DATA
                              ;segment

   APTR      DD   COUNT       ;store offset and segment part
                              ;of COUNT
```

4. An ASCII string of more than two characters—DB only.

```
   MESSAGE   DB   'HELLO THERE'

   MYHERO    DB   'ALEISTER CROWLEY'
```

5. A list of initializations.

```
   STUFF     DB   10, 'A STRING', 0, 'Q'

   NUMBS     DW   1, 2, 3, 4, 0FFFFH
```

6. A repeated initialization, where the quantity in the '( )' is repeated 'number DUP' times.

```
TENS      DB  10 DUP (10)

PATTERN   DW  100 DUP (0,1,65535)
```

When a number is stored in 16 bits, it is stored with its low-order byte preceding the high-order byte in memory. For example, if you were to code

```
DW    1234H
```

it would be stored as

```
34  12
low high
———————→   increasing memory addresses
```

in memory. If you specify a string in a DB directive it will be stored with one ASCII character per byte in the same order as the characters appear in the string.

```
DB   'ABC'
```

is stored as

41 42 43

in memory.

Field Values:

[name]

A unique ASM86 identifier. It defines a variable whose offset will be the current location counter. Its type will be the type of the data initialization unit. Its length will be equal to the number of bytes initialized.

init

There are many possible values for init depending on the usage and context. Init has five possible types, listed below. The form used will depend on what type of initialization you wish to perform. The different forms and contexts are noted below.

1. A constant expression.
   a. 1 byte initialization—a constant or expression that evaluates to 8-bits (i.e., −255 to +255 decimal).
   b. 2 byte initialization—a constant or expression that evaluates to 16-bits (i.e., −65535 to +65535 decimal).
   c. 4 byte initialization—
      1. A constant or expression that evaluates to 16-bits (a 17-bit number). The upper 16 bits are sign-extended in assemblers that support the 8087, else they are initialized to 0H.
      2. Short integer in the range $-2^{32}+1$ to $+2^{32}-1$, which is −4 294 967 295 to +4 294 967 295
      3. Real in the range -2^{128} to -2^{126}, 0, $+2^{126}$ to $+2^{128}$, or approximately −3.4E38 to −1.2E −38, 0, 1.2E −38 to 3.4E38.

d. 8-byte initialization—

1. Long integer in the range $-2^{64}+1$ to $+2^{64}-1$, which is −18 446 744 073 709 551 615 to +18 446 744 073 709 551 615.
2. Real in the range -2^{1024} to -2^{-1022}, 0, $+2^{-1022}$ to $+2^{1024}$, or approximately −1.7E308 to −2.3E −308, 0, 2.3E −308 to 1.7E308.
3. A constant (17-bit number), which will be sign-extended to fit in a DQ field.

e. 10-byte initialization—

1. Long integer in the range $-10^{18}+1$ to $+10^{18}-1$, which is −999999999999999999 to +999999999999999999. The number will be stored as packed decimal (BCD) format.
2. Real in the range -2^{16384} to -2^{-16382}, 0, $+2^{-16382}$ to $+2^{16384}$, or approximately −1.1E4932 to −3.4E−4932, 0, 3.4E−4932 to +1.1E4932.

2. The character '?' for indeterminate initialization.

In situations where you wish to reserve storage but do not need to initialize that area to any particular value, you can use the special character "?". This character specifies that the area will be reserved. The reserved area will be initialized with an indeterminate value. It can be used with any of the data initialization directives.

```
ABYTE   DB  ?     ;reserve a byte
AWORD   DW  ?     ;reserve a word (2 bytes)
ADWORD  DD  ?     ;reserve a dword (4 bytes)
AQWORD  DQ  ?     ;reserve a qword (8 bytes)
ATBYTE  DT  ?     ;reserve a tbyte (10 bytes)
```

When used in a special DUP construct, "?" can be used to specify no initialization (see below).

3. Initializing with an address-expression—DW and DD only.

You can initialize a DW or DD with a variable name, label name, segment name, or group name. When you use a variable or label name in a DW, you are initializing with the offset of that variable or label.

```
DW  COUNT              ;store the offset of COUNT
                       ;from its segment
DW DATAGRP:COUNT       ;store the offset of COUNT
                       ;from its group (DATAGRP)
```

Using a segment name or group name in a DW will store the paragraph number of that item.

```
DW  CODE       ;store the paragraph number of CODE
               ;segment
```

In a DD, the use of a variable or label name will store the offset of the variable or label in the lower order word and the segment part (paragraph number) in the higher order word. This forms a pointer to that item.

```
DD  COUNT      ;store a pointer to COUNT
```

which is equivalent to:

```
DW  COUNT          ;store the offset of COUNT
DW  SEG COUNT      ;store the paragraph number of
                   ;COUNT's segment
```

Use of segment or group name in a DD will store the paragraph number in the low order word and initialize the higher order word with 00H.

4. Initializing with a string—DB only.

   In a DB you can define a string up to 255 characters long. Each character is stored in a byte, where successive characters occupy successive bytes. The string must be enclosed with single quotes. If you wish to include a single quote in a string, code it as two consecutive quotes. Examples are given below.

   ```
   ALPHABET  DB  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

   DIGITS    DB  '0123456789'

   WITHQUOTE DB  'THIS AIN"T HARD!''    ;inserting quote
                                        ;in string
   ```

   In a DW and DD you may code a string of either one or two characters. In this case, the string is interpreted to be a number and it will be stored as a number. For example.

   ```
   NUMB  DW  'AB'
   ```

   is equivalent to

   ```
   NUMB  DW  4142H
   ```

   where the low byte is stored first followed by the high byte. The same convention is true for a DD. In that case, the upper 16 bits will be initialized to 00H.

5. Initializing with a repeated value.

   There is a special construct that can initialize an area of memory with a repeated value or list of values. The form for this construct is given below.

   repeatval   DUP   (val   [, ...]   )

   "Repeatval" is a positive number from 1 to 65535. It specifies the number of data initialization units to be initialized (bytes, words, dwords, qwords, or tbytes). "Val" may be any of the following:

   1. An expression—either numeric (appropriate to the data initialization unit) or an address-expression for a DW or DD.
   2. A "?" for indeterminate initialization. If the special form

      ```
      DB   repeatval   DUP   (?)    or
      DW   repeatval   DUP   (?)    or
      DD   repeatval   DUP   (?)    or
      DQ   repeatval   DUP   (?)    or
      DT   repeatval   DUP   (?)
      ```

      is used, then no data initialization record will be produced in the object module, but the area will be reserved. Any other use of the "?" will cause a data initialization record to be produced, but the value used for initialization will be indeterminate.

      As an example:

      ```
      WORD1    DB   2 DUP (?)
      WORD2    DW   1 DUP (?)
      ```

      will both reference word variables without initializing data, whereas

```
WORD3    DW   ?
WORD4    DB   1 DUP (?,?)
WORD5    DB   1 DUP (?), 1 DUP (?)
WORD6    DB   2 DUP (1 DUP (?))
```

will all initialize words to an indeterminate value.

3. A string where the data initialization unit is a DB.

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

4. A list of the above items following the rules given above for each item.

```
STRINGS    DB    10  DUP ('HELLO','GOODBYE')
ADDEXPS    DW     3  DUP (COUNT, START, NEXT)
NUMBS      DD   100  DUP (1, 0FFFFH, 15 10101010B)
DIFFERENT  DB    25  DUP (2, 'NSJRAJ', 3)
```

5. "Val" may also be another DUP statement, following again all the above rules. DUP's may be nested up to eight levels deep.

```
MORESTRINGS  DB  15  DUP ('HELLO', 3 DUP ('GOODBYE'))
MORENUMBS    DW  27  DUP (1, 3, 5, DUP (2, 5, 7))
NESTEDDUP    DB   3  DUP (4 DUP (5 DUP (1, 6 DUP (0))))
```

Introduction to Records

ASM86 has a special data initialization statement that allows you to construct bit-encoded data structures called records. A record may be either 8 or 16 bits in size. Each record is defined to have a number of fields containing a certain number of bits per field. You can store information in these fields and also access that information. Records are useful where you wish to access specific bits in a data structure. These could be flag bits, fields in a data structure used to store a real number, etc. There are special operators used to access the fields in a record. These are discussed in Chapter 4. There are two steps in using a record. The first defines a "template" for the record. This specifies the size of the record and its fields. The second step uses the record name in a data initialization statement to actually allocate the storage. These steps are described below.

The RECORD Directive

Record Template Definition

Syntax:

name   RECORD   field-name:exp [= inival]   [, ...]

Description:

A record is a bit pattern you define in order to format bytes or words for bit-packing. A record can be from 1 to 16 bits in size. Records are first defined through the Record Template Definition. Data can then be allocated and initialized through the use of the record name in a data initialization statement (given below). Some examples:

```
ERRORFLAGS    RECORD  IOERR:3=0, SYSTEMERR:3=0, MEMERR:3=0

SIGNEDNUMB    RECORD  SIGN:1, NUMBER:15
```

Field Values:

name

This is a unique ASM86 identifier, which is the name for the record template defined.

field-name

This is a unique ASM86 identifier, which defines a bit field within the record.

exp

This is a constant or expression that evaluates to be a number in the range 1 to 16. This value specifies the number of bits in the field. (If a symbol is used in an expression, it must not be a forward reference.) The sum of the "exp's" in a record definition must not exceed 16; if they do, an error will be reported.

[= initval]

This is a constant or expression that evaluates to a number that can be represented by the number of bits defined for that field. This optional clause defines a default value for the field. If no initval is specified, the default value is zero. This default value can be overriden during allocation and initialization.

"Partial" Records

A "partial" record is a record that does not fully occupy a byte or word. The assembler will right-justify the fields within the record in the least significant bit-positions of the byte or word defined by the record. The undefined (unallocated) bits have a value of zero when the record is used to allocate storage. If you defined a record as below

```
QUASI    RECORD  A:6, B:6
```

it will be formatted as follows:

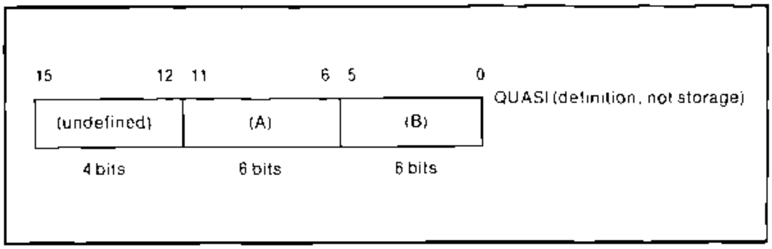


Figure 3-1. "Partial" Record Definition

Record Allocation and Initialization

Syntax:

[name] record-name <[exp] [, ...] >

or

[name] record-name repeat-val DUP (<[exp] [, ...] >)

Description:

Use of this form will allocate data in the form specified by the record template used. You may override any default values given in the record definition. For example,

```
FLAGS      ERRORFLAGS <0,3,0>

FLAGS1     ERRORFLAGS <>        ;no initialization overrides

PLUSONE    SIGNEDNUM8 <0,1>

MINUS15    SIGNEDNUM8 <1, 15>
```

Field Values:

[name]

A unique ASM86 identifier that is a name for the byte or word allocated.

record-name

This is the name of the previously defined record template that defines how the bits within the byte or word are to be allocated.

[exp]

You may optionally override default values of record fields when you allocate the storage. The "exp" must evaluate to a number that will fit in the number of bits in the field you wish to override. You may override all, some, or none of the fields in the record template. The following rules apply:

For a record with N fields, each field is represented in the allocation statement, as shown below—

<f1,f2,f3,...,fn>

To override a particular field, place the value in the position of that field in the allocation statement. To override "f3" you would code

<,,2>

To override "fn" you would code

<,,,...,2>

Each "empty" override (the ",") specifies one field; you can skip fields up to the field you wish to override just by typing a "," for that field. You do not need to type anything for fields after the one you wish to override if you are not specifying any values for them. To allocate a record with no overrides you simply code:

< >

repeatval

A positive integer that indicates the number of records to be allocated.

Introduction to Structures

You can define "structured" data blocks built from the basic types of data initialization statements. These data blocks are called "structures." A structure is composed of data initialization statements that define the fields within a block of data. Each of the fields can be separately accessed. For example, if you wish to define a data structure that contains a complex number that has two fields, you could code the following:

```
COMPLEX    STRUC

REALPART       DQ  0
COMPLEXPART    DQ  0

COMPLEX    ENDS
```

This code defines a template that can then be used to allocate storage. To store the complex number 1.2 * 3.5i, you would code the following:

```
VALUE   COMPLEX   <1.2, 3.5>
```

To perform any calculations based on this value, you would refer to the fields of the structure as

```
VALUE.REALPART
```

and

```
VALUE.COMPLEXPART
```

in the instruction (see Chapter 4).

The STRUC Directive

Structure Template Definition

Syntax:

```
    name   STRUC
             .
             .
             .
   [fieldname]   data-init
             .
             .
             .
    name   ENDS
```

Description:

A structure is a "structured" data type. This is similar to a "record" data type in Pascal, except that the type of elements you may define for a structure are restricted to the data types allowed in ASM86, (i.e., byte, word, dword, qword, and tbyte). A STRUC/ENDS pair defines a storage template with various subfields of possibly different types. This template can then be used to allocate data based on the "structure" of the template. You may define values for the fields that can then be overriden (with some exceptions) when the structure is used to initialize storage. An example of a structure is shown below.

```
EMPLOYEE    STRUC

EMPNAME     DB '                    ';20 chars allowed
HOURRATE    DD 5.60                  ;dollars per hour
NUMBHOURS   DB ?                     ;hours per week to be used

EMPLOYEE    ENDS
```

This structure template could then be used to create data structures for different employees. You can override the initial values when the data is allocated and you may programmatically change the values in allocated structures (see Chapter 4).

Field Values:

name

A unique ASM86 identifier that is the name for the structure template defined.

fieldname

A unique ASM86 identifier. This name will be used to access the fields within an allocated structure. It represents an offset from the base of the allocated structure. In the example above, the field HOURRATE would have an offset of 20 from the beginning of the structure. This value (expressed by the fieldname) is used in instructions to access the field. (See Chapter 4.) A fieldname has the following attributes:

segment—none
offset—number of bytes from start of structure
type—type of init

data-init

This may be any allowed data initialization statement (DB, DW, DD, DQ, or DT). Refer to the section "Defining Variables" for the details on all the allowed forms.

Structure Allocation and Initialization

Syntax:

[name] structure-name < [exp] [, ...] >

or

[name] structure-name repeatval DUP (< [exp] [, ...] >)

Description:

Use of this statement will allocate storage based on the structure template used. The amount of storage allocated will be a function of the number of bytes defined in the template. Initial values in the fields may be overriden with certain restrictions (see below). An array of structures can be allocated by using the form with a "DUP". For example,

```
ACOMPLEXNUMB  COMPLEX  <1.6, 7.8>

JONES  EMPLOYEE  <'JONES, SAM', 2.00, 60>

PEOPLE EMPLOYEE  20 DUP (<>)
```

Field Values:

[name]

A unique ASM86 identifier. This name will define a variable whose segment part will be the current segment and whose offset will be the current location counter. Its type will be an integer equal to the number of bytes allocated by the template.

structure-name

A name of a previously defined structure template.

repeatval

A positive integer that indicates the number of structures to be allocated.

exp

This field is a value that will override the default value given in the template definition. Its type must match the type of the field. It may be either a constant, an expression, a string, or the indeterminate initialization character, "?". The value can only be used to override fields that meet the following restrictions:

1. The field specified in the structure template definition cannot be a list of values or a DUP clause.
2. A DB that is initialized with a single string of two or more characters can be overriden only with another string. If the overriding string is shorter than the original string, the remaining characters of the default string are used. If the overriding string is longer, it is truncated.
3. The value must fit within the field you wish to override.

   Example of overridable fields—

```
OVERRIDABLE   STRUC

ASTRING       DB   'ABCDEFG'
DONTCARE      DW   ?
AREAL         DD   3.14159

OVERRIDABLE   ENDS
```

   Example of non-overridable fields—

```
NONOVERRIDE   STRUC

ALIST         DB   1, 2, 3
ADUP          DW   10 DUP (?)

NONOVERRIDE   ENDS
```

For a structure with N fields, each field is represented in the allocation statement as shown below—

<f1,f2,f3,....fn>

To override a particular field, place the value in the position of that field in the allocation statement. To override "f3" you would code

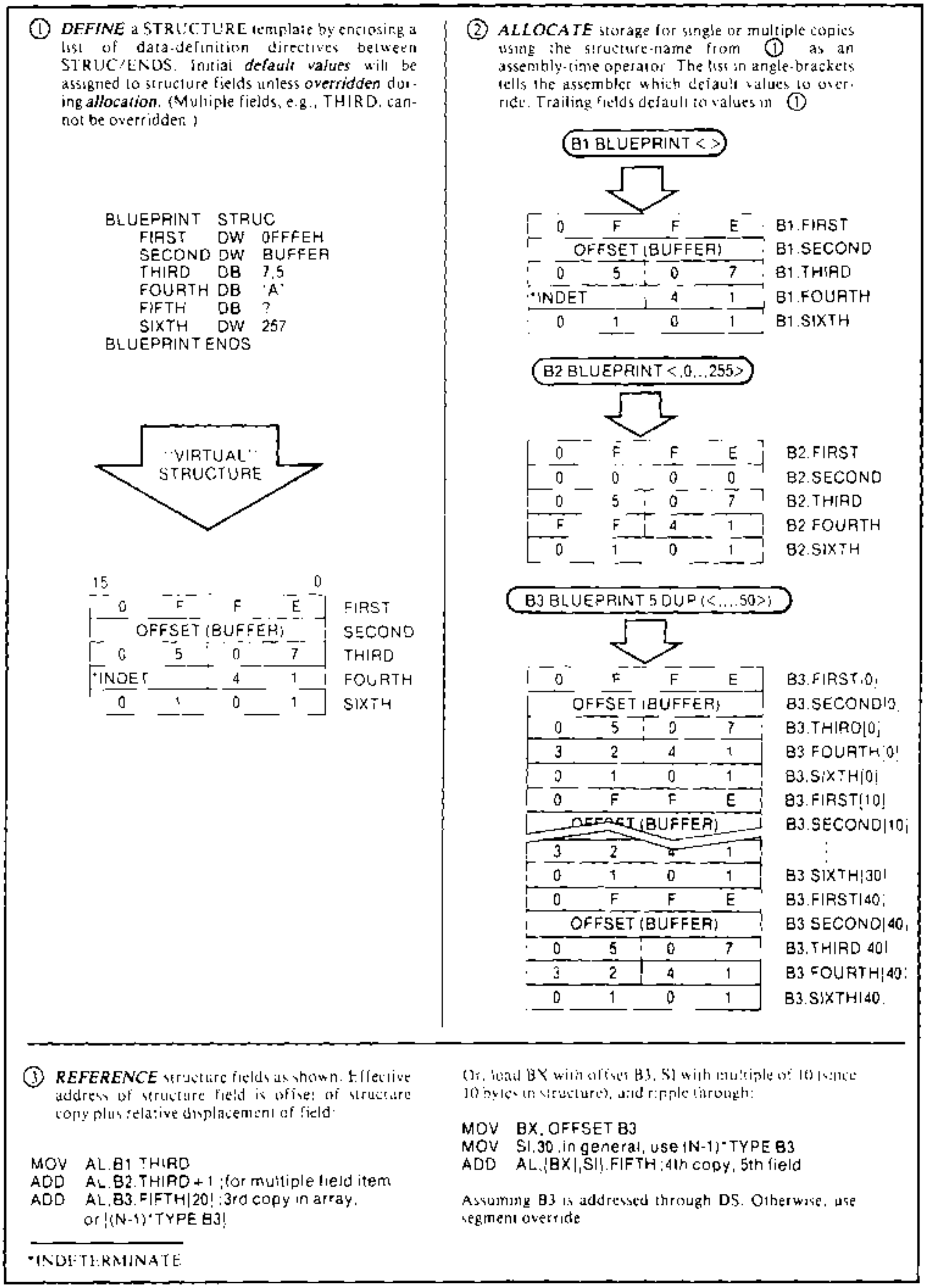
<,,2>

To override "fn" you would code

<,,,...,2>

Each "empty" override (",") specifies one field. You can skip fields up to the field you wish to override just by typing a "," for that field. You do not need to type anything for any fields after the one you wish to override if you are not specifying any values for them. To allocate a structure with no overrides you simply code:

< >



① ***DEFINE*** a STRUCTURE template by enclosing a list of data-definition directives between STRUC/ENDS. Initial ***default values*** will be assigned to structure fields unless *overridden* during ***allocation***. (Multiple fields, e.g., THIRD, cannot be overridden.)

```
BLUEPRINT   STRUC
     FIRST    DW   0FFFEH
     SECOND DW   BUFFER
     THIRD    DB   7,5
     FOURTH DB   'A'
     FIFTH     DB   ?
     SIXTH    DW   257
BLUEPRINT ENDS
```

② ***ALLOCATE*** storage for single or multiple copies using the structure-name from ① as an assembly-time operator. The list in angle-brackets tells the assembler which default values to override. Trailing fields default to values in ①

③ ***REFERENCE*** structure fields as shown. Effective address of structure field is offset of structure copy plus relative displacement of field:

```
MOV   AL,B1.THIRD
ADD   AL,B2.THIRD+1 ;for multiple field item
ADD   AL,B3.FIFTH[20] ;3rd copy in array,
      or [(N-1)*TYPE B3]
```

Or, load BX with offset B3, SI with multiple of 10 (since 10 bytes in structure), and ripple through:

```
MOV   BX, OFFSET B3
MOV   SI,30 ;in general, use (N-1)*TYPE B3
ADD   AL,[BX][SI].FIFTH ;4th copy, 5th field
```

Assuming B3 is addressed through DS. Otherwise, use segment override

*INDETERMINATE

Figure 3-2. Structure Definition and Allocation

Defining Labels

A label, a symbolic name for a particular location in an instruction sequence, may be defined in one of three ways. The first way is the most common. The format is shown below:

```
label:  [instruction]
```

where "label" is a unique ASM86 identifier and "instruction" is an 8086/8087/8088 instruction. This label will have the following attributes:

1. Segment—the current segment being assembled.
2. Offset—the current value of the location counter.
3. Type—will be NEAR.

An example of this form of label definition is:

```
ALAB:  MOV AX, COUNT
```

The second means of defining a label is the PROC directive. This can be used to define either a near or far label. The third means is the LABEL directive. (Do not confuse the use of the term "label" with the name of this directive.) Either a near or far label can be defined. See below for a discussion of the PROC and LABEL directives.

The PROC Directive

Syntax:

```
name   PROC   [type]
         .
         .
         .
name   ENDP
```

Description:

A PROC directive is used to define a label and to delineate a sequence of instructions that are usually interpreted to be a subroutine, that is, CALLed either from within the same physical segment (near) or from another physical segment (far). The primary use of the PROC directive is to give a type to the RET instruction enclosed by the PROC/ENDP pair. A PROC is different from a high-level language subroutine or procedure. There is no scoping of names in a PROC. All user defined variables and labels in a program must be unique. Also, there is no "block structuring" of PROC's. If a PROC is defined within a PROC, execution can "fall into" the PROC. For example

```
P1     PROC NEAR

       MOV  AX, 15       ;execution begun here will
       ADD  DX, AX       ;continue through to the MOV AX, 0

P2     PROC NEAR

       MOV  AX, 0
       CMP  AX, COUNT
       JE   LAB
```

```
      SUB  COUNT, 1
LAB:  MOV  AX, 0
      RET                 ;exit P1 and P2 here!

P2    ENDP

      CMP  DX, 10         ;never will be executed!!!
      JE   LAB
      RET

P1    ENDP
```

The 8086/8088 has two types of RET instructions, either near or far, that must correspond to the type of CALL made. Given below is an example of both a near and a far PROC, each with their appropriate CALL.

Example 1—A NEAR PROC.

```
LOCALCODE  SEGMENT  PUBLIC

ANEARPROC  PROC  NEAR
             .
             .     ;some code
             .
           RET     ;will be near RET
ANEARPROC  ENDP
             .
             .
             .
           CALL ANEARPROC     ;a near CALL
             .
             .
             .
LOCALCODE  ENDS
```

Example 2—A FAR PROC.

```
GLOBALCODE SEGMENT WORD
             .
             .
             .
AFARPROC   PROC  FAR
             .
             .     ;some code
             .
             .
           RET     ;will be a far RET
AFARPROC   ENDP

GLOBALCODE ENDS

SPECSEG    SEGMENT BYTE
             .
             .
           CALL AFARPROC     ;will be a far call
                             ;intersegment
             .
             .
SPECSEG    ENDS
```

Field Values:

name

This is a unique ASM86 identifier that defines a label whose segment attribute is the current segment, and whose offset is the current location counter. Its type is defined in the PROC directive.

type

This field specifies the type for the label defined. The possible values are:

1. None specified—defaults to NEAR.
2. NEAR—to define a near label.
3. FAR—to define a far label.

This field will specify to the assembler what type of CALL instruction to generate for the procedure and what type of RET instruction to code for any RET instruction found between the PROC/ENDP pair.

The LABEL Directive

Syntax:

```
name   LABEL   type
```

Description:

The LABEL directive creates a name for the current location of assembly, whether data or code. You use the LABEL directive to define a variable or a label that will have the following attributes:

1. Segment—the current segment being assembled.
2. Offset—the current offset within that segment.
3. Type—the operand to the LABEL directive.

The LABEL directive is useful for defining a different name with possibly a different type for a location that is named through the usual means. For example, if you desire to access two consecutive bytes as both a word and as two different bytes, the following usage of the LABEL directive will allow both forms of access.

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

It can also be used to define two labels of different types for the same location of code. This is useful if a section of code is to be called both near and far. (The programmer must be careful in this case to insure that the right RET is executed for the type of CALL made.) The following (potentially deadly) example illustrates this use.

```
AFARLABEL  LABEL  FAR
NEARLAB:   MOV    AX, BX
```

Field Values:

name

A unique ASM86 identifier.

type

This field identifies the type that is to be assigned to this name and location. It can specify a variable or a label depending on the type. This field can have the following values:

1. BYTE—defines a variable of type byte.
2. WORD—defines a variable of type word.
3. DWORD—defines a variable of type dword.
4. QWORD—defines a variable of type qword.
5. TBYTE—defines a variable of type tbyte.
6. A structure name—the type will be equal to the number of bytes allocated by the structure.
7. A record name--the type will either be a byte or word depending on the size of the record.
8. NEAR—defines a label of type near.
9. FAR—defines a label of type far.

The Location Counter ($)

The location counter keeps track of the current offset within the current segment that is being assembled. This value is symbolized by the character "$", which may be used in certain contexts, (i.e., expressions or instructions) (see Chapter 4). This symbol represents a near label, whose attributes are:

segment—current segment
offset —current offset
type—near

The assembler will maintain the correct offset within a segment even if the segment is repeatedly "opened" and "closed" in the module with the appropriate SEGMENT/ENDS pairs.

The ORG Directive

Syntax:

ORG exp

Description:

The ORG directive allows you to control the location counter within the current segment. You use the ORG directive to set the location counter to the desired value. Be careful in the use of this directive not to overwrite any previously allocated data or code by ORGing to a location previously allocated. The ORG directive is used to locate code or data at a particular location (offset) within a segment. Used with an absolute segment, you can specify the actual location in memory in which the code or data will be located.

Field Values:

exp

This is an expression that is evaluated modulo 65536. The expression must not include any forward references. You may use the value of the current location counter, "$" in an expression, such as:

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

Avoid expressions of the form

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

since this will overwrite your last 1000 bytes of assembly (or will re-ORG high in the current segment if the expression evaluates to a negative number).

The EVEN Directive

Syntax:

EVEN

Description:

The EVEN directive ensures that the code or data following the use of the directive will be aligned on a word boundary. For 8086 data, this may result in a faster fetch-time. The assembler will insert a NOP (90H) in front of the code or data, if it is necessary, to force the word alignment. The EVEN directive cannot be used in a byte aligned segment—an error message will be issued.

The PURGE Directive

Syntax:

PURGE name [, ...]

Description:

The PURGE directive deletes the definition of a specified symbol, allowing the symbol to be redefined. All occurrances of the symbol following the PURGE directive and the redefinition of the symbol will use the new definition. It will remain undefined after it is purged unless it is redefined. A reference to a symbol after a purge, but before a redefinition is a forward reference to the redefinition. If no redefinition occurs, the reference will cause an error. The following types of symbols cannot be purged—

1. Register names
2. The symbol ??SEG.
3. Hands-off keywords (see list in Appendix C).
4. A symbol that appears in a PUBLIC statement.

Using the PURGE Directive to Control Debug Information

The PURGE directive can be used to control the symbol information placed in the object module by the assembler when the DEBUG control is specified. (See the ASM86 macro assembler manual for a description of the DEBUG control. If you do not wish to have information placed in the object module for certain symbols, you can purge those symbols at the end of the program just before the END statement.

Accessing Data—Operands and Expressions 4

8086/8087/8088 Instruction Statements

Syntax:

[label:] [prefix] mnemonic [operand [, operand]]

Description:

The instruction statements form the core of an assembly language program. These statements define the actual program that the CPU (and NDP) will execute. This chapter describes the operands used in the assembly language. The 8086/8087/8088 instruction set is defined and discussed in Chapter 6. The operand field specifies the object of the machine operation. For a two operand instruction, one of the operands is considered a destination operand and the other is the source operand. This form is given below.

INSTRUCTION   DESTINATION, SOURCE

Some examples, shown below, illustrate some instruction statements:

```
        MOV  AX,  0      ;place 0 into AX

        ADD  CL,  DL     ;CL = CL + DL

ALAB:   REP  MOVSB       ;with prefix instruction and label
```

Refer to Chapter 6 for the use of the Prefix instructions.

Field Values:

[label:]

A unique ASM86 identifier, followed by a colon, that is used to define a label. (See Chapter 3.)

[prefix]

An 8086/8088 Prefix instruction, i.e., LOCK and REP instructions. (See Chapter 6.)

mnemonic

An 8086/8088 or 8087 instruction. These are fully described in Chapter 6.

operand

There are many possible types of operands, including registers, constant values, variables, and labels. The operand you specify will depend on the instruction coded. All of the various operand types are discussed below.

Operand Types

Registers

The 8086/8088 registers can be used as explicit operands to many instructions. In two-operand instructions they may be used for both source and destination. The register set is shown below.

Segment Registers:

CS, DS, SS, ES

General Registers (16 Bits):

AX, BX, CX, DX, SP, BP, SI, DI

General Registers (8 Bit):

AL, AH, BL, BH, CL, CH, DL, DH

Pointer and Index Registers:

BX, BP, SI, DI

The different sets overlap. Each of the general registers (8 and 16 bit) can participate in arithmetic and logical operations. The Pointer and Index registers are also used in certain address modes (see Register Expression section below). The segment registers can be used in MOV's, PUSH's, and POP's.

Floating Point Stack

The 8087 has it's own set of 'registers' called the floating-point stack. There are eight stack elements that can be referenced. The form is ST(i), where 'i' refers to the element 0 through 7. The top-of-stack is always ST(0), which may be abbreviated as ST.

Immediate Operands

An immediate operand is a constant value (number). This is a "17 bit" number (see Chapter 3). Immediate operands are used as source operands in an 8086/8088 instruction statement. For example,

```
MOV  AL, 5          ;AL = 5

CMP  AX, 0FFFFH     ;compare AX to 0FFFFH
```

An immediate operand can also be an expression that evaluates to a number. This chapter discusses all the types of expressions.

Examples of expressions as immediate operands:

```
CMP  AL, 15 OR 5              ;an expression example--compare
                              ;AL with 15

ADD  DX, (23 * 2) / 10        ;add 4 to DX
```

Memory Operands

A memory operand refers to a particular location in memory. The general term for a memory operand is an "address expression." An address expression may be a simple variable or label name, or it may involve registers, structure fields, and/or constants. Each address expression will reflect a particular addressing mode. The 8086/8088 has many different types of addressing modes. They are:

Direct Address

The operand is a simple variable or label name. The name expresses the offset of the operand that is used to calculate the address.

```
MOV  AX, COUNT      ;move the word value at memory location
                    ;COUNT into AX

JMP  ALAB           ;jump to memory location ALAB
```

Register Indirect Address

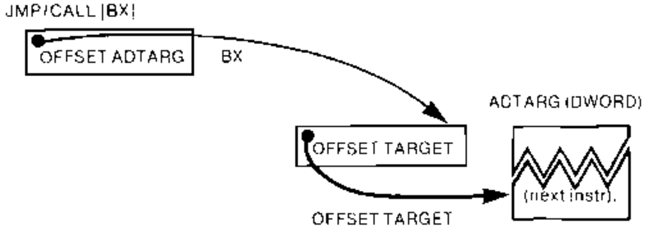
In this case the offset of the memory location is contained in one of the pointer or index registers (BX, BP or SI, DI). To address the location you must first load the offset into the register and then use the register name in brackets as the operand. For example, to indirectly address a variable you would code the following:

```
MOV  BX, OFFSET AVAR

MOV AX, [BX]           ;AX = contents of AVAR
```

A JMP or CALL instruction can use any 16-bit general register for indirect addressing.

```
MOV  AX, OFFSET ALAB
JMP AX                   ;no [] are needed here!--simple
                         ;indirect jump to ALAB

MOV  TARG, OFFSET ALAB
MOV  BX, OFFSET TARG
CALL  [BX]               ;[] used here (a register
                         ;expression)!--two level indirect
                         ;jump to ALAB
```

The two levels of indirection in JMP/CALL [BX] are schematically depicted as follows:



Based Address

The base address mode is similar to register indirect mode except that, in this case, a displacement is added to the contents of the register. With this mode the register can point to the base of a data structure in memory and the displacement can then be used to access a field within the data structure.

```
MOV  BX, OFFSET DATASTRUC      ;BX = base of DATASTRUC
MOV  AX, [BX + 5]              ;AX = word located at the
                               ;fifth byte of DATASTRUC
```

Based addressing is typically used with either BX or BP as the base register though SI and DI may also be used. The displacement may be either 8 or 16 bits.

Indexed Address

Indexed addressing is similar to based addressing except that the registers SI or DI are used along with a variable name. These registers are used as an index from the offset represented by the variable name. The contents of the register used as an index specifies a byte displacement from the offset of the variable. You may also use a displacement value in the operand.

```
        MOV  SI, 0                ;set indices to 0
        MOV  DI, 0
        MOV  CX, LENGTH SOURCE
ALAB:   MOV  AX, SOURCE[SI]       ;indexed address
        MOV  DEST[DI], AX
        INC  SI
        INC  SI                   ;index next word in SOURCE
        INC  DI
        INC  DI                   ;index next word in DEST
        LOOP ALAB
```

Based Indexed Address

This mode uses the contents of a base register (BX, BP), the contents of an index register (SI, DI), and an optional displacement. With this mode you may point the base register at the base of a data structure and then use the index register as an index into that structure.

```
        MOV  BX, OFFSET ARRAYSTRUC      ;load base address

        MOV  SI, 0                      ;index value

ALAB:   MOV  AX, [BX + SI]              ;get element

           .
           .
           .
        ADD  SI, 2                      ;increment index
        JMP  ALAB
```

Segment Register Defaults

Variable references such as:

```
[BX]
[BP]
WORD PTR [DI]
[BX].FIELDNAME
BYTE PTR [BP]
```

are termed "anonymous references" because no variable name is given from which a segment can be determined. (The structure field in the fourth example has a type and offset, but no segment associated with it.)

Segment registers for anonymous references are determined by hardware defaults, unless you explicitly code a segment prefix operator. The hardware defaults are:

- [BX] normally defaults to segment register DS
- [BP] normally defaults to segment register SS
- When an index register is used without a base register (as in WORD PTR [DI] or [SI + 5]), the default segment register is DS
- When an index register is used with a base register (as in [BP][SI] or BYTE PTR [BX][DI]), the default segment register is that of the base register (SS or DS, in these cases).

There are two variable-referencing exceptions for defaults:

1. Operations which implicitly reference the stack (PUSH, POP, CALL, RET, INT, and IRET) always use SS, and cannot be overridden. (The construct [SP] is not an addressing mode, and thus you cannot assemble e.g. MOV [SP], BX, much less override it.)
2. String instructions always use ES as a segment register for operands pointed to by DI.

Special care must be taken to ensure that the correct segment is addressed when an anonymous offset is specified. Unless you code a segment prefix override, the hardware default segment will be addressed, and the anonymous offset applied to it.

Thus, if a programmer's declared variables all reside in segment SEG1:

```
SEG1   SEGMENT
        .
        .
        .
FOO    DW  500 DUP (0)   ; 500 words of 0's
        .
        .
        .
SEG1   ENDS
```

and if his ASSUME directive in segment CODE1 is as follows:

```
ASSUME CS:CODE1, DS:SEG1
```

then all references to named variables in segment SEG1 will assemble correctly. But suppose our programmer elects to use BP as an index register to access elements of FOO in SEG1, as follows:

```
MOV BP, OFFSET FOO        ; Load BP with offset of FOO in
                          ; SEG1.
MOV AX, [BP]              ; Put first word of FOO into AX.
                          ; No assembly-time error, but wrong
                          ; seg-reg (SS instead of DS) at
                          ; run-time.
```

Because no variable name is present (for ASSUME to check), and because no segment override prefix is specified, the [BP] reference, by default, specifies an offset address that will be combined with the SS segment register, and not the DS, as intended. The code should read:

```
MOV BP, OFFSET FOO        ;Load BP with offset of FOO in
                          ;SEG1.
MOV AX, DS:[BP]           ;Use DS seg-reg for DATA1, put
                          ;first word of FOO into AX.
```

Overview of Expressions

An expression can define a value that initializes data or is used as an operand to an instruction. An expression can specify a numeric value or define an address in memory that will then serve as an instruction operand. There are many different kinds of operators that you may use to create expressions:

- Arithmetic Operators

  high and low (HIGH, LOW)
  addition and subtraction (+, −)
  multiplication and division (*, /, MOD)
  shifting operators (SHR, SHL)
- Logical Operators (AND, OR, XOR, NOT)
- Relational Operators (EQ, LE, LT, GE, GT, NE)
- Attribute operators

  attribute overriding operators (segment override, PTR, SHORT, THIS)
  attribute value operators (SEG, OFFSET, TYPE, LENGTH, SIZE)
- Record-Specific Operators (shift count, MASK, WIDTH)

These operators can be used to define numbers or with the attribute operators you may define variables or labels. Each type of operator is discussed below.

Types of Expression Operands

Numbers

A number or constant (17-bit number) can be used in most expressions. There are some limitations in the use of relocatable numbers (these are numbers whose values are unknown during assembly). These limitations and the definition for relocatable numbers are discussed below. The attribute value operators (e.g., OFFSET) return numbers that can be relocatable. Simple numbers or constants (such as "2") can be used without any limitations for most operators and expression types. An absolute number is a value known at assembly time.

Address Expressions

An address expression defines a location in memory. This location can be viewed as either a variable or label, depending on the type of expression used. The simplest address expression is the name of a variable or label. In this case, the name implies addressing using the offset of the variable or label.

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

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

ADD  DX, COUNT[2]       ;is equivalent to COUNT + 2
```

A register expression is an address expression that uses a pointer and/or index register. This form was shown above under the discussion of addressing modes. The different types of register expressions are shown below:

1. [pointerreg] or [indexreg]

```
[BX]          [SI]
[BP]          [DI]
```

2. [pointerreg + indexreg]

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

3. [pointerreg + disp] or [indexreg + disp]

```
[BX + disp]        [SI + disp]
[BP + disp]        [DI + disp]
```

4. [pointerreg + indexreg + disp]

```
[BX + SI + disp]
[BX + DI + disp]
[BP + SI + disp]
[BP + DI + disp]
```

NOTE

Disp can be either an 8 or 16 bit displacement.

NOTE

You may also substitute a set of "[ ]" for the "+" in these types of expressions. For example, the following are equivalent forms:

```
[BX + SI]       is the same as    [BX][SI]
[BP + DI + 2]   is the same as    [BX][DI][2]
```

A register expression can be combined with a simple address expression to form a more complex address. This allows for indexed variables or doubly-indexed variables. The form is:

name [reg exp]

Examples:

```
COUNT[BX]           ;simple index

COUNT[BX + 2]       ;index plus displacement

COUNT[BX + SI]      ;double index
```

A register expression implies that the address of the operand will be computed using the run-time contents of the registers used. For the examples above, the offset of the variable COUNT will be added to the contents of the register(s) in the register expression.

You may use a register expression by itself as an operand. This case is called an anonymous reference because the reference has no explicit type (either byte, word, etc.). When using this form you must insure that a type is specified. For a two operand instruction where one of the operands is a register, the assembler will determine the type from the type of the register. For example:

```
MOV  CX,  [BX]           ;move word pointed to by BX
```

In all other cases using an anonymous reference, you must specify the type using the PTR operator (discussed below). For example:

```
MOV  WORD PTR [DI], 5    ;assign two bytes

INC  BYTE PTR [BX + 2]   ;increment a byte
```

Accessing Structure Fields

Another form of address expression uses a structure field name as a displacement that is added to an offset. A field name represents an offset within the structure (see Chapter 3). For example,

```
ASTRUC   STRUCTURE

ABYTE    DB  0        ;offset = 0
AWORD    DW  0        ;offset = 1
BYTE2    DB  0        ;offset = 3

ASTRUC   ENDS
```

The field names can then be combined with a variable name or register expression to form an address expression. This address expression has the following attributes:

segment—same as variable or machine default for register expression

offset —offset of variable or register expression plus the offset of the field within the structure.

type —type of structure field.

For example,

```
ANARRAY   DB  1,2,3,4,

          MOV  AL, ANARRAY.BYTE2   ;AL will equal 4

          MOV  CX, ANARRAY.AWORD   ;CX will equal 0302H

          MOV  BX, OFFSET ANARRAY  ;BX holds offset

          MOV  AL, [BX].ABYTE      ;AL will equal 1
```

Relocatable Expressions

Address expressions (those involving variables and labels) and numeric expressions may have results which cannot be known until logical segments have been combined and located. These expressions are termed "relocatable." The following rules define when an expression is relocatable. There are some restrictions on the use of relocatable expressions with some of the operators. These restrictions are noted below for each operator.

1. Segments and Groups—the name of a segment or group can be used to represent its paragraph number in an expression. This value is relocatable for all segments and groups except for a segment defined with the "AT exp" form of the SEGMENT directive. These values are assigned by the locator or loader. This type of relocatability is called "Base relocatability."

   Example:

```
DATAGRP  GROUP  DATA1, DATA2

DATA1  SEGMENT PUBLIC
          .
          .
          .
DATA1  ENDS

DATA2  SEGMENT PUBLIC

SEGSTORE  DW  DATAGRP       ;DATAGRP is base relocatable

SEGBASE   DW  DATA1         ;DATA1 is base relocatable

DATA2  ENDS
```

2. Variables and Labels—a variable or label is not considered to be relocatable if it is defined in a "non-relocatable segment." This is a segment that has either a PARA or PAGE alignment type and is not a PUBLIC or STACK segment, or it was defined with the "AT exp" combine-type. Use of a variable name in an expression implies the value of its offset within its segment. This value will be relocatable for any variable or label that is defined in a "relocatable" segment or in an EXTRN directive. A relocatable variable or label is "offset relocatable." These values are assigned by the linker.

   Example:

```
DATA SEGMENT PUBLIC

ABYTE   DB  0

AWORD  DW  ABYTE         ;ABYTE is offset relocatable

DATA ENDS
```

3. Numbers—a constant is relocatable if it is defined in an EXTRN directive with type ABS. In this case the term "relocatable" indicates that the value of the number, defined in another module, is unknown at assembly time.

   Example:

```
EXTRN  ANUMBER:ABS

DATA SEGMENT
AWORD  DW  ANUMBER        ;ANUMBER is relocatable

DATA ENDS
```

Arithmetic Operators

HIGH/LOW

Syntax:

```
HIGH   operand
LOW    operand
```

Description:

These operators are called the "byte isolation" operators. HIGH and LOW accept either a numeric expression or a variable or label as an operand. HIGH returns the high-order byte; LOW the low-order byte. If the operand is an absolute number then the result will be absolute. In all other cases, the result will be relocatable. An error will result if these operators are used with an operand or expression involving a segment or group name. For example,

```
MOV      AH, HIGH (1234H)       ;AH = 12H
TENHEX   EQU  LOW (OFF10H)      ;TENHEX = 10H
```

These operators can be applied to each other; if Q is a relocatable value, the following identities apply:

```
LOW  LOW   Q = LOW  Q
LOW HIGH   Q = HIGH Q
HIGH LOW   Q = 0
HIGH HIGH  Q = 0
```

Field Values:

operand

A numeric expression or a variable or label name.

Multiplication and Division

Syntax:

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

Description:

You may only use these operators with absolute numbers, and the result is always an absolute number. Either operand may be a numeric expression, as long as the expression evaluates to an absolute number. Some examples,

```
CMP  AL, 2*4        ;compare AL to 8
MOV CX, 123H/16     ;CX = 12H
```

Field Values:

operand

An absolute number.

Shift Operators

Syntax:

Shift right:	operand	SHR	count
Shift left:	operand	SHL	count

Description:

The shift operators will perform a "bit-wise" shift of the operand. The operand will be shifted "count" bits either to the right or the left. Bits shifted into the operand will be set to 0. The operands must be numeric expressions that evaluate to absolute numbers. For example,

```
MOD  BX, 0FACBH SHR 4      ;BX = BX + 0FACH
```

Field Values:

operand

A numeric expression that evaluates to an absolute number.

count

An absolute number that represents the number of bits the operand is to be shifted, either right or left.

Addition and Subtraction

Syntax:

```
Addition:       operand  +  operand

Subtraction:    operand  –  operand
```

Description:

These operators can be used with either absolute or relocatable operands, but there are certain restrictions in the use of relocatable operands. The following shows all the allowed uses of absolute and relocatable operands.

ABS = an absolute operand

RELOC = a relocatable operand

```
ABS   +  ABS       ABS   –  ABS

ABS   +  RELOC     RELOC –  ABS

RELOC +  ABS       RELOC –  RELOC
```

NOTE

"reloc—reloc" is only allowed for operands with the same type of relocatability and the quantities are defined in the same segment (both are either base or offset relocatable). The result of "reloc-reloc" is an absolute number.

Field Values:

operand

An expression evaluating to an absolute number or a variable or label name.

Relational Operators

Syntax:

```
equal:                   operand  EQ  operand

not equal:               operand  NE  operand

less than:               operand  LT  operand

less than or equal:      operand  LE  operand

greater than:            operand  GT  operand

greater than or equal:   operand  GE  operand
```

Description:

The relational operators may have operands that are:

a. both absolute numbers

b. variable or label names (defined in the current module), that have the same type of relocatability.

The result of a relational operation is always an absolute number. They return an 8-or 16-bit result of all 1's for TRUE and all 0's for FALSE. Some examples,

```
MOV   AL, 3 EQ 0        ; AL = 0 (false)
MOV   BX, 2 LE 15       ; BX = 0FFFFH (true)
```

Field Values:

operand

An absolute number or a variable or label name.

Logical Operators

Syntax:

operand OR operand

operand XOR operand

operand AND operand

NOT operand

Description:

The logical operators may only be used with absolute numbers. They always return an absolute number.

A logical operator can be either:

1. OR—logical "or", maps 0's in corresponding positions into 0 and 1's elsewhere in the result, for example,

   11011001B OR 10011011B = 11011011B

2. XOR—exclusive "or", maps corresponding bits equal in value into 0, and corresponding bits unequal in value into 1, for example,

   10111011B XOR 11011101B = 01100110B

3. AND—logical "and", maps 1's in corresponding positions into 1 and 0's elsewhere in the result, for example,

   10110011B AND 1101101B = 10000001B

4. NOT logical negation, forms the 1's complement by mapping 1's to 0's and 0's to 1's, for example,

   NOT(10101111B) = 01010000B

Field Values:

operand

An absolute number.

Attribute Overriding Operators

Segment Override

Syntax:

CS:varlab

DS:varlab

SS:varlab

ES:varlab

segname:varlab

groupname:varlab

Description:

The segment override is used to override the segment attribute of a variable or label. There are two uses for this override, the first is similar to an ASSUME, and the second is used in order to store the correct offset of a variable or label.

The first form uses a segment register as the "segpart" of a memory address. In this case you are specifying from which segment register the variable or label is addressable. This form is similar to an ASSUME, except that it is restricted to a single statement. It is also more error prone than the use of an ASSUME because you must explicitly code the override for each reference to a variable or label. The explicit use of a segment override takes precedence over any ASSUME directive. The following example illustrates the use.

```
ASSUME DS:DATA, CS:CODE

DATA SEGMENT

ABYTE DB 0

DATA ENDS

CODE SEGMENT

     MOV  BL, ABYTE       ;reference is covered by the
                          ;ASSUME

     MOV  BL, ES:ABYTE    ;no ASSUME is required here for
                          ;this reference

CODE ENDS
```

Another use of this form is to override the implicit use of a segment register in accessing data. The 8086/8088 will use the DS register in order to access data. When the following line of code is executed, the DS register is used.

```
MOV   BL, [BX]
```

You may use the segment override to change this implicit use. If, for example, your data is addressable through the ES register and you do not have an ASSUME, you can code the following form:

```
MOV   BL, ES:[BX]
```

The instruction that is assembled will be preceded by a "segment override prefix" byte that forces the 8086/8088 to use the ES register in order to calculate the physical address of the variable. The same effect will occur if you ASSUME your data into ES.

The second use of the segment override is to insure that your use of the OFFSET operator (see below) will return the correct offset of your variable or label. When a variable or label is defined in a segment that is part of a group, then you must use the segment override with the group name when you use the OFFSET operator (see the discussion of the OFFSET operator given below). This is to ensure that the offset from the group base, rather than the segment base, is returned.

Field Values:

varlab

A variable name, label name, or address-expression.

segname

A segment name.

groupname

A group name.

PTR Operator

Syntax:

type PTR name

Description:

The PTR operator is used to define a memory reference with a certain type. The assembler determines the correct instruction to assemble based on the type of the operands to the instruction. There are certain instances where you may specify an operand that has no type. These cases involve the use of numeric or register expressions. Here the PTR operator is used to specify the type of the operand. The following examples illustrate this use:

```
MOV  WORD  PTR  [BX], 5      ;set word pointed to by BX = 5

INC  DS:BYTE  PTR  10        ;increment byte at offset 10
                             ;from DS
```

This form can also be used to override the type attribute of a variable or label. If, for example, you wished to access an already defined word variable as two bytes, you could code the following:

```
MOV  CL, BYTE  PTR  AWORD          ;get first byte

MOV  DL, BYTE  PTR  AWORD + 1      ;get second byte
```

Field Values:

type

This field can have one of the following values:

1. BYTE
2. WORD
3. DWORD
4. QWORD
5. TBYTE
6. NEAR
7. FAR

name

This field can be:

1. A variable name.
2. A label name.
3. An address or register expression.
4. An integer that represents an offset.

SHORT Operator

Syntax:

SHORT label

Description:

The SHORT operator is used to specify that the label referenced by a JMP instruction is within +127 bytes at the instruction. This operator is only used when the label is forward referenced in the instruction. When the assembler encounters a forward reference, it must make certain assumptions. When a label is forward referenced, the assembler assumes that it will require two bytes to represent the relative offset of the label. By correctly using the SHORT operator, you can save a byte of code when you use a forward reference. If the label is not within the specified range, an error will occur. The following example illustrates the use of the SHORT operator.

```
JMP    FWDLAB        ;three byte instruction

JMP SHORT FWDLAB     ;two byte instruction
```

Field Values:

label

A label addressable through CS.

Attribute Value Operators

The operators discussed below return the numerical values of the attributes of a variable or label. These operators do not change the attribute of the variable or label used.

THIS Operator

Syntax:

THIS type

Description:

The THIS operator defines a memory location at the current location of assembly. This location can be either a variable or a label. Its segment attribute will be the current segment being assembled and its offset will be the value of the current location counter. Its type will be specified by the operand to this operator. Use of this operator is similar to the use of the LABEL directive. This operator is used either in conjunction with the EQU directive (see below) or as part of an operand to an instruction. (The latter form will be rarely used.) It can be used to define another name with an alternate type for a data item; for example

```
AWORD       EQU   THIS WORD
BYTE1       DB    0
BYTE2       DB    0
```

is equivalent to:

```
AWORD       LABEL WORD
BYTE1       DB    0
BYTE2       DB    0
```

Use of the symbol "$" (the location counter symbol) is equivalent to THIS NEAR.

Field Values:

type

This field can have the following values:

1. BYTE
2. WORD
3. DWORD
4. QWORD
5. TBYTE
6. NEAR
7. FAR

SEG Operator

Syntax:

SEG varlab

Description:

This operator returns the segment value of the variable or label, a base relocatable quantity. Use of this operator can have two interpretations, depending on the context used. In an ASSUME directive, you may use this operator to specify the segment in which an object is defined. For example,

```
ASSUME CS:SEG START, DS:SEG COUNT
```

specifies that CS will hold the paragraph number of the segment containing "start" and that DS will hold the paragraph number of the segment in which "count" was defined. This construct is useful with objects for which you do not know the segment in which they are defined (most likely defined in another module). In this case the expression is a symbolic representation of the segment's name.

The other type of interpretation is that of a paragraph number. Here it is used either to store the paragraph number in a variable or to initialize a segment register.

```
SETSTART  DW   SEG COUNT        ;store the paragraph number
                                ;for the segment

INIT:     MOV  AX, SEG COUNT
          MOV  DS, AX           ;init DS with count's
                                ;segment
```

The SEG operator should be avoided when groups are used. Variables and labels are relative to the base of the group and not to the segment in which they are defined. The value returned by the SEG operator for an element that is contained within a group will not reflect the group base.

Field Values:

varlab

The name of a variable or label.

OFFSET Operator

Syntax:

OFFSET varlab

Description:

This operator returns the offset of the variable or label from the base of the segment in which it is defined. In most cases, the value returned is not set until link time, i.e., it is a relocatable number. The OFFSET operator is used primarily to initialize variables or registers to be used for indirect addressing. Some instructions explicitly use indirect addressing when accessing data. When coding these instructions, you are required to initialize a register to the offset value of the data you wish to access. The following example demonstrates this use—

```
TRANSLATE:  MOV  BX, OFFSET ASCIITABLE
            MOV  AL, VALUE
            XLAT                 ;BX points to translation
                                 ;table
```

If a variable or label is contained in a group (its segment is defined to be in a group), then you must use a group override with the OFFSET operator. This ensures that the offset used is from the group base and not from the individual segment base. For example,

```
DGROUP  DATA1, DATA2

DATA1   SEGMENT  PUBLIC
          .
          .
          .
DATA1   ENDS

DATA2   SEGMENT PUBLIC

ASCIITABLE  DB  0
            DB  1
            .
            .
            .
            DB  128

DATA2    ENDS

CODE     SEGMENT  PUBLIC

TRANSLATE: MOV  BX, OFFSET DGROUP:ASCIITABLE  ;need group
                                              ;override
                                              ;here
           MOVE  AL, VALUE
           XLAT                   ;BX points to
                                  ;translation table

CODE     ENDS
```

Field Values:

varlab

The name of a variable or label.

TYPE Operator

Syntax:

TYPE varlab

Description:

The TYPE operator returns a value that represents the type of the operand. This value can be useful in certain instruction sequences where the type of the operand is used to calculate a value used in incrementing a pointer. For example,

```
        MOV  BX, OFFSET ARRAY
        MOV  CX, LENGTH ARRAY         ;LENGTH = # of elements
        MOV  SI, 0                    ;used as index into array
ALAB:   ADD  AX, [BX + SI]            ;array element added to
                                      ;AX
          .
          .
          .
        ADD  SI, TYPE ARRAY           ;increment the pointer by
        LOOP ALAB                     ;the size of an array
                                      ;element
```

TYPE returns the following values, depending on the type of the operand—

1. A byte—returns 1.
2. A word—returns 2.
3. A dword—returns 4.
4. A qword—returns 8.
5. A tbyte—returns 10.
6. A structure name—returns a value equal to the number of bytes declared in the structure definition.
7. A near label—returns 255.
8. A far label—returns 254.

Field Values:

varlab

The name of a variable, structure, or label.

LENGTH Operator

Syntax:

LENGTH variable

Description:

LENGTH returns the number of data units (bytes, words, or dwords) that have been allocated for a variable. The data unit is equal to the type of the variable. This operator is useful for setting a counter for a loop that accesses the elements of an array (see example above).

```
AWORDARRAY   DW  150  DUP (0)       ;LENGTH = 150

ABYTEARRAY   DB  1,2,3,4,5,6,7      ;LENGTH = 7
```

Field Values:

variable

The name of a variable.

SIZE Operator

Syntax:

SIZE variable

Description:

The SIZE operator returns the number of bytes allocated for a variable. This value is related to the LENGTH and TYPE operators through the following identity:

SIZE = LENGTH * TYPE

Some examples,

```
AWORDARRAY   DW  150 DUP (0)          ;SIZE = 300

ABYTEARRAY   DB  1,2,3,4,5,6,7        ;SIZE = 7

             MOV AX, SIZE AWORDARRAY  ;AX = 300

ASIZE        DB  SIZE ABYTEARRAY      ;ASIZE is initialized
                                      ;to 7
```

Field Values:

variable

The name of a variable.

Record Specific Operators

Use of records may involve three special operators. These operators allow you to isolate and access the fields defined within a record. Since the fields in a record are mapped into bits and not into byte-aligned structures, you may require that these fields be masked off (in order to isolate only specific bits) and then shifted into the lower order bits of a byte. (The record-specific operators are described individually below.)

A record name can also be used in an expression. In this case the record is used to specify a number based on the initialization used. For example, if you define the record

```
R  RECORD  F1:8, F2:8
```

you could use it to define a numeric expression that will evaluate to a constant number.

```
MOV  AX, R<0ABH, 'C'>                 ;AX = 0AB43H
MOV  BX, R<5,7> + R<3,4>              ;BX = 080BH
MOV  CX, R<86H, 23H> XOR R<135, 35>   ;CX = 100H
```

Shift Count

Syntax:

recfieldname

Description:

Use of the record field name specifies the number of bits the record must be shifted in order to move the field in it to the low order bits of a byte or word (depending on the size of the record). For example, if you had defined the following record:

```
PATTERN   RECORD   A:3, B:1, C:2, D:4, E:6

AREC      PATTERN  < >
```

you could use the following sequence of code to isolate and access the field C in the record:

```
      MOV  DX,  AREC        ;move record into DX
      AND  DX,  MASK C      ;mask out fields A,B,C,D,E--
                            ;00001100000000013 is the
                            ;value used
      MOV  CL,  C           ;field name as shift count--10
                            ;is the value used
      SHR  DX,  CL          ;DX is now equal to value of
                            ;field C
```

Field Values:

recfieldname

The name of a field within a record.

MASK Operator

Syntax:

MASK record-field

Description:

Use of this operator defines a value that can be used to mask off fields in a record (i.e., a value with 1's in those bit positions specified by the record field, and 0's elsewhere), leaving only the record-field specified. This operator is used with an AND (or TEST) instruction with the operands being 1) the record stored either in a register or a memory location, and 2) an expression using the MASK operator. See the previous example for an illustration.

Field Values:

record-field

The name of a field within a record.

WIDTH Operator

Syntax:

```
WIDTH  rec
```

Description:

The WIDTH returns a value equal to the number of bits in either a record or a record field. From the above example:

```
DB  WIDTH AREC      ;equals 16

DB  WIDTH C         ;equals 2
```

Field Values:

rec

Either a record name or record field name.

Operator Precedence

The following is a list, in decreasing order of precedence, of the classes of operators. All expressions are evaluated from left to right following the precedence rules. You may override this order of evaluation and precedence through the use of parentheses.

Highest Precedence

1. Parenthesized expressions, angle-bracket (record) expressions, square-bracket expressions, the structure "dot" operator, and the operators LENGTH, SIZE, WIDTH, and MASK.
2. PTR, OFFSET, SEG, TYPE, THIS, and "name:" (segment override).
3. HIGH and LOW.
4. Multiplication and division: *, /, MOD, SHR, SHL.
5. Addition and subtraction: +, –.
   a. unary
   b. binary
6. Relational: EQ, NE, LT, LE, GT, GE.
7. Logical NOT.
8. Logical AND.
9. Logical OR and XOR.
10. The SHORT operator.

Lowest Precedence

The EQU Directive

Syntax:

equ-name EQU equ-value

Description:

The EQU directive is a very powerful means to define symbols for many of the ASM86 constructs. These symbols can form names that have more mnemonic value, or that form a "shorthand" notation for a complex construct. In the FIELD VALUE section below, many examples are given showing its use.

Field Values:

equ-name

A unique ASM86 identifier.

equ-value

This field can have the following values:

1. A variable or label name (may be forward referenced).

```
ALABEL      EQU   ALAB

ALAB:       MOV   AX, 0
```

2. An 8086/8088 register name.

```
COUNT       EQU  CX
POINTER     EQU  BX

            MOV  COUNT, 10                ;CX = 10
            MOV  POINTER, OFFSET ARRAY    ;BX = offset
                                          ;of array
               .
               .
               .
```

3. 8086/8087/8088 instruction names.

```
DATAMOVE    EQU  MOV
INCREMENT   EQU  INC

            DATAMOVE   AX, BX
            INCREMENT  AX
```

4. A numeric constant (integer or floating-point).

```
PI      EQU  3.14159
TOTAL   EQU  6
```

The precision of a floating-point number used in an EQUate is determined by the context in which it is used. For example,

```
        DD  PI      ;single precision

        DQ  PI      ;double precision
```

5. An assembly-time expression involving numeric values.

```
E1      EQU  2 + 3

E2      EQU  E1 AND 4

E3      EQU  (E1 - E2) / 12
```

6. A register expression. These may be single register expressions or they may also include a segment override. This construct is useful in defining data items to be accessed on the stack.

```
STACKWORD   EQU  WORD PTR SS:[BP + 2]

AVAR        EQU  [BX + 3]

ANEXTRAVAR  EQU  ES:[BX]
```

Program Linkage Directives 5

Overview of Program Linkage

ASM86 supplies the necessary directives to support multi-modular programs. A program may be composed of many individual modules that are separately assembled or compiled. Each module may define variables or labels that other modules may use. The mechanisms in ASM86 for communicating symbol information from module to module are the PUBLIC/EXTRN directives. The PUBLIC directive defines those symbols that may be used by other modules. The EXTRN directive defines for a given module these symbols (defined elsewhere) that can be used. In order to uniquely name different object modules that are to be linked together, use the NAME directive. The END directive, which is required in all modules, can be used to specify a "main module," that is, a module which contains the code that will be initially executed upon loading the program. It supplies a means to specify the start address of the program that will be initialized by the loader. For assemblers running on 8086-based systems, initialization values for other segment registers may also be specified in the END directive.

The PUBLIC Directive

Syntax:

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

Description:

The PUBLIC directive specifies which symbols in the module are available to other modules at link-time. These symbols may be variables, labels, or constants (17-bit numbers, defined using EQU). All other symbols will be flagged as an error.

Field Values:

name

Any user-defined variable, label, or constant (17-bit number).

The EXTRN Directive

Syntax:

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

Description:

The EXTRN directive specifies those symbols, which may be referenced in the module, that have been declared "public" in another module. The EXTRN directive will specify the name of the symbol and its type.

Field Values:

name

The name of the symbol declared to be public in another module.

type

The type of the symbol declared public in another module. This type should agree with the type of the symbol declared public. This field can have the following values:

1. BYTE—a variable of type byte.
2. WORD—a variable of type word.
3. DWORD—a variable of type dword.
4. QWORD—a variable of type qword.
5. TBYTE —a variable of type tbyte.
6. A structure name: indicates a variable whose type is equal to the number of bytes allocated in the structure definition.
7. A record name: type will be either a byte or word depending on the size of the record.
8. NEAR- -a label of type near.
9. FAR- -a label of type far.
10. ABS—a constant (17-bit number), always of type word.

The Placement of EXTRN's

You must be careful in placing the EXTRN directive because the location of the EXTRN directive in relation to the definition of program segments is very critical. The following rules apply:

1. If you know the segment in which the external symbol is defined, then place the EXTRN directive between a SEGMENT/ENDS pair that is identical to the SEGMENT/ENDS pair in which the object was defined in the other module. The object can be used like any other variable or label. For example, if in the module SCAN.A86, you defined a variable such as the one below

```
DATA     SEGMENT WORD PUBLIC

COUNT    DB   0

PUBLIC   COUNT

DATA     ENDS
```

   you would place the EXTRN directive in the module, PARSE.A86, in the following way:

```
DATA    SEGMENT WORD PUBLIC

EXTRN   COUNT:BYTE

DATA    ENDS
```

2. If you do not know the segment in which the external symbol is defined, or if the segment in which it is defined is non-combinable, then place the EXTRN directive outside of all SEGMENT/ENDS pairs in your program. To address the external symbol you must load the segment part (paragraph number) of the symbol into a segment register using the SEG operator.

   ```
   MOV   AX, SEG CCUNT
   MOV   ES, AX
   ```

   Then you must either use an ASSUME directive to verify addressability such as

   ```
   ASSUME   ES:SEG COUNT
               .
               .

   MOV   DX, COUNT
   ```

   or use a segment override for each use of that symbol.

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

The END Directive

Syntax:

END [regint [, ...]]

Description:

The END directive is required in all ASM86 module programs. It is, appropriately, the last statement in the module. Its occurrence terminates the assembly process; any text found beyond the END directive will be ignored (and an error will be issued). Another purpose of the END directive is to define the module as being a MAINMODULE. This implies that the code contained in the module will be the code that is initially executed when the program is loaded into memory. Execution will begin at the label in your code specified as the start address in the END directive.

The END directive can also be used to define the initial contents of DS and SS. It specifies values to be placed in the segment registers by the loader as it loads the program prior to execution. If this alternate means of initializing these registers is used, then the initial values for CS:IP are required. You could also choose to write some code to do the same initialization.

For example:

```
ASSUME   CS:CODE, DS:DATA, SS:STACK

DATA   SEGMENT

  ABYTE   DB   0

DATA   ENDS

STACK  SEGMENT   STACK

           DW    10 DUP (?)
  STACKTOP DW 0
```

```
STACK ENDS

CODE  SEGMENT

START:  MOV   AL, ABYTE       ; DS is already initialized

         .
         .
         .

CODE  ENDS

END  START, DS:DATA, SS:STACK:STACKTOP  ; CS:IP points to
                                        ; CODE:START
                                        ;  DS  points  to
                                       DATA
                                        ; SS:SP points to
                                        ; STACK:STACKTOP
```

Field Values:

[regint|

This field defines the contents for a segment register (and also the registers IP and SP). To initialize the segment registers, the following formats apply:

'Segname' is either a segment name or a group name. It identifies the paragraph number to be loaded into the segment register.

'Labelname' is the name of a label defined in the module. Its offset will be used to initialize IP.

'Varname' is the name of a variable defined in the module. Its offset will be used to initialize SP.

To initialize CS and IP:

labelname	(the segment part of the label is used for CS)
orCS:labelname	(same as 'labelname")
or	
CS:segname:labelname	(the segment part or paragraph number that is to be loaded into CS is taken from segname)

To initialize SS and SP:

SS:segname:varname	(SP will be initialized to the offset of varname)
or	
SS:segname	(SP will be initialized to be equal to the size of the segment)

To initialize DS:

DS:segname

The NAME Directive

Syntax:

```
NAME   modname
```

Description:

Each object module that will be linked to others must have a unique name. The NAME directive is used to specify this name. If it is not used, an error will occur and the assembler will give the object module the default name ANONYMOUS. Using this default name can cause problems when linking together assembly language modules. LIB-86 will report an error if two modules have the same name.

Field Values:

modname

A user-defined identifier. Hands-off and dual-function keywords are invalid and will cause an error.

The 8086/8087/8088 Instruction Set 6

The 8086/8088 Instruction Set

Instruction Statement Formats

The format for the instruction statement was introduced in Chapter 4. The format is shown below:

```
[label:] [prefix] mnemonic [operand [, operand]];
```

This chapter describes the 8086/8087/8088 instruction set. The instruction set consists of a set of mnemonics that select different machine operations. The instruction set encyclopedia at the end of this chapter describes each of these mnemonics, their operations, and allowed operands.

Addressing Modes

The 8086 instruction set provides several different ways to address operands. Most two-operand instructions allow either memory or a register to serve as one operand, and either a register or a constant within the instruction to serve as the other operand. Memory to memory operations are excluded.

Operands in memory may be addressed *directly* with a 16-bit offset address, or *indirectly* with *base* (BX or BP) and/or *index* (SI or DI) registers added to an optional 8- or 16-bit displacement *constant*. This constant can be the name of a variable or a pure number. When a name is used, the displacement constant is the variable's offset (see Chapter 4).

The result of a two-operand operation may be directed to either memory or a register. Single-operand operations are applicable uniformly to any operand except immediate constants. Virtually all 8086 operations may specify either 8- or 16-bit operands.

Memory Operands

Operands residing in memory may be addressed in four ways:

- Direct 16-bit offset address
- Indirect through a base register, BX or BP, optionally with an 8- or 16-bit displacement
- Indirect through an index register, SI or DI, optionally with an 8- or 16-bit displacement
- Indirect through the sum of one base register and one index register, optionally with an 8- or 16-bit displacement.

The location of an operand in an 8086 register or in memory is specified by up to three fields in each instruction. These fields are the mode field (mod), the register field (reg), and the register/memory field (r/m). When used, they occupy the second byte of the instruction sequence. This byte is referred to as the Modrm byte of the instruction.

The mode field occupies the two most significant bits, 7 and 6, of the byte, and specifies how the r/m field (bits 2, 1, 0) is used in locating the operand. The r/m field can name a register that holds the operand or can specify an addressing mode (in combination with the mod field) that points to the location of the operand in memory. The reg field occupies bits 5, 4, and 3 following the mode field, and can specify that one operand is either an 8-bit register or a 16-bit register. In some instructions, this reg field gives additional bits of information specifying the instruction, rather than only encoding a register.

Description

The effective address (EA) of the memory operand is computed according to the *mod* and *r/m* fields:

```
if mod  = 00 then DISP = 0*, disp-low and disp-high are absent
if mod  = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod  = 10 then DISP = disp-high:disp-low
if r/m  = 000 then EA = (BX) + (SI) + DISP
if r/m  = 001 then EA = (BX) + (DI) + DISP
if r/m  = 010 then EA = (BP) + (SI) + DISP
if r/m  = 011 then EA = (BP) + (DI) + DISP
if r/m  = 100 then EA = (SI) + DISP
if r/m  = 101 then EA = (DI) + DISP
if r/m  = 110 then EA = (BP) + DISP*
if r/m  = 111 then EA = (BX) + DISP

*except if mod = 00 and r/m = 110 then
 EA = disp-high: disp-low
```

Instructions referencing 16-bit objects interpret EA as addressing the low-order byte; the word is addressed by EA+1,EA.

Encoding

mod reg r/m	disp-low	disp-high

Segment Override Prefixes

General register BX and pointer register BP may serve as base registers. When BX is the base the operand by default resides in the current Data Segment and the DS register is used to compute the physical address of the operand. When BP is the base, the operand by default resides in the current Stack Segment and the SS segment register is used to compute the physical address of the operand. When both base and index registers are used, the operand by default resides in the segment determined by the base register, i.e., BX means DS is used, BP means SS is used. When an index register alone is used, the operand by default resides in the current Data Segment. The physical address of most other memory operands is by default computed using the DS segment register (exceptions are noted below). These assembler-default segment register selections may be overridden by preceding the referencing instruction with a segment override prefix.

Description

The segment register selected by the *reg* field of a segment prefix is used to compute the physical address for the instruction this prefix precedes. This prefix may be combined with the LOCK and/or REP prefixes, although the latter has certain requirements and consequences –see REP.

Encoding

```
0 0 1 reg 1 1 0
```

reg is assigned according to the following table:

	Segment
00	ES
01	CS
10	SS
11	DS

Exceptions

The physical addresses of all operands addressed by the SP register are computed using the SS segment register, which may not be overridden. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden.

Register Operands

The four 16-bit general registers and the four 16-bit pointer and index registers may serve interchangeably as operands in nearly all 16-bit operations. Three exceptions to note are multiply, divide, and some string operations, which use the AX register implicitly. The eight 8-bit registers of the HL group may serve interchangeably in 8-bit operations. Multiply, divide, and some string operations use AL implicitly.

Description

Register operands may be indicated by a distinguished field, in which case REG will represent the selected register, or by an encoded field, in which case EA will represent the register selected by the r/m field. Instructions without a "w" bit always refer to 16-bit registers (if they refer to any register at all); those with a "w" bit refer to either 8- or 16-bit registers according to "w".

Encoding

General Registers:

Distinguished Field.

```
|        reg |   or   |    reg     |
```

for mode = 11 EA = r/m (a register).

```
|  11  reg  |
```

REG is assigned according to the following table:

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

Instructions that reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

```
FLAGS   X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)
```

where X is undefined.

Immediate Operands

All two-operand operations except multiply, divide, and the string operations allow one source operand to appear within the instruction as immediate data. Sixteen-bit immediate operands that have a high-order byte that is the sign extension of the low-order byte may be abbreviated to eight bits.

Three points about immediate operands:

- Immediate operands always *follow* addressing mode displacement constants (when present) in the instruction.
- The low-order byte of 16-bit immediate operands always precedes the high-order byte.
- The 8-bit immediate operands of instructions with s:w = 11 are sign-extended to 16-bit values.

String Instructions and Memory References

Table 6-1 shows the mnemonics of the string instructions that can be coded without operands (MOVSB, MOVSW, etc.) or with operands (MOVS, etc.).

The string instructions are unusual in several respects:

1. Before coding a string instruction, you must:
   - Load SI with the offset of the source string.
   - Load DI with the offset of the destination string.
2. One of the forms of REP (REP, REPZ, REPE, REPNE, REPNZ) can be coded immediately preceding (but separated from by at least one blank) the primitive string operation mnemonic (thus, REPNZ SCASW is one possibility). This specifies that the string operation is to be repeated the number of times determined by CX. (Refer to instruction descriptions.)
3. Each can be coded with or without symbolic memory operands.
   - If symbolic operands are coded, the assembler can check the addressability of the operands.

Table 6-1. String Instruction Mnemonics

Operation Being Performed	Mnemonic if Operand Is Byte String	Mnemonic if Operand Is Word String	Mnemonic if Symbolic Operands Are Coded*
Move	MOVSB	MOVSW	MOVS
Compare	CMPSB	CMPSW	CMPS
Load AL/AX	LODSB	LODSW	LODS
Store from AL/AX	STOSB	STOSW	STOS
Compare to AL/AX	SCASB	SCASW	SCAS
Block Input	INSB	INSW	INS
Block Output	OUTSB	OUTSW	OUTS

*If symbolic operands are coded, the assembler can check their addressability. Also, their TYPEs determine the opcode generated.

- Anonymous references that use the hardware defaults should be coded using the operand-less forms (e.g. MOVSB, MOVSW), to avoid the cumbersome (but otherwise required):

  ```
  MOVS ES:BYTE PTR [DI], [SI]
  ```

  as opposed to the simple:

  ```
  MOVSB
  ```

- Anonymous references that do not use the hardware defaults require both segment and type to be explicitly specified:

  ```
  MOVS ES:BYTE PTR [DI], SS:[SI]
  ```

- Never use [BX] or [BP] addressing modes with string instructions.

4. If the instruction mnemonic is coded without operands (e.g., MOVSB, MOVSW), then the segment register defaults are as follows:
   - SI defaults to an offset in the segment addressed by DS,
   - DI is required to be an offset in the segment addressed by ES.

   Thus, the direction of data flow for the default case in which no operands are specified is from the segment addressed by DS to the segment addressed by ES.
5. If the instruction mnemonic is coded with operands (e.g. MOVS, CMPS), the operands can be anonymous (indirect) or they can be variable references.

Example:

```
DESTSTRING  EQU ES:BYTE PTR [DI]

SRCSTRING   EQU  DS:BYTE PTR [SI]

ASSUME CS:CODE, DS:DATA, ES:DATA1

DATA SEGMENT

SRCARRAY  DB  10 DUP (1)

DATA ENDS

DATA1 SEGMENT

DESTARRAY  DB  10 DUP (?)

DATA1 ENDS

CODE SEGMENT
         MOV  AX, DATA
         MOV  DS, AX        ;INIT DS
         MOV  AX, DATA1
         MOV  ES, AX        ;INIT ES

         MOV  SI, OFFSET SRCARRAY
         MOV  DI, OFFSET DESTARRAY

                            ;INIT POINTER REGISTERS

         MOV  CX, 10        ;NUMBER OF ELEMENTS
         REP MOVS DESTSTRING, SRCSTRING
```

```
                              ;PERFORM STRING OPERATION
        .
        .
        .
CODE ENDS
```

Mnemonic Synonyms

There are some machine operations that can have different mnemonics. The different mnemonics are all synonyms in that they refer to the same machine instructions. They are supplied by the assembler to allow you to think of the operation in terms that are more helpful for your task. Many of the conditional jump instructions have more than one mnemonic. When used after a compare, the conditional jump mnemonic can express the type of compare or the result of the compare in terms of the flags that were set. For example,

```
CMP  DEST, SRC
JE   LAB1          ;jump if dest is equal to source
```

or

```
CMP  DEST, SRC
JZ   LAB1          ;jump if zero flag set (dest = src)
```

In both cases, the same instruction will be encoded for the jump. Programmers familiar with other assembly languages that use conditional jump mnemonics that refer to flags may be more comfortable using this form. However, the first form that expresses the relationship the compare is checking between the operands is more expressive.

Organization of the Instruction Set

Instructions are described in this section in six functional groups:

- Data transfer
- Arithmetic
- Logic
- String manipulation
- Control transfer
- Processor control

Each of the first three groups mentioned in the preceding list is further subdivided into an array of codes that specify whether the instruction is to act upon immediate data, register or memory locations, whether 16-bit words, or 8-bit bytes are to be processed, and what addressing mode is to be employed. All of these codes are listed and explained in detail, but you do not have to code each one individually. The context of your program automatically causes the assembler to generate the correct code. There are three general categories of instructions within each of the three functional groups mentioned:

1. Register or memory space to or from register
2. Immediate data to register or memory
3. Accumulator to or from registers, memory, or ports

Data Transfer

Data transfer operations are divided into four classes:

1 general purpose

2 accumulator-specific

3 address-object

4 flag

None affect flag settings except SAHF and POPF.

General Purpose Transfers

Four general purpose data transfer operations are provided. These may be applied to most operands, though there are specific exceptions. The general purpose transfers (except XCHG) are the only operations that allow a segment register as an operand.

— MOV performs a byte or word transfer from the source (rightmost) operand to the destination (leftmost) operand.

— PUSH decrements the SP register by two and then transfers a word from the source operand to the stack element currently addressed by SP.

— POP transfers a word operand from the stack element addressed by the SP register to the destination operand and then increments SP by 2.

— XCHG exchanges the byte or word source operand with the destination operand. The segment registers may not be operands of XCHG.

Accumulator-Specific Transfers

Three accumulator-specific transfer operations are provided:

— IN transfers a byte (or word) from an input port to the AL register (or AX register). The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K input ports.

— OUT is similar to IN except that the transfer is from the accumulator to the output port.

— XLAT performs a table lookup byte translation. The AL register is used as an index into a 256-byte table addressed by the BX register. The byte operand so selected is transferred to AL.

Address-Object Transfers

Three address-object transfer operations are provided:

— LEA (load effective address) transfers the offset address of the source operand to the destination operand. The source operand must be a memory operand and the destination operand must be a 16-bit general, pointer, or index register.

— LDS (load pointer into DS) transfers a "pointer-object" (i.e., a 32-bit object containing an offset address and a segment address) from the source operand (which must be a doubleword memory operand) to a pair of destination registers. The segment address is transferred to the DS segment register. The offset address is transferred to the 16-bit general, pointer, or index register that you coded.

— LES (load pointer into ES) is similar to LDS except that the segment address is transferred to the ES segment register.

Flag Register Transfers

Four flag register transfer operations are provided:

— LAHF (load AH with flags) transfers the flag registers SF, ZF, AF, PF, and CF (the 8080 flags) into specific bits of the AH register.

— SAHF (store AH into flags) transfers specific bits of the AH register to the flag registers. SF, ZF, AF, PF, and CF.

— PUSHF (push flags) decrements the SP register by two and transfers all of the flag registers into specific bits of the stack element addressed by SP.

— POPF (pop flags) transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by two.

Arithmetic

The 8086/8088 provides the four basic mathematical operations in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard twos complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer). Correction operations are provided to allow arithmetic to be performed directly on unpacked decimal digits or on packed decimal representations.

Flag Register Settings

Six flag registers are set or cleared by arithmetic operations to reflect certain properties of the result of the operation. They generally follow these rules (see also Appendix C):

— CF is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise, CF is cleared.

— AF is set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the low-order four bits of the result; otherwise, AF is cleared.

— ZF is set if the result of the operation is zero; otherwise, ZF is cleared.

— SF is set if the high-order bit of the result of the operation is set; otherwise, SF is cleared.

— PF is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise, PF is cleared (odd parity).

— OF is set if the operation results in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa; otherwise, OF is cleared.

Addition

Five addition operations are provided:

— ADD performs an addition of the source and destination operands and returns the result to the destination operand.

— ADC (add with carry) performs an addition of the source and destination operands, adds one if the CF flag is found previously set, and returns the result to the destination operand.

— INC (increment) performs an addition of the source operand and one, and returns the result to the operand.

— AAA (unpacked BCD (ASCII) adjust for addition) performs a correction of the result in AL of adding two unpacked decimal operands, yielding an unpacked decimal sum.

— DAA (decimal adjust for addition) performs a correction of the result in AL of adding two packed decimal operands, yielding a packed decimal sum.

Subtraction

Seven subtraction operations are provided:

— SUB performs a subtraction of the source from the destination operand and returns the result to the destination operand.

— SBB (subtract with borrow) performs a subtraction of the source from the destination operand, subtracts one if the CF flag is found previously set, and returns the result to the destination operand.

— DEC (decrement) performs a subtraction of one from the source operand and returns the result to the operand.

— NEG (negate) performs a subtraction of the source operand from zero and returns the result to the operand.

— CMP (compare) performs a subtraction of the source destination operand, causing the flags to be affected, but does not return the result.

— AAS (unpacked BCD (ASCII) adjust for subtraction) performs a correction of the result in AL of subtracting two unpacked decimal operands, yielding an unpacked decimal difference.

— DAS (decimal adjust for subtraction) performs a correction of the result in AL of subtracting two packed decimal operands, yielding a packed decimal difference.

Multiplication

Three multiplication operations are provided:

— MUL performs an unsigned multiplication of the accumulator (AL or AX) and the source operand, returning a double length result to the accumulator and its extension (AL and AH for 8-bit operation, AX and DX for 16-bit operation). CF and OF are set if the top half of the result is non-zero.

— IMUL (integer multiply) is similar to MUL except that it performs a signed multiplication. CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

AAM (unpacked BCD (ASCII) adjust for multiply) performs a correction of the result in AX of multiplying two unpacked decimal operands, yielding an unpacked decimal product.

Division

There are three division operations provided and two sign-extension operations to support signed division:

— DIV performs an unsigned division of the accumulator and its extension (AL and AH for 8-bit operation, AX and DX for 16-bit operation) by the source operand and returns the single length quotient to the accumulator (AL or AX), and returns the single length remainder to the accumulator extension (AH or DX). The flags are undefined. Division by zero generates an interrupt of type 0.

— IDIV (integer division) is similar to DIV except that it performs a signed division.

— AAD (unpacked BCD (ASCII) adjust for division) performs a correction of the dividend in AL before dividing two unpacked decimal operands, so that the result will yield an unpacked decimal quotient.

— CBW (convert byte to word) performs a sign extension of AL into AH.

— CWD (convert word to double word) performs a sign extension of AX into DX.

Logic

The 8086/8088 provides the basic logic operations for both 8- and 16-bit operands.

Single-Operand Operations. Three-single-operand logical operations are provided:

— NOT forms the one's complement of the source operand and returns the result to the operand. Flags are not affected.

— Shift operations of four varieties are provided for memory and register operands: SHL (shift logical left), SHR (shift logical right), SAL (shift arithmetic left), and SAR (shift arithmetic right). Single bit shifts, and variable bit shifts with the shift count taken from the CL register are available. The CF flag becomes the last bit shifted out, OF is defined only for shifts with count of 1, and is set if the final sign bit value differs from the previous value of the sign bit, and PF, SF, and ZF are set to reflect the resulting value.

— Rotate operations of four varieties are provided for memory and register operands: ROL (rotate left), ROR (rotate right), RCL (rotate through CF left), and RCR (rotate through CF right). Single bit rotates, and variable bit rotates with the rotate count taken from the CL register, are available. The CF flag becomes the last bit rotated cut; OF is defined only for shifts with count of 1, and is set if the final sign bit value differs from the previous value of the sign bit.

Two-Operand Operations

Four two-operand logical operations are provided. The CF and OF flags are cleared on all operations; SF, PF, and ZF reflect the result.

— AND performs the bitwise logical conjunction of the source and destination operand and returns the result to the destination operand.

— TEST performs the same operations as AND, causing the flags to be affected but does not return the result.

- OR performs the bitwise logical inclusive disjunction of the source and destination operand and returns the result to the destination operand.

— XOR performs the bitwise logical exclusive disjunction of the source and destination operand and returns the result to the destination operand.

String Manipulation

One-byte instructions perform various primitive operations for the manipulation of byte and word strings (sequences of bytes or words). Any primitive operation can be performed repeatedly in hardware by preceding its instruction with a repeat prefix (see REP). The single-operation forms may be combined to form complex string operations with repetition provided by iteration operations.

Hardware Operation Control

All primitive string operations use the SI register to address the source operands. The DI register is used to address the destination operands that reside in the current extra segment. If the DF flag is cleared, the operand pointers are incremented after each operation, once for byte operations and twice for word operations. If the DF flag is set, the operand pointers are decremented after each operation. See Processor Control for setting and clearing DF.

Any of the primitive string operation instructions may be preceded with a one-byte prefix indicating that the operation is to be repeated until the operation count in CX is satisfied. The test for completion is made prior to each repetition of the operation. Thus, an initial operation count of zero in CX will cause zero executions of the primitive operation.

The repeat prefix byte also designates a value to compare with the ZF flag. If the primitive operation is one that affects the ZF flag, and the ZF flag is unequal to the designated value after any execution of the primitive operation, the repetition is terminated. This permits the scan operation, for example, to serve as a scan-while or a scan-until.

During the execution of a repeated primitive operation, the operand index registers (SI and DI) and the operation count register (CX) are updated after each repetition, whereas the instruction pointer will retain the offset address of the repeat prefix byte (assuming it immediately precedes the string operation instruction). Thus, an interrupted repeated operation will be correctly resumed when control returns from the interrupting task.

Using more than one prefix on an instruction is processor dependent. Please refer to the User's Manual for your processor for further information.

Primitive String Operation

Five primitive string operations are provided:

— MOVS (MOVSB, MOVSW) transfers a byte (or word) operand from the source (rightmost) operand to the destination (leftmost) operand. As a repeated operation, this provides for moving a string from one location in memory to another.

— CMPS (CMPSB, CMPSW) subtracts the rightmost byte (or word) operand from the leftmost operand and affects the flags but does not return the result. As a repeated operation, this provides for comparing two strings. With the appropriate repeat prefix it is possible to determine after which string element the two strings become unequal, thereby establishing an ordering between the strings.

— SCAS (SCASB, SCASW) subtracts the destination byte (or word) operand from AL (or AX) and affects the flags but does not return the result. As a repeated operation, this provides for scanning for the occurrence of, or departure from, a given value in the string.

— LODS (LODSB, LODSW) transfers a byte (or word) operand from the source operand to AL (or AX). This operation ordinarily would not be repeated.

— STOS (STOSD, STOSW) transfers a byte (or word) operand from AL (or AX) to the destination operand. As a repeated operation, this provides for filling a string with a given value.

In all the cases above, the source operand is addressed by SI and the destination operand is addressed by DI. Only in CMPB/CMPW does the DI-indexed operand appear as the rightmost operand.

Software Operation Control

The repeat prefix provides for rapid iteration in a hardware-repeated string operation. The iteration control operations (see LOOP) provide this same control for implementing software loops to perform complex string operations. These iteration operations provide the same operation count update, operation completion test, and ZF flag tests that the repeat prefix provides.

By combining the primitive string operations and iteration control operations with other operations, it is possible to build sophisticated yet efficient string manipulation routines. One instruction that is particularly useful in this context is XLAT. It permits a byte fetched from one string to be translated before being stored in a second string, or before being operated upon in some other fashion. The translation is performed by using the value in the AL register as an index into a table pointed at by the BX register. The translated value obtained from the table then replaces the value initially in the AL register (see XLAT).

Control Transfer

Four classes of control transfer operations may be distinguished: calls, jumps, and returns; conditional transfers; iteration control; and interrupts.

All control transfer operations cause the program execution to continue at some new location in memory, possibly in a new code segment. Conditional transfers are provided for targets in the range -128 to +127 bytes from the transfer.

Calls, Jumps, and Returns

Two basic varieties of calls, jumps, and returns are provided—those that transfer control within the current code segment, and those that transfer control to an arbitrary code segment, which then becomes the current code segment. Both direct and indirect transfers are supported; indirect transfers make use of the standard addressing modes as described in above.

The three transfer operations are described below.

— CALL pushes the offset address of the next instruction onto the stack (in the case of an inter-segment transfer the CS segment register is pushed first) and then transfers control to the target operand.

— JMP transfers control to the target operand.

— RET transfers control to the return address saved by a previous CALL operation, and optionally may adjust the SP register so as to discard stacked parameters.

Intra-segment direct calls and jumps specify a self-relative direct displacement, thus allowing *position independent code.* A shortened jump instruction is available for transfers in the range −128 to +127 bytes from the instruction for code compaction.

Conditional Jumps

The conditional transfers of control perform a jump contingent upon various Boolean functions of the flag registers. The destination must be within a −128 to +127 byte range of the instruction. Table 6-2 shows the available instructions, the conditions associated with them, and their interpretation.

Table 6-2. 8086/8087 Conditional Transfer Operations

Instruction	Condition	Interpretation
JE or JZ	ZF = 1	"equal" or "zero"
JL or JNGE	(SF xor OF) = 1	"less" or "not greater or equal"
JLE or JNG	((SF xor OF) or ZF) = 1	"less or equal" or "not greater"
JB or JNAE or JC	CF = 1	"below" or "not above or equal" or "carry"
JBE or JNA	(CF or ZF) = 1	"below or equal" or "not above"
JP or JPE	PF = 1	"parity" or "parity even"
JO	OF = 1	"overflow"
JS	SF = 1	"sign"
JNE or JNZ	ZF = 0	"not equal" or "not zero"
JNL or JGE	(SF xor OF) = 0	"not less" or "greater or equal"
JNLE or JG	((SF xor OF) or ZF) = 0	"not less or equal" or "greater"
JNB or JAE or JNC	CF = 0	"not below" or "above or equal" or "no carry"
JNBE or JA	(CF or ZF) = 0	"not below or equal" or "above"
JNP or JPO	PF = 0	"not parity" or "parity odd"
JNO	OF = 0	"not overflow"
JNS	SF = 0	"not sign"

*"Above" and "below" refer to the relation between two unsigned values, while "greater" and "less" refer to the relation between two signed values.

Iteration Control

The iteration control transfer operations perform leading- and trailing-decision loop control. The destination of iteration control transfers must be within a −128 to +127 byte range of the instruction. These operations are particularly useful in conjunction with the string manipulation operations.

There are four iteration control transfer operations provided:

— LOOP decrements the CX ("count") register by one and transfers if CX is not zero.

— LOOPZ (also called LOOPE) decrements the CX register by one and transfers if CX is not zero and the ZF flag is set (loop while zero or loop while equal).

— LOOPNZ (also called LOOPNE) decrements the CX register by one and transfers if CX is not zero and the ZF flag is cleared (loop while not zero or loop while not equal).

— JCXZ transfers if the CX register is zero.

Interrupts

Program execution control may be transferred by means of operations similar in effect to that of external interrupts. All interrupts perform a transfer by pushing the flag registers onto the stack (as in PUSHF), and then performing an indirect inter-segment call through an element of an interrupt transfer vector located at absolute locations 0 through 3FFH. This vector contains a four-byte element for each of up to 256 different interrupt types.

Three interrupt transfer operations provided.

- INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. A one-byte form of this instruction is available for interrupt type 3.
- INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 if the OF flag is set (trap on overflow). If the OF flag is cleared, no operation takes place.
- IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

Processor Control

Various instructions and mechanisms are provided for control and operation of the processor and its interaction with its environment.

Flag Operations

There are seven operations provided that operate directly on individual flag registers.

- CLC clears the CF flag.
- CMC complements the CF flag.
- STC sets the CF flag.
- CLD clears the DF flag, causing the string operations to auto-increment the operand pointers.
- STD sets the DF flag, causing the string operations to auto-decrement the operand pointers.
- CLI clears the IF flag, disabling external interrupts (except for the non-maskable external interrupt).
- STI sets the IF flag, enabling external interrupts after the execution of the next instruction.

Processor Halt

The HLT instruction causes the 8086 processor to enter its halt state. The halt state is cleared by an enabled external interrupt or RESET.

Processor Wait

The WAIT instruction causes the processor to enter a wait state if the signal on its TEST pin is not asserted. The wait state may be interrupted by an enabled external interrupt. When this occurs the saved code location is that of the WAIT instruction, so that upon return from the interrupting task, the wait state is re-entered. The wait state is cleared and execution resumed when the TEST signal is asserted. Execution resumes without allowing external interrupts until after the execution of the next instruction. This instruction allows the processor to synchronize itself with external hardware.

Processor Escape

The ESC instruction provides a mechanism by which other processors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 processor does no operation for the ESC instruction other than to access a memory operand.

Bus Lock

A special one-byte prefix may precede any instruction causing the processor to assert its bus-lock signal for the duration of the operation caused by that instruction. This has use in multiprocessing applications (see LOCK).

Single Step

When the TF flag register is set, the processor generates a type 1 interrupt after the execution of each instruction. During interrupt transfer sequences caused by any type of interrupt, the TF flag is cleared after the push-flags step of the interrupt sequence. No instructions are provided for setting or clearing TF directly. Rather, the flag register image saved on the stack by a previous interrupt operation must be modified, so that the subsequent interrupt return operation (IRET) restores TF set. This allows a diagnostic task to single-step through a task under test, while still executing normally itself.

If the single-stepped instruction itself clears the TF flag, the type 1 interrupt will still occur upon completion of the single-stepped instruction. If the single-stepped instruction generates an interrupt or if an enabled external interrupt occurs prior to the completion of the single-stepped instruction, the type 1 interrupt sequence will occur after the interrupt sequence of the generated or external interrupt, but before the first instruction of the interrupt service routine is executed.

The 8086/8088 hardware protects the execution of the instruction immediately following a POP or a MOV to a segment register instruction from any kind of interrupt, including type 1 interrupts used to single-step. When single-stepping through a task under test, the single-step interrupt is not recognized until the instruction following the POP or MOV to a segment register instruction is executed.

Example

```
TEST _TASK      SEGMENT
                ASSUME       CS:TEST _TASK
INSTRUC1:       POP          DS
INSTRUC2:       POP          BX
INSTRUC3:       ADD          AX, [BX]
     .
     .
     .
TEST  TASK      ENDS
```

When single-stepping through TEST _TASK, INSTRUC1 steps to INSTRUC3 since the single-step interrupt is not recognized by the 8086/8088 until the instruction following the POP to the DS segment register (POP BX) is executed.

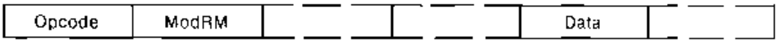
Instruction Description Formats

The formats presented in the individual instruction descriptions and briefly discussed here reflect the assembly language processed by the 8086/8087/8088 Macro Assembler (ASM86).

Format Boxes

The individual instruction descriptions show first a format box such as the following example.

Mem/Reg * Immediate to Reg



These are byte-wise representations of the object code generated by the assembler and are interpreted as follows:

- Opcode is the 8-bit opcode for the instruction. The actual opcode generated is defined in the "Opcode" column of the instruction table that follows each format box.
- ModRM is the byte that specifies the operands of the instruction. It contains a 2-bit mode field (MOD), a 3-bit register field (REG), and a 3-bit Register or Memory (R/M) field.
- Dashed blank boxes following the ModRM box are for any displacement required by the mode field.
- Data is for a byte of immediate data.
- A dashed blank box following a Data box is used whenever the immediate operand is a word quantity.

Instruction Detail Tables

Following each format box, an instruction detail table shows the opcode, the number of clocks required for the operation to take place, the actual operation performed, and a coding example for each variant of the instruction.

The instruction detail table for the instruction IMUL is shown below. The examples in the table are neither complete nor restrictive; anyplace there is a memory operand, any of the seven memory addressing modes can be used.

Opcode	Clocks	Operation	Coding Example
F6	80-98	AX ← AL * Reg 8	IMUL BL
F6	(86-104) + EA	AX ← AL * Mem 8	IMUL BYTESOMETHING
F7	128-154	DX:AX ← AX * Reg 16	IMUL BX
F7	(134-160) + EA	DX:AX ← AX * Mem 16	IMUL WORDSOMETHING

Flags

The flags produced by each instruction are represented by a table such as the following:

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

The top line in the table represents the individual flags, and the lower line shows the effect on each flag by the instruction. The letters, numbers and symbols used in the table are defined as follows:

Flag	Definition
O	Overflow
D	Direction (used in string ops)
I	Interrupt Enable (1–enabled)
T	Single Step Trap Flag (causes interrupt 1 after next instruction)
S	Sign
Z	Zero
A	Auxiliary Carry (used primarily in BCD ops)
P	Parity
C	Carry

Effect Code	Effect
X	Modified by the instruction; result depends on operands.
-	Not modified.
U	Undefined after the instruction.
1	Set to 1 by the instruction.
0	Set to 0 by the instruction.

Table 6-3. Symbols

8086/8088 Descriptor	Meaning
AX	Accumulator (16-bit)
AH	Accumulator (high-order byte)
AL	Accumulator (low-order byte)
BX	Register BX (16-bit), which may be split and addressed as two 8-bit registers
BH	High-order byte of register BX.
BL	Low-order byte of register BX.
CX	Register CX (16-bit), which may be split and addressed as two 8-bit registers.
CH	High-order byte of register CX.
CL	Low-order byte of register CX.
DX	Register DX (16-bit), which may be split and addressed as two 8-bit registers.
DH	High-order byte of register DX
DL	Low-order byte of register DX.
SP	Stack pointer (16-bit)
BP	Base pointer (16-bit)
IP	Instruction Pointer (16-bit)
Flags	16-bit register space, in which nine flags reside.
DI	Destination Index register (16-bit)
SI	Stack Index register (16-bit)
CS	Code Segment register (16-bit)
DS	Data Segment register (16-bit)
ES	Extra Segment register (16-bit)
SS	Stack Segment register (16-bit)

Table 6-3. Symbols (Cont'd.)

8086/8088 Descriptor	Meaning
REG8	The name or encoding of an 8-bit CPU register location.
REG16	The name or encoding of an 16-bit CPU register location.
LSRC, RSRC	Refer to operands of an instruction, generally left source and right source when two operands are used. The leftmost operand is also called the destination operand, and the rightmost is called the source operand.
reg	A field that specifies REG8 or REG16 in the description of an instruction.
EA	Effective address (16-bit)
r/m	Bits 2, 1, and 0 of the MODRM byte used in accessing memory operands. This 3-bit field defines EA, in conjunction with the mode and w fields.
mode	Bits 7 and 6 of the MODRM byte. This 2-bit field defines the addressing mode.
w	A 1-bit field in an instruction, identifying byte instructions (w=0), and word instructions (w=1)
d	A 1-bit field in an instruction, "d" identifies direction, i.e. whether a specified register is source or destination.
(...)	Parentheses mean the contents of the enclosed register or memory location.
(BX)	Represents the contents of register BX, which can mean the address where an 8-bit operand is located. To be so used in an assembler instruction, BX must be enclosed only in square brackets.
((BX))	Means this 8-bit operand, the contents of the memory location pointed at by the contents of register BX. This notation is only descriptive for use in this chapter. It cannot appear in source statements.
(BX) + 1, (BX)	Means the address (of a 16-bit operand) whose low-order 8-bits reside in the memory location pointed at by the contents of register BX and whose high-order 8-bits reside in the next sequential memory location, (BX) + 1
((BX) + 1, (BX))	Means the 16-bit operand that resides there.
Concatenation, e.g., ((DX) + 1: (DX))	Means a 16-bit word that is the concatenation of two 8-bit bytes, the low-order byte in the memory location pointed at by DX and the high-order byte in the next sequential memory location.
addr	Address (16-bit) of a byte in memory
addr-low	Least significant byte of an address.
addr-high	Most significant byte of an address.
addr + 1: addr	Addresses of two consecutive bytes in memory, beginning at addr.
data	Immediate operand (8-bit if w=0; 16-bit if w=1)
data-low	Least significant byte of 16-bit data word.
data-high	Most significant byte of 16-bit data word.
disp	Displacement
disp-low	Least significant byte of 16-bit displacement.
disp-high	Most significant byte of 16-bit displacement.
←	Assignment
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulo
&	And
	Inclusive or
‖	Exclusive or

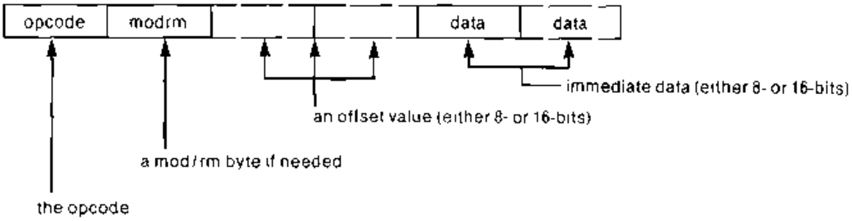
Table 6-4. Effective Address Calculation Time

EA Components		Clocks*
Displacement Only		6
Base or Index Only	(BX BP, SI, DI)	5
Displacement + Base or Index	(BX, BP, SI, DI)	9
Base + Index	BP + DI, BX + SI	7
	BP + SI, BX + DI	8
Displacement + Base + Index	BP + DI + DISP BX + SI + DISP	11
	BP + SI + DISP BX + DI + DISP	12

*Add 2 clocks for segment override

MNEMONIC—Sample 8086/8088 Instruction

Format


opcode
modrm
data
data
immediate data (either 8- or 16-bits)
an offset value (either 8- or 16-bits)
a mod/rm byte if needed
the opcode

Opcode	Clocks	Operation	Coding Example
(the value of the opcode byte)	(number of clocks required)	(the machine operation)	MNEMONIC

Operation

(A description of the machine operation.)

Flags

O D I T S Z A P C

(shows the effect on the flags)

Description

(Describes the use/operation of the instruction.)

AAA—ASCII Adjust for Addition

Format

Opcode

Opcode	Clocks	Operation	Coding Example
37	4	adjust AL, flags, AH	AAA

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do;
   AL ← AL + 6
   AH ← AH + 1
   CF ← AF ← 1
end;
AL ← AL & 0FH
```

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	U	U	X	U	X

Description

AAA is used to correct the result of adding two unpacked BCD digits in the AL register. After the normal byte addition, AAA tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then a carry bit is added to the AH register, and the low nibble of AL is increased by 6 to produce the decimal digit. AL is masked to 4 bits whether an adjustment was performed or not, thus always leaving an unpacked BCD result in the low nibble of AL. High nibble data does not affect the corrected result of the addition, so ASCII digits can be added correctly by following the AAA with an OR AL,30H to restore the result to an ASCII character. The digit carry, in AH, is not affected by this restoration.

AAD—ASCII Adjust for Division

Format

Long —— Opcode

Opcode	Clocks	Operation	Coding Example
D5,0A	60	Adjust AL, AH prior to division	AAD

Operation

AL ← AL + (AH * 0AH)
AH ← 0

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	X	X	U	X	U

Description

AAD is used to prepare 2 unpacked BCD digits (least significant in AL, most significant in AH) for a division operation that will yield an unpacked result. This is accomplished by multiplying AH by 10 and adding the product to AL. Then AH is zeroed, leaving AX with the binary equivalent of the original unpacked 2-digit number.

AAM—ASCII Adjust for Multiplication

Format

Long — — Opcode

Opcode	Clocks	Operation	Coding Example
D4,0A	83	Adjust AL, AH after multiplication	AAM

Operation

AH ← (AL / 0AH)
AL ← (AL MOD 0AH)

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	X	X	U	X	U

Description

AAM is used to produce 2 unpacked BCD digits (least significant in AL, most significant in AH) after a multiplication of 2 unpacked digits. This is accomplished by dividing the binary product in AL by ten. The quotient is left in AH as the most significant digit, and the remainder is left in AL as the least significant digit.

AAS—ASCII Adjust for Subtraction

Format

Opcode

Opcode	Clocks	Operation	Coding Example
3F	4	adjust AL, flags, AH	AAS

Operation

```
if (AL & 0FH) > 9 or AF = 1 then do:
  AL ← AL - 6
  AH ← AH - 1
  CF ← AF ← 1
end;
AL ← AL & 0FH
```

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	U	U	X	U	X

Description

AAS is used to correct the result of subtracting two unpacked BCD digits in the AL register. After the normal byte subtraction, AAS tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If the AF is set or the low nibble of AL is greater than 9, then a borrow bit is subtracted from AH, and the low nibble of AL is decreased by 6 to produce the proper decimal digit. AL is masked to 4 bits whether an adjustment was performed or not, thus always leaving an unpacked BCD result in the low nibble of AL. High nibble data does not affect the corrected result of the subtraction, so ASCII digits can be subtracted correctly by following the AAS with an OR AL,30H to restore the result to an ASCII character. The digit borrow, in AH, is not affected by this restoration.

ADC—Integer Add With Carry

Format

Memory/Reg + Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
12	3	Reg8 ← CF + Reg 8 + Reg8	ADC BL,CL
12	9+EA	Reg8 ← CF + Reg8 + Mem8	ADC BL,BYTESOMETHING
13	3	Reg16 ← CF + Reg16 + Reg16	ADC BX,CX
13	9+EA	Reg16 ← CF + Reg16 + Mem16	ADC BX,WORDSOMETHING
10	16+EA	Mem8 ← CF + Mem8 + Reg8	ADC BYTESOMETHING,BL
11	16+EA	Mem16 ← CF + Mem16 + Reg16	ADC WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
14	4	AL ← CF + AL + Immed8	ADC AL,5
15	4	AX ← CF + AX + Immed16	ADC AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*–(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← CF + Reg8 + Immed8	ADC BL,32
80	17+EA	Mem8 ← CF + Mem8 + Immed8	ADC BYTESOMETHING,32
81	4	Reg16 ← CF + Reg16 + Immed16	ADC BX,1234H
81	17+EA	Mem16 ← CF + Mem16 + Immed16	ADC WORDSOMETHING,1234H
83	4	Reg16 ← CF + Reg16 + Immed8	ADC BX,32
83	17+EA	Mem16 ← CF + Mem16 + Immed8 (Immed8 is sign-extended before add in last 2 cases)	ADC WORDSOMETHING,32

Operation

LeftOpnd ← CF + LeftOpnd + RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

The sum of two operands and the initial state of the carry flag replaces the left operand.

ADD—Integer Addition

Format

Memory/Reg + Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
02	3	Reg8 ← Reg8 + Reg8	ADD BL,CL
02	9+EA	Reg8 ← Reg8 + Mem8	ADD BL,BYTESOMETHING
03	3	Reg16 ← Reg16 + Reg16	ADD BX,CX
03	9+EA	Reg16 ← Reg16 + Mem16	ADD BX,WORDSOMETHING
00	16+EA	Mem8 ← Mem8 + Reg8	ADD BYTESOMETHING,BL
01	16+EA	Mem16 ← Mem16 + Reg16	ADD WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
04	4	AL ← AL + Immed8	ADD AL,5
05	4	AX ← AX + Immed16	ADD AX,400H

Immed to Memory/Reg

Opcode	ModRm*			Data	

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 + Immed8	ADD BL,32
80	17+EA	Mem8 ← Mem8 + Immed8	ADD BYTESOMETHING,32
81	4	Reg16 ← Reg16 + Immed16	ADD BX,1234H
81	17+EA	Mem16 ← Mem16 + Immed16	ADD WORDSOMETHING,1234H
83	4	Reg16 ← Reg16 + Immed8	ADD BX,32
83	17+EA	Mem16 ← Mem16 + Immed8 (Immed8 is sign-extended before add in last 2 cases)	ADD WORDSOMETHING,32

Operation

LeftOpnd ← LeftOpnd + RightOpnd

Flags

```
O D I T S Z A P C
X - - - X X X X X
```

Description

The sum of two operands replaces the left operand.

AND—Logical AND

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
22	3	Reg8 ← Reg8 AND Reg8	AND BL,CL
22	9 + EA	Reg8 ← Reg8 AND Mem8	AND BL,BYTESOMETHING
23	3	Reg16 ← Reg16 AND Reg16	AND BX,CX
23	9 + EA	Reg16 ← Reg16 AND Mem16	AND BX,WORDSOMETHING
20	16 + EA	Mem8 ← Mem8 AND Reg8	AND BYTESOMETHING,BL
21	16 + EA	Mem16 ← Mem16 AND Reg16	AND WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
24	4	AL ← AL AND Immed8	AND AL,4
25	4	AX ← AX AND Immed16	AND AX,400H

Immed to Memory/Reg

Opcode	ModRm*			Data	

* - (Reg field = 100)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 AND Immed8	AND BL,3FH
80	17 + EA	Mem8 ← Mem8 AND Immed8	AND BYTESOMETHING,3FH
81	4	Reg16 ← Reg16 AND Immed16	AND BX,3FFH
81	17 + EA	Mem16 ← Mem16 AND Immed16	AND WORDSOMETHING,3FFH

Operation

LeftOpnd ← LeftOpnd and RightOpnd
OF ← CF ← 0

Flags

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

0 - - - X X U X 0
```

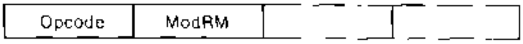
Description

The result of a bitwise logical AND of the two operands replaces the left operand. The carry and overflow flags are cleared.

BOUND—Check Array Against Bounds [iAPX 186]

For 186 clocks, see Appendix H.

Format



Opcode	Operation	Coding Example
62	if Reg16 < Mem16 at EA, or Reg16 > Mem16 at EA - 2 then INTERRUPT 5	BOUND BX,ARRAYFOO 4

Operation

```
if left-operand (a register) < lower-limit (a word variable at EA)
  or left-operand > upper-limit (at EA + 2) then do;
    INTERRUPT 5.
end if;
```

Flags

N	PL	O	D	I	T	S	Z	A	P	C
-	--	-	-	-	-	-	-	-	-	-

Description

BOUND is used to ensure that a signed array index is within the limits defined by a two-word block of memory. This two word block might typically be found just before the array itself and therefore be accessible at a constant offset of −4 from the array, simplifying the addressing. The first word of the block at the effective address contains the lower limit, and the second word contains the upper limit for the index, which is in the register operand of the instruction. The effective address cannot be a register operand   that is, the two-word block cannot be in registers.

CALL—Call

Format

Within segment or group, IP relative

Opcode	DispL	DispH

Opcode	Clocks	Operation	Coding Example
E8	19	IP ← IP + Disp16 —(SP) ← return link	CALL NEAR_LABEL_FOO

Within segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
FF	16	IP ← Reg16 —(SP) ← return link	CALL SI
FF	21 + EA	IP ← Mem16 —(SP) ← return link	CALL WORD PTR [SI]
FF	21 + EA	IP ← Mem16 —(SP) ← return link	CALL POINTER_TO_FRED

Operation

```
if IP-relative then do,
  IP ← IP + Disp16;
  —(SP) ← return link;
else do;
  IP ← (EA);
  —(SP) ← return link;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

There are two types of within-segment or group calls: one that is IP-relative and is specified by the use of a NEAR label as the target address, and one in which the target address is taken from a register or variable pointer without modification (i.e., is NOT IP-relative). In the first case, the 16-bit displacement is relative to the first byte of the next instruction.

The second case is specified when the operand is any (16-bit) general, base, or index register—as in CALL AX, CALL BP, or CALL DI, respectively—or when the operand is a word-variable, as in CALL WORD PTR [BP] or CALL OPEN_ROUTINE[BX] (assuming that OPEN_ROUTINE is declared a word array or structure element). When the effective address is a variable, as in the preceding two examples, DS is the implied segment register for all EA's not using BP.

CALL

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

Inter-segment or group, Direct

Opcode	offset	offset	segbase	segbase

Opcode	Clocks	Operation	Coding Example

9A	28	CS ← segbase
		IP ← offset
			CALL FAR LABEL FOO

Operation

CS ← segbase;
IP ← offset;
—(SP) ← return link;

Flags

O D I T S Z A P C

– – – – – – – – –

Inter-segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example

FF	37 + EA	CS ← segbase
		IP ← offset
			CALL DWORD PTR FOO

Operation

CS ← (EA + 2);
JP ← (EA);

Flags

O D I T S Z A P C

– – – – – – – – –

Description

An intersegment or group (long or far) CALL will transfer control by replacing both the values in CS and IP. This effectively transfers control to another segment or group by changing both the base (paragraph number) and offset values.

CBW—Convert Byte to Word

Format

Opcode

Opcode	Clocks	Operation	Coding Example
98	2	convert byte in AL to word in AX	CBW

Operation

```
if (AL AND 80H) = 80H then do;
  AH ← 0FFh
else do;
  AH ← 0
end.
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

CBW converts the byte in AL to a word in AX by sign extension of AL through AH. No flags are affected.

CLC—Clear Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F8	2	clear the carry flag	CLC

Operation

CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	0

Description

CLC clears the carry flag, CF. No other flags are affected.

CLD—Clear Direction Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FC	2	clear direction flag	CLD

Operation

DF ← 0

Flags

```
O D I T S Z A P C
- 0 - - - - - - -
```

Description

CLD clears the direction flag, DF. No other flags are affected.

CLI—Clear Interrupt Enable Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FA	2	clear interrupt flag	CLI

Operation

IF ← 0

Flags

O	D	I	T	S	Z	A	P	C
-	-	0	-	-	-	-	-	-

Description

CLI clears the interrupt enable flag, IF. No other flags are affected.

CMC—Complement Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F5	2	complement carry flag	CMC

Operation

```
if CF = 1 then do;
  CF ← 0
else do;
  CF ← 1
end;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	X

Description

CMC complements the carry flag, CF. No other flags are affected.

CMP—Compare Two Operands

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
3A	3	flags ← Reg8 - Reg8	CMP BL,CL
3A	9 + EA	flags ← Reg8 - Mem8	CMP BL,BYTESOMETHING
3B	3	flags ← Reg16 - Reg16	CMP BX,CX
3B	9 + EA	flags ← Reg16 - Mem16	CMP BX,WORDSOMETHING
38	9 + EA	flags ← Mem8 - Reg8	CMP BYTESOMETHING,BL
39	9 + EA	flags ← Mem16 - Reg16	CMP WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
3C	4	Flags ← AL - Immed8	CMP AL,5
3D	4	Flags ← AX - Immed16	CMP AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 111)

Opcode	Clocks	Operation	Coding Example
80	4	flags ← Reg8 - Immed8	CMP BL,32
80	10 + EA	flags ← Mem8 - Immed8	CMP BYTESOMETHING,32
81	4	flags ← Reg16 - Immed16	CMP BX,1234H
81	10 + EA	flags ← Mem16 - Immed16	CMP WORDSOMETHING,1234H
83	4	flags ← Reg16 - Immed8	CMP BX,32
83	10 + EA	flags ← Mem16 - Immed8	CMP WORDSOMETHING,32
		(Immed 8 is sign-extended before sub in last 2 cases)	

Operation

flags ← LeftOpnd - RightOpnd

Flags

```
O D I T S Z A P C
X - - - X X X X X
```

Description

The flags are set by the subtraction of the right operand from the left operand. Neither operand is modified. A table of signed and unsigned comparisons supported by conditional jumps is provided under the 'Jcond' heading of this chapter.

CWD—Convert Word to Doubleword

Format

Opcode

Opcode	Clocks	Operation	Coding Example
99	5	convert word in AX to doubleword in DX:AX	CWD

Operation

```
if (AX AND 8000H) = 8000H then do;
  DX ← 0FFFFH
else do;
  DX ← 0
end;
```

Flags

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

Description

CWD converts the word in AX to a doubleword in DX:AX by sign extension of AX through DX. No flags are affected.

CWD

Format

Opcode

Opcode	Clocks	Operation	Coding Example
27	4	adjust AL. flags AH	DAA

Operation

```
if (AL & 0FH) > 9 or AF – 1 then do:
   AL ← AL + 6
   AF ← 1
end;
if AL > 9F or CF = 1 then do;
   AL ← AL + 60H
   CF ← 1
end.
```

Flags

O	D	I	T	S	Z	A	P	C
U	-	-	-	X	X	X	X	X

Description

DAA is used to correct the result of adding two bytes, each of which contains two packed BCD digits, in order to produce a packed decimal result. After the normal byte addition in AL, DAA tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then the low nibble of AL is increased by 6 to produce the correct decimal digit, and the high nibble of AL is incremented, effecting the digit carry.

Whether this first adjustment is made or not, a second adjustment is made if AL is greater than 9FH or if the CF is set, indicating a carry out of the high digit. In this case, 60H is added to AL and the CF is set.

DAS—Decimal Adjust for Subtraction

Format

Opcode

Opcode	Clocks	Operation	Coding Example
2F	4	adjust AL, flags, AH	DAS

Operation

```
if (AL & 0FH) > 9 or AF – 1 then do;
  AL ← AL - 6
  AF ← 1
end;
if AL > 9F or CF = 1 then do,
  AL ← AL - 60H
  CF ← 1
end;
```

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	X	X	X	X	X

Description

DAS is used to correct the result of subtracting two bytes, each of which contains two packed BCD digits, in order to produce a packed decimal result. After the normal byte subtraction in AL, DAS tests the auxiliary carry flag (AF), which is set by a carry out of the low nibble of AL. If either the AF is set or the low nibble of AL is greater than 9, then the low nibble of AL is reduced by 6 to produce the correct decimal digit.

Whether this first adjustment is made or not, a second adjustment is made if AL is greater than 9FH or the CF is set, indicating a borrow out of the high digit. In this case, 60H is subtracted from AL and the CF is set.

DEC—Decrement by 1

Format

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
48 + reg	2	Reg16 ← Reg16 - 1	DEC BX

Memory/Byte Register

Opcode	ModRM*		

*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
FE	3	Reg8 ← Reg8 - 1	DEC BL
FE	15+EA	Mem8 ← Mem8 - 1	DEC BYTESOMETHING
FF	15+EA	Mem16 ← Mem16 - 1	DEC WORDSOMETHING

Operation

Operand ← Operand - 1

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

Description

The operand is decremented by 1.

DIV—Unsigned Division

Format

Memory/Reg with AX or DX:AX

Opcode	ModRM*		

*—(Reg field = 110)

Opcode	Clocks	Operation	Coding Example
F6	80-90	AH,AL ← AX / Reg8	DIV BL
F6	(86-96) + EA	AH,AL ← AX / Mem8	DIV BYTESOMETHING
F7	144-162	DX,AX ← DX:AX / Reg16	DIV BX
F7	(150-168) + EA	DX,AX ← DX:AX / Mem16	DIV WORDSOMETHING

Operation

```
if byte-operation then do;
  if AX / divisor > 0FFH then INT 0;
  else do;
    AL ← AX / divisor        /* unsigned division */
    AH ← AX MOD divisor      /* unsigned modulo */
  end if;
else do;                     /* word-operation */
  if DX:AX / divisor > 0FFFFH then INT 0
  else do;
    AX ← DX:AX / divisor     /* unsigned division */
    DX ← DX:AX MOD divisor   /* unsigned modulo */
  end if;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	U	U	U	U	U

Description

Depending on the opcode, either a word in AX is divided by a byte found in a register or memory location, or a doubleword in DX:AX is divided by a word register or memory location. A doubleword dividend is stored with its high word in DX and low word in AX, and the results are: DX gets the unsigned modulo, and AX gets the unsigned quotient. For a word dividend (byte divisor), the dividend is in AX and the results are: AH gets the unsigned modulo, and AL gets the unsigned quotient. In either case, if the result is too big to fit in the designated register (AX or AL) then an interrupt of type 0 is performed to allow the overflow to be handled.

ENTER—High Level Procedure Entry [iAPX 186]

For 186 clocks, see Appendix H.

Format

Opcode	DataL	DataH	Level

Opcode	Operation	Coding Example
C8	build new stack frame	ENTER NUMDYNS.LEXLVL

Operation

```
right-operand    display level
left-operand  = number of bytes of dynamic storage needed by the routine
—(SP) •  BP:
temp ← SP;
if display level > 0 then
   repeat level - 1 times;
      —(SP) ← —(BP):
   end repeat;
   —(SP) ← temp;
end if;
BP ← temp:
SP ← SP - number of dynamics;
```

Flags

N	PL	O	D	I	T	S	Z	A	P	C
–	––	–	–	–	–	–	–	–	–	–

Description

ENTER is used to create the stack frame required by most block-structured high-level languages. The first parameter specifies how many bytes of dynamic storage is to be allocated on the stack for the routine being entered, while the second corresponds to the lexical nesting level of the routine and determines how many stack frame pointers are copied into the new stack frame from the preceding frame. This list of pointers is also known as the DISPLAY. BP is used as the current stack frame pointer. ENTER first pushes BP and saves the address of the BP-save for later use. If the lexical level is greater than 0, then the list of outer frame pointers from the preceding frame is copied to the new frame, the stack is marked with the temporary holding the address of the top of this list, and BP is set to the current value of SP. Then the dynamics are allocated by subtracting the number of bytes of dynamics from SP.

ESC—Escape

Format

Opcode +i	ModRM		

Opcode	Clocks	Operation	Coding Example
D8+i	8+EA	data bus ← (EA)	ESC 6,ARRAY
D8+i	2	data bus ← (EA)	ESC 20,AL

Operation

if mod ≠ 11 then data bus ← (EA)
if mod = 11 then no operation

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The ESC instruction provides a mechanism by which other processors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 processor does no operation for the ESC instruction other than to access a memory operand and place it on the bus.

HLT—Halt

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F4	2	halt operation	HLT

Operation

cease operation;

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The HLT instruction causes the 8086/8088 processor to enter its halt state. The halt state is cleared by an enable interrupt or reset.

IDIV—Signed Division

Format

Memory/Reg with AX or DX:AX

Opcode	ModRM*		

*—(Reg field =111)

Opcode	Clocks	Operation	Coding Example
F6	101-112	AH,AL ← AX / Reg8	IDIV BL
F6	(107-118) + EA	AH,AL ← AX / Mem8	IDIV BYTESOMETHING
F7	165-184	DX,AX ← DX:AX / Reg16	IDIV BX
F7	(171-190) + EA	DX,AX ← DX:AX / Mem16	IDIV WORDSOMETHING

Operation

```
if byte-operation then do;
  if AX / divisor > 7FH or AX / divisor ← 80H then INT 0;
  else do,
    AL ← AX / divisor          /* signed division */
    AH ← AX MOD divisor        /* signed modulo */
  end if;
else do,                       /* word-operation */
  if DX:AX / divisor > 7FFFH or DX:AX / divisor ← 8000H then INT 0;
  else do;
    AX ← DX:AX / divisor       /* signed division */
    DX ← DX:AX MOD divisor /* signed modulo */
  end if;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
U	–	–	–	U	U	U	U	U

Description

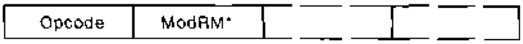
Depending on the opcode, either a word in AX is divided by a byte in a register or memory location, or a dword in DX:AX is divided by a word register or memory location. A dword dividend is stored with its high word in DX and low word in AX, and the results are: DX gets the signed modulo, and AX gets the signed quotient. For a word dividend (byte divisor) the dividend is in AX, and the results are: AH gets the signed modulo, and AL gets the signed quotient. In either case, if the result is too big to fit in the designated register (AX or AL) then an interrupt of type 0 is performed to allow the overflow to be handled.

IMUL—Signed Multiplication

For 186 clocks, see Appendix H.

Format

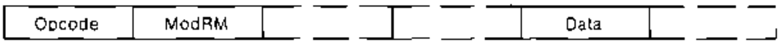
Memory/Reg with AL or AX



*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
F6	80-98	AX ← AL * Reg8	IMUL BL
F6	(86-104)+EA	AX ← AL * Mem8	IMUL BYTESOMETHING
F7	128-154	DX:AX ← AX * Reg16	IMUL BX
F7	(134-160)+EA	DX:AX ← AX * Mem16	IMUL WORDSOMETHING

Mem/Reg * Immediate to Reg [iAPX 186]



Opcode	Operation	Coding Example
6B	Reg 16 ← Reg 16 * Immed 8	IMUL BX,SI,5
6B	Reg 16 ← Reg 16 * Immed 8	IMUL BX,5 ;product → BX
6B	Reg 16 ← Mem 16 * Immed 8	IMUL BX,WORDSMTHING,5
69	Reg 16 ← Reg 16 * Immed 16	IMUL BX,SI,400H
69	Reg 16 ← Reg 16 * Immed 16	IMUL BX,400H ;product → BX
69	Reg 16 ← Mem 16 * Immed 16	IMUL BX,WORDSMTHING,400H

Operation

```
if byte-operation then do;          /* byte operation, word result */
   AX ← AL * (Mem8 or Reg8);
   if AH is a sign extension of AL then CY ← OF ← 0;
   else CY ← OF ← 1;
else if word-operation then do;     /* word-operation, dword result */
   DX:AX ← AX * (Mem16 or Reg16);
   if DX is a sign extension of AX then CY ← OF ← 0;
   else CY ← OF ← 1;
else do;                            /* immed-operation, word result */
   Reg16 ← Immed16 * (Mem16 or Reg16);
   if product fits in destination register then CY ← OF ← 0;
   else CY ← OF ← 1;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	–	–	–	U	U	U	U	X

IMUL

Description

There are two types of integer (signed) multiplication in the ASM86, distinguishable by the types of operands and the precision of the result:

1. Multiply a byte memory or register operand by a byte in AL, producing a word result in AX (called 'byte-operation, word result' above).
2. Multiply a word memory or register operand by a word in AX, producing a dword result in DX:AX (called 'word-operation, dword result' above).

There is a third type of integer (signed) multiplication in the iAPX 186, distinguishable by the types of operands and the precision of the result:

3. Multiply a word memory or register operand by a word (or byte, which will be sign-extended to a word) of immediate data, producing a word result in a register. This instruction uses the full capability of the MODRM byte; therefore the destination need not be the same register as contained the multiplicand. For example, IMUL BX,SI,5 will multiply the contents of the SI register by 5 and leave the (word) result in BX (called 'immed-operation, word result' above).

IN—Input Byte, Word

Format

Fixed port

Opcode	Port

Opcode	Clocks	Operation	Coding Example
E4	10	AL ← Port8	IN AL,BYTEPORTNUMBER
E5	10	AX ← Port8	IN AL,BYTEPORTNUMBER

Variable port

Opcode

Opcode	Clocks	Operation	Coding Example
EC	8	AL ← Port16(in DX)	IN AL,DX
ED	8	AX ← Port16(in DX)	IN AX,DX

Operation

```
if fixed-port then
   portnumber in instruction;
   0 ≤ portnumber ≤ 0FFH;
else
   portnumber in DX;
   0 ≤ portnumber ≤ 0FFFFH;
end if;
if byte-input then AL ← ioport[portnumber],
else AX ← ioport[portnumber];
```

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

IN transfers a byte or word from the specified input port to AL or AX. Use of the fixed port format allows access to ports 0 through FF, and encodes the port number in the instruction. To use the variable port format you load the DX register with a 16 bit port number and then code the mnemonic 'DX' in place of a constant port number. This format allows access to 64k ports.

INC—Increment By 1

Format

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
40 + reg	2	Reg16 ← Reg16 + 1	INC BX

Memory/Byte Register

Opcode	ModRM*		

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
FE	3	Reg8 ← Reg8 + 1	INC BL
FE	15 + EA	Mem8 ← Mem8 + 1	INC BYTESOMETHING
FF	15 + EA	Mem16 ← Mem16 + 1	INC WORDSOMETHING

Operation

Operand ← Operand + 1

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	-

Description

The operand is incremented by 1.

INT/INTO—Interrupt

Format

Opcode	type

Opcode	Clocks	Operation	Coding Example
CC	52	Interrupt 3	INT 3
CD	51	Interrupt 'type'	INT 5
CE	53 or 4	Interrupt 4 if FLAGS OF 1, else NOP	INTO

Operation

```
SP ← SP - 2
—(SP) ← FLAGS
IF ← 0
TF ← 0
SP ← SP - 2
—(SP) ← CS
CS ← TYPE * 4 + 2
SP ← SP - 2
—(SP) ← IP
IP ← TYPE * 4
```

Flags

O	D	I	T	S	Z	A	P	C
-	-	0	0	-	-	-	-	-

Description

INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. The one-byte form of this instruction generates a type 3 interrupt.

INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 (location 10H) if the OF flag is set (trap on overflow). If the OF flag is clear, no operation takes place.

IRET—Return from Interrupt

Format

Opcode

Opcode	Clocks	Operation	Coding Example
CF	24	Return from interrupt	IRET

Operation

```
IP ← (SP) + +
SP ← SP + 2
CS ← (SP) + +
SP ← SP + 2
FLAGS ← (SP) + +
SP ← SP + 2
```

Flags

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

Description

IRET returns control to an interrupted routine by transferring control to the return address saved by a previous interrupt operation and restoring the saved flag registers (as in POPF).

Jcond—Jump on Condition

Operation

```
if condition is true then do;
   sign-extend displacement to 16 bits;
   IP ← IP + sign-extended displacement;
   end if,
```

Format

Opcode	Disp

Opcode	Clocks	Operation	Coding Example	
77	16 or 4	jump if above	JA	TARGETLABEL (CF OR ZF)=0
73	16 or 4	jump if above or equal	JAE	TARGETLABEL CF=0
72	16 or 4	jump if below	JB	TARGETLABEL CF=1
76	16 or 4	jump if below or equal	JBE	TARGETLABEL (CF OR ZF)=1
72	16 or 4	jump if carry set	JC	TARGETLABEL CF=1
74	16 or 4	jump if equal	JE	TARGETLABEL ZF=1
7F	16 or 4	jump if greater	JG	TARGETLABEL ((SF XOR OF) OR ZF)=0
7D	16 or 4	jump if greater or equal	JGE	TARGETLABEL (SF XOR OF)=0
7C	16 or 4	jump if less	JL	TARGETLABEL (SF XOR OF)=1
7E	16 or 4	jump if less or equal	JLE	TARGETLABEL ((SF XOR OF) OR ZF)=1
76	16 or 4	jump if not above	JNA	TARGETLABEL (CF OR ZF)=1
72	16 or 4	jump if neither above nor equal	JNAE	TARGETLABEL CF=1
73	16 or 4	jump if not below	JNB	TARGETLABEL CF=0
77	16 or 4	jump if neither below nor equal	JNBE	TARGETLABEL (CF OR ZF)=0
73	16 or 4	jump if no carry	JNC	TARGETLABEL CF=0
75	16 or 4	jump if not equal	JNE	TARGETLABEL ZF=0
7E	16 or 4	jump if not greater	JNG	TARGETLABEL ((SF XOR OF) OR ZF)=1
7C	16 or 4	jump if neither greater nor equal	JNGE	TARGETLABEL (SF XOR OF)=1
7D	16 or 4	jump if not less	JNL	TARGETLABEL (SF XOR OF)=0
7F	16 or 4	jump if neither less nor equal	JNLE	TARGETLABEL ((SF XOR OF) OR ZF)=0
71	16 or 4	jump if no overflow	JNO	TARGETLABEL OF=0
7B	16 or 4	jump if no parity	JNP	TARGETLABEL PF=0
79	16 or 4	jump if positive	JNS	TARGETLABEL SF=0
75	16 or 4	jump if not zero	JNZ	TARGETLABEL ZF=0
70	16 or 4	jump if overflow	JO	TARGETLABEL OF=1
7A	16 or 4	jump if parity	JP	TARGETLABEL PF=1
7A	16 or 4	jump if parity even	JPE	TARGETLABEL PF=1
7B	16 or 4	jump if parity odd	JPO	TARGETLABEL PF=0
78	16 or 4	jump if sign	JS	TARGETLABEL SF=1
74	16 or 4	jump if zero	JZ	TARGETLABEL ZF=1
E3	18 or 6	jump if CX is zero (does not test flags)	JCXZ	TARGETLABEL

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Jcond

Description

Conditional jumps (except for JCXZ, explained below) test the flags, which presumably have been set in some meaningful way by a previous instruction. Because there are, in many instances, several meaningful and useful ways to interpret a particular state of the flags, ASM86 allows different mnemonics for each interpretation to resolve to the same op-code. This means that some op-codes are, in effect, synonyms for others. As an example, consider that a programmer who has just compared a character to another in AL might wish to jump if the two were equal (JE), while another who had just ANDed AX with a bit field mask would prefer to consider only whether the result was zero or not (he would use JZ, a synonym for JE).

JCXZ differs from the other conditional jumps in that it actually tests the contents of the CX register for zero, rather than interrogating the flags. This instruction is useful following a conditionally repeated string operation (REPE SCASB for example) or conditional loop instruction (such as LOOPNE TARGETLABEL), both of which may terminate for either of two reasons. These instructions implicitly use a limiting count in the CX register, and looping (or repeating) ends either when the CX register goes to zero or when the condition specified in the instruction (flags indicating equals in both of the above cases) occurs. JCXZ is useful when the two terminations must be handled differently.

In every case, if the condition specified in the conditional jump is true, the signed displacement byte is sign extended to a word and added to the IP, which has been updated to point to the first byte of the next instruction. This limits the range of the conditional jump to 127(decimal) bytes beyond and 126 bytes before the instruction (remember, the IP was incremented by 2 to point to the next instruction before the displacement was added).

JMP—Jump

Format

Within segment or group, IP relative

Opcode	DispL	DispH

Opcode	Clocks	Operation	Coding Example
E9	15	IP ← IP + Disp16	JMP NEAR LABEL FOO
EB	15	IP ← IP + Disp8 (Disp8 sign-extended)	JMP SHORT NR LAB FOO

Within segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
FF	11	IP ← Reg16	JMP SI
FF	18+EA	IP ← Mem16	JMP WORD PTR [SI
FF	18+EA	IP ← Mem16	JMP POINTER TO _FRED

Operation

```
if IP-relative then do;
  if short then sign-extend Disp8 to Disp16;
  IP ← IP + Disp16,
else do;
  IP ← (EA).
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

There are two types of within-segment jumps: one which is IP-relative and is specified by the use of a NEAR label as the target address; and one in which the target address is taken from a register or variable pointer without modification (i.e. is NOT IP-relative). In the first case, the displacement—which is relative to the first byte of the next instruction—may be either a full word or a byte which will be sign-extended to a word.

The second case is specified when the operand is any (16-bit) general, base, or index register—as in JMP AX, JMP BP, or JMP DI, respectively—or when the operand is a word-variable, as in JMP WORD PTR [BP], or JMP CS:CASE_TABLE[BX] (assuming that CASE_TABLE was defined as an array of word pointers). When the effective address is a variable, as in the preceding two examples, DS is the implied segment register for all EA's not using BP. Note especially the difference between JMP BX and JMP [BX]. In the first jump the new IP is taken from a register, while in the second it comes from a word variable which is pointed at by the register.

JMP

Inter-segment or group, Direct

Opcode	offset	offset	segbase	segbase

Opcode	Clocks	Operation	Coding Example

EA	15	CS ← segbase
		IP ← offset
			JMP FAR_LABEL_FOO

Operation

CS ← segbase
IP ← offset

Flags

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

Inter-segment or group, Indirect

Opcode	ModRM*		

*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example

FF	24+EA	CS ← segbase
		IP ← offset
			JMP CASE TABLE[BX]

Operation

CS ← EA.segbase;
IP ← EA.offset;

Flags

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

Description

The long jumps transfer control using both an offset and paragraph number (segbase), which may be either included in the instruction itself or found in a DWORD variable.

LAHF—Load AH From Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9F	4	copy low byte of flags word to AH	LAHF

Operation

```
  AH ← SF:ZF:X:AF:X:PF:X:CF
/* 'x' indicates non-specified bit value */
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The Sign, Zero, Auxiliary carry, Parity, and Carry Flags are transferred to AH in the following format:

SF goes to AH bit7
ZF goes to AH bit6
AF goes to AH bit4
PF goes to AH bit2
CF goes to AH bit0

The remaining bits are indeterminate. No flags are altered.

LDS/LES—Load Pointer to DS/ES and Register

Format

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
C4	16+EA	dword pointer at EA goes to reg16 (1st word) and ES (2nd word)	LES BX,DWORDPOINTER
C5	16+EA	dword pointer at EA goes to reg16 (1st word) and DS (2nd word)	LDS BX,DWORDPOINTER

Operation

```
Reg16 ← Mem16 @ EA              /* offset part of Virtual Address DWord */
DS (or ES) ← Mem16 @ EA + 2     /* selector part of Virtual Address DWord */
```

Flags

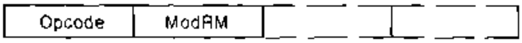
O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The double word in the memory location designated by the effective address and 3 successive bytes is treated as two word operands. The first of these in EA:EA+1 is the offset part of the pointer and is loaded into the designated word-register. The second word, at EA+2:EA+3, is the paragraph number (segment base) of the address, and is loaded into the DS or ES register.

LEA—Load Effective Address

Format



Opcode	Clocks	Operation	Coding Example
8D	2+EA	Reg16 ← EA	LEA BX,SOMEVARIABLE [SI]

Operation

```
if EA = register then UDtrap;
else Reg 16 ← offset(EA)
```

Flags

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

Description

The effective address of the memory operand is put in the specified register. You should use this instruction only if EA requires run time calculation, i.e., has indexing with index or base register. Otherwise, you should use MOV reg, OFFSET variable.

LEAVE—High Level Procedure Exit [iAPX 186]

For 186 clocks, see Appendix H.

Format

Opcode

Opcode	Operation	Coding Example
C9	release current stack frame and return to prior frame.	LEAVE

Operation

```
SP ← BP;          /* burn off dynamics and display */
BP ← (SP)++;      /* recover old frame pointer */
```

Flags

N	PL	O	D	I	T	S	Z	A	P	C
–	––	–	–	–	–	–	–	–	–	–

Description

LEAVE is the complementary operation to ENTER, and reverses the effects of that instruction. By copying BP to SP, LEAVE releases all the stack space used by a routine for its dynamics and display. The old frame pointer is now popped into BP, restoring the caller's frame, and a subsequent RET xx instruction will follow the back link and remove any arguments pushed on the stack for the exiting routine.

LOCK—Assert Bus Lock

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F0	2	assert the bus lock next instruction	LOCK XCHG AX,SEMAPHORE

Operation

None.

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

A special one-byte lock prefix may precede any instruction. It causes the processor to assert its bus-lock signal for the duration of the operation caused by the instruction. In multiple processor systems with shared resources it is necessary to provide mechanisms to enforce controlled access to those resources. Such mechanisms, while generally provided through software operating systems, require hardware assistance. A sufficient mechanism for accomplishing this is a *locked exchange* (also known as test-and-set-lock).

It is assumed that external hardware, upon receipt of that signal, will prohibit bus access for other bus masters during the period of its assertion.

The instruction most useful in this context is an exchange register with memory. A simple software lock may be implemented with the following code sequence:

```
Check:  MOV   AL,1      ;set AL to 1 (implies locked)
  LOCK  XCHG  Sema,AL   ;test and set lock
        TEST  AL, AL    ;set flags based on AL
        JNZ   Check     ;retry if lock already set
        .
        .
        MOV   Sema,0    ;clear the lock when done
```

The LOCK prefix may be combined with the segment override and/or REP prefixes, although the latter has certain problems. (See REP.)

LOOP/LOOPE/LOOPNE/ LOOPZ/LOOPNZ—Loop Control

Format

Opcode	Disp

Opcode	Clocks	Operation	Coding Example
E1	18 or 6	dec CX; loop if equal and CX not 0	LOOPE TARGETLABEL
E0	19 or 5	dec CX; loop if not equal and CX not 0	LOOPNE TARGETLABEL
E1	18 or 6	dec CX; loop if zero and CX not 0	LOOPZ TARGETLABEL
E0	19 or 5	dec CX; loop if not zero and CX not 0	LOOPNZ TARGETLABEL
E2	17 or 5	dec CX; loop if CX not 0	LOOP TARGETLABEL

Operation

```
CX ← CX - 1;
if (condition is true) and (CX <> 0) then do;
  sign-extend displacement to 16 bits;
  IP ← IP + sign-extended displacement;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The LOOP instructions are intended to provide iteration control and combine loop index management with conditional branching. To use the LOOP instruction you load an unsigned iteration count into CX, then code the LOOP at the end of a series of instructions to be iterated. Each time LOOP is executed the CX register is decremented and a conditional branch to the top of the loop is performed. The five variants of the instruction (LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ) allow branching on three sets of conditions, since two pairs of variants are synonymous. Conditions for branching are:

LOOP	branches if CX non-zero after decrementing;
LOOPZ, LOOPE	branch if CX non-zero and ZF = 1;
LOOPNZ, LOOPNE	branch if CX non-zero and ZF = 0.

In every case, if the condition specified in the conditional loop is true, the signed displacement byte is sign extended to a word and added to the IP, which has been updated to point to the first byte of the next instruction. This limits the range of the conditional loop to 127 (decimal) bytes beyond and 126 bytes before the instruction (remember, the IP was incremented by 2 to point to the next instruction before the displacement was added).

MOV—Move Data

Format

Memory/Reg to or from Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
88	9 + EA	Mem8 ← Reg8	MOV BYTESOMETHING,AL
88	2	Reg8 ← Reg8	MOV BL,AL
89	9 + EA	Mem16 ← Reg16	MOV WORDSOMETHING,AX
89	2	Reg16 ← Reg16	MOV BX,AX
8A	8 + EA	Reg8 ← Mem8	MOV AL,BYTESOMETHING
8B	8 + EA	Reg16 ← Mem16	MOV AX,WORDSOMETHING

Direct-Addressed Memory to or from AX/AL

Opcode	AddrL	AddrH

Opcode	Clocks	Operation	Coding Example
A0	10	AL ← Mem8	MOV AL,BYTESOMETHING
A1	10	AX ← Mem16	MOV AX,WORDSOMETHING
A2	10	Mem8 ← AL	MOV BYTESOMETHING,AL
A3	10	Mem16 ← AX	MOV AX,WORDSOMETHING

Immed to Reg

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
B0 + reg	4	Reg 8 ← Immed8	MOV CL,5
B8 + reg	4	Reg16 ← Immed16	MOV SI,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
C6	4	Reg8 ← Immed8	MOV BL,32
C6	10 + EA	Mem8 ← Immed8	MOV BYTESOMETHING,32
C7	4	Reg16 ← Immed16	MOV BX,1234H
C7	10 + EA	Mem16 ← Immed16	MOV WORDSOMETHING,1234H

Memory/Reg to or from SReg

Opcode	ModRM*		

*—(Reg field = SReg)

MOV

Opcode	Clocks	Operation	Coding Example
8C	9+EA	Mem16 ← SReg	MOV WORDSOMETHING,DS
8C	2	Reg16 ← SReg	MOV AX,DS
8E	8+EA	SReg* ← Mem16	MOV DS,WORDSOMETHING
8E	2	SReg* ← Reg16	MOV DS,AX
		*CS not allowed	

Operation

LeftOpnd ← RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

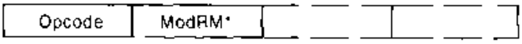
Description

The right operand (source) is copied to the left operand (destination). The right operand is not modified. No flags are affected.

MUL—Unsigned Multiplication

Format

Memory/Reg with AL or AX



*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
F6	70-77	AX ← AL * Reg8	MUL BL
F6	(76-83) + EA	AX ← AL * Mem8	MUL BYTESOMETHING
F7	118-133	DX:AX ← AX * Reg16	MUL BX
F7	(124-139) + EA	DX:AX ← AX * Mem16	MUL WORDSOMETHING

Operation

```
if byte-operation then do;          /* byte operation, word result */
   AX ← AL * (Mem8 or Reg8);
   if AH = 0 then CY ← OF ← 0;
   else CY ← OF ← 1;
else if word-operation then do;     /* word-operation, dword result */
   DX:AX ← AX * (Mem16 or Reg16);
   if DX = 0 then CY ← OF ← 0;
   else CY ← OF ← 1;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	U	U	U	U	X

Description

There are two types of unsigned multiplication in the 8086/8088, distinguishable by the types of operands and the precision of the result:

1. Multiply a byte memory or register operand by a byte in AL, producing a word result in AX (called 'byte-operation, word result' above).
2. Multiply a word memory or register operand by a word in AX, producing a dword result in DX:AX (called 'word-operation, dword result' above).

In both types of multiply the carry and overflow flags are used to signal whether the product has exceeded the precision of the operands which produced it. Thus, when multiplying two bytes, if the product is larger than can be expressed in a byte (i.e. prod > 256.) then the CY and OF flags will be set; otherwise, they will be cleared.

NEG—Negate an Integer

Format

Memory/Reg

Opcode	ModRM*		

*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
F6	3	Reg8 ← 00H - Reg 8	NEG BL
F7	3	Reg16 ← 0000H - Reg16	NEG BX
F6	16 + EA	Mem8 ← 00H Mem8	NEG BYTESOMETHING
F7	16 + EA	Mem16 ← 0000H - Mem16	NEG WORDSOMETHING

Operation

Operand ← 2's complement of Operand

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	1*

*except when operand is zero, then CF ← 0

Description

The two's complement of the register or memory operand replaces the old operand value.

NOP—No Operation

Format

Opcode

Opcode	Clocks	Operation	Coding Example
90	3	no operation	NOP

Operation

Perform no operation.

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

NOP is a one-byte filler instruction which takes up space but affects none of the machine context except IP.

NOT—Form One's Complement

Format

Memory/Reg

Opcode	ModRM*		

*—(Reg field = 010)

Opcode	Clocks	Operation	Coding Example
F6	3	Reg8 ← 0FFH - Reg8	NOT BL
F6	16 + EA	Mem8 ← 0FFH - Mem8	NOT BYTESOMETHING
F7	3	Reg16 ← 0FFFFH - Reg16	NOT BX
F7	16 + EA	Mem16 ← 0FFFFH - Mem16	NOT WORDSOMETHING

Operation

Operand ← one's complement of Operand

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The operand is inverted, that is, every 1 becomes a 0 and vice versa.

OR—Logical Inclusive OR

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
0A	3	Reg8 ← Reg8 OR Reg8	OR BL,CL
0A	9 + EA	Reg8 ← Reg8 OR Mem8	OR BL,BYTESOMETHING
0B	3	Reg16 ← Reg16 OR Reg16	OR BX,CX
0B	9 + EA	Reg16 ← Reg16 OR Mem16	OR BX,WORDSOMETHING
08	16 + EA	Mem8 ← Mem8 OR Reg8	OR BYTESOMETHING,BL
09	16 + EA	Mem16 ← Mem16 OR Reg16	OR WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
0C	4	AL ← AL OR Immed8	OR AL,5
0D	4	AX ← AX OR Immed16	OR AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 OR Immed8	OR BL,32
80	17 + EA	Mem8 ← Mem8 OR Immed8	OR BYTESOMETHING,32
81	4	Reg16 ← Reg16 OR Immed16	OR BX,1234H
81	17 + EA	Mem16 ← Mem16 OR Immed16	OR WORDSOMETHING,1234H

Operation

LeftOpnd ← LeftOpnd or RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The inclusive OR of two operands replaces the left operand. The carry and overflow flags are cleared.

OUT—Output Byte, Word

Format

Fixed port

Opcode	Port

Opcode	Clocks	Operation	Coding Example
E6	10	Port8 ← AL	OUT BYTEPORTNUMBER AL
E7	10	Port8 ← AX	OUT BYTEPORTNUMBER AX

Variable port

Opcode

Opcode	Clocks	Operation	Coding Example
EE	8	Port16 (in DX) ← AL	OUT DX,AL
EF	8	Port16 (in DX) ← AX	OUT DX,AX

Operation

```
if fixed-port then
   portnumber in instruction;
   0 ≤ portnumber ≤ 0FFH;
else
   portnumber in DX;
   0 ≤ portnumber ≤ 0FFFFH;
end if;
if byte-output then ioport[portnumber] ← AL;
else ioport[portnumber] ← AX;
```

Flags

```
O D I T S Z A P C
– – – – – – – – –
```

Description

OUT transfers a byte from AL or a word from AX to the specified output port. Use of the fixed port format allows access to ports 0 through FF, and encodes the port number in the instruction. To use the variable port format you load the DX register with a 16 bit port number and then code the mnemonic 'DX' in place of a constant port number. This format allows access to 64k ports.

POP—Pop a Word From the Stack

Format

Word Memory

Opcode	ModRM*		

*—(Reg field=000)

Opcode	Clocks	Operation	Coding Example
8F	17+EA	Mem16 ← (SP)++	POP WORDSOMETHING

Word Register

Opcode+reg

Opcode	Clocks	Operation	Coding Example
58+reg	8	Reg16 ← (SP)++	POP BX

Segment Register

Opcode+SReg

Opcode	Clocks	Operation	Coding Example
07+(SReg*8)	8	SReg ← (SP)++	POP DS

Operation

Operand ← TOS,
SP ← SP + 2;

Flags

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

Description

The word on the top of the stack replaces the previous contents of the memory, register, or segment register operand. The stack pointer is incremented by 2 to point to the new top of stack.

If the destination operand is a segment register, the value POPed will be a paragraph number.

POP CS is NOT allowed.

POPA—Pop All Registers [iAPX 286]

For 186 clocks, see Appendix H.

Format

Opcode

Opcode	Operation	Coding Example
61	restore registers from the stack	POPA

Operation

```
DI ← (SP) + + ;
SI ← (SP) + + ,
BP ← (SP) + + :
SP ← SP + 2;          /* POP AND IGNORE SP */
BX ← (SP) + + ;
DX ← (SP) + + ;
CX ← (SP) + + ;
AX ← (SP) + + ;
```

Flags

N	PL	O	D	I	T	S	Z	A	P	C
-	--	-	-	-	-	-	-	-	-	-

Description

POPA restores the registers pushed by PUSHA, except that the SP value is ignored.

POPF—Pop the TOS into the Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9D	8	FLAGS ← (SP)++	POPF

Operation

Flags ← TOS;
SP ← SP + 2;

Flags

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

Description

The TOS is copied to the Flags and the stack pointer is incremented by 2 to point to the new top of stack. Bit position to flag assignments are:

OF	← bit 11
DF	← bit 10
IF	← bit 9
TF	← bit 8
SF	← bit 7
ZF	← bit 6
AF	← bit 4
PF	← bit 2
CF	← bit 0

PUSH—Push a Word Onto the Stack

For 186 clocks, see Appendix H.

Format

Memory/Reg

Opcode	ModRM*		

*—(Reg field=110)

Opcode	Clocks	Operation	Coding Example
FF	16 + EA	(SP) ← Mem16	PUSH WORDSOMETHING

Word Register

Opcode + reg

Opcode	Clocks	Operation	Coding Example
50 + reg	11	-(SP) ← Reg16	PUSH BX

Word Immediate [iAPX 186]

Opcode	Data	

Opcode	Operation	Coding Example
6A	(SP) ← Immed8 (sign extended)	PUSH 5
68	—(SP) ← Immed16	PUSH 400H

Segment Register

Opcode + SReg

Opcode	Clocks	Operation	Coding Example
06 + (SReg*8)	10	(SP) ← SReg	PUSH DS

Operation

```
SP ← SP - 2;
TOS ← Operand;
```

PUSH

Flags

```
O D I T S Z A P C
- - - - - - - - -
```

Description

The stack pointer is decreased by 2 and the word operand is copied to the new top of stack.

PUSHA—Push All Registers [iAPX 186]

For 186 clocks, see Appendix H.

Format

Opcode

Opcode	Operation	Coding Example
60	save registers on the stack	PUSHA

Operation

```
temp ← SP;
—(SP) ← AX;
—(SP) ← CX;
—(SP) ← DX,
—(SP) ← BX;
—(SP) ← temp;
—(SP) ← BP.
—(SP) ← SI.
—(SP) ← DI;
```

Flags

N	PL	O	D	I	T	S	Z	A	P	C
-	--	-	-	-	-	-	-	-	-	-

Description

PUSHA saves the registers noted above on the stack.

PUSHF—Push the Flags to the Stack

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9C	10	—(SP) ← FLAGS	PUSHF

Operation

SP ← SP - 2;
TOS ← Flags;

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The stack pointer is decremented by 2 and the flags are copied to the new top of stack. Flag to bit position assignments are:

bit 11 ← OF
bit 10 ← DF
bit 9 ← IF
bit 8 ← TF
bit 7 ← SF
bit 6 ← ZF
bit 4 ← AF
bit 2 ← PF
bit 0 ← CF

RCL—Rotate Left Through Carry

For 186 clocks, see Appendix H.

Format

Memory or Reg by 1

Opcode	ModRM*		

*—(Reg field=010)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg 8 by 1	RCL BL,1
D0	15+EA	rotate Mem8 by 1	RCL BYTESOMETHING,1
D1	2	rotate Reg 16 by 1	RCL BX,1
D1	15+EA	rotate Mem16 by 1	RCL WORDSOMETHING,1

Memory or Reg by count in CL

Opcode	ModRM*		

*—(Reg field=010)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	RCL BL,CL
D2	20+EA+4/bit	rotate Mem8 by CL	RCL BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	RCL BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	RCL WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]

Opcode	ModRM*			count

*—(Reg field - 011)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	RCL BL,5
C0	rotate Mem8 by Immed8	RCL BYTESOMETHING,5
C1	rotate Reg16 by Immed8	RCL BX,5
C1	rotate Mem16 by Immed8	RCL WORDSOMETHING,5

Operation

```
if variable-bit-rotate then count=CL or count=Immed8;
else count=1;
do until count=0
  tempcf ← CF;
  CF ← high-order-bit of operand;
  operand ← operand * 2 + tempcf;
  count ← count - 1.
```

RCL

```
end do;
if not variable-bit-rotate then do;
   if high-order-bit of operand < > CF then OF ← 1;
   else OF ← 0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Description

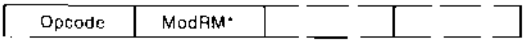
The register or memory operand is rotated left through the CF according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the resulting carry flag. Only CF and OF are affected.

RCR—Rotate Right Through Carry

For 186 clocks, see Appendix H.

Format

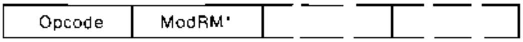
Memory or Reg by 1



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	RCR BL,1
D0	15+EA	rotate Mem8 by 1	RCR BYTESOMETHING,1
D1	2	rotate Reg16 by 1	RCR BX,1
D1	15+EA	rotate Mem16 by 1	RCR WORDSOMETHING,1

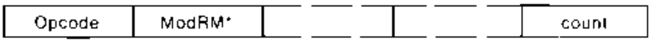
Memory or Reg by count in CL



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	RCR BL,CL
D2	20+EA+4/bit	rotate Mem8 by CL	RCR BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	RCR BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	RCR WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]



*—(Reg field = 011)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	RCR BL,5
C0	rotate Mem8 by Immed8	RCR BYTESOMETHING,5
C1	rotate Reg16 by Immed8	RCR BX,5
C1	rotate Mem16 by Immed8	RCR WORDSOMETHING,5

Operation

```
if variable-bit-rotate then count=CL or count=Immed8;
else do;
  count=1;
  if high-order-bit of operand <> CF then OF ← 1;
  else OF ← 0;
end if;
do until count=0
  tempcf ← CF;
  CF ← low-order-bit of operand;
```

RCR

```
   operand ←  operand / 2,
   high-order-bit of operand ← tempcf;
   count ← count - 1;
end do;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Description

The register or memory operand is rotated right through the CF according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. If the shift count is 1, the overflow flag is set if the high bit of the un-rotated operand differs from the original carry flag. Only CF and OF are affected.

REP/REZ/REPE/REPNE/REPNZ—Repeat Prefix

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F3	2	repeat next instruction until CX-0	REP MOVSB
F3	2	repeat next instruction until CX-0 or ZF=1	REPE SCASB REPZ SCASB
F2	2	repeat next instruction until CX-0 or ZF 0	REPNE SCASB REPNZ SCASB

Operation

```
do while CX <> 0;
   /* acknowledge pending interrupts */
   /* perform string operation in subsequent byte */
   CX ← CX - 1;  /* does not affect flags */
   if string operation = SCAS or CMPS and
      ZF <> repeat condition then undo;
end do;
```

Flags

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

Description

The REP prefix causes a succeeding string operation to be repeated until the count in CX goes to zero (REP causes CX to be decremented after each repetition of the string op). If the string operation is either SCAS or CMPS (or a variant of those such as SCASB...) then the ZF is compared to the repeat condition after the string op is performed, and the repeat is terminated if the ZF does not match the condition. For example, REPE SCASB will scan a string, comparing each byte to the AL register, as long as the ZF is 1, indicating 'EQUAL'.

REP, REPE, and REPZ are synonymous, as are REPNZ and REPNE.

Execution of the repeated string operation will not resume properly following an interrupt if more than one prefix is present preceding the string primitive. Execution will resume one byte before the primitive (presumably where the repeat resides), thus ignoring the additional prefixes.

RET—Return From Subroutine

Format

Opcode

Opcode	Clocks	Operation	Coding Example
C3	8	intra-segment return	RET
CB	18	inter-segment return	RET

Return and add constant to SP

Opcode	DataL	DataH

Opcode	Clocks	Operation	Coding Example
C2	12	intra-segment ret and add	RET 8
CA	17	inter-segment ret and add	RET 8

Operation

```
IP ← (SP) + + ;
SP ← SP + 2;
if intersegment then
  CS ← (SP) + + ;
  SP ← SP + 2;
if add immediate to SP then
  SP ← SP + immediate constant;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

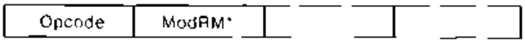
RET transfers control through a back-link on the stack, reversing the effects of a CALL instruction. If the intra-segment RET is used, the back-link is assumed to be just the return-IP, while inter-segment RETs assume both IP and CS are on the stack. RETs may optionally add a constant to the stack pointer, effectively removing any arguments to the called routine which were pushed prior to the CALL.

ROL—Rotate Left

For 186 clocks, see Appendix H.

Format

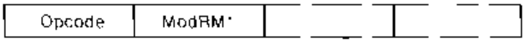
Memory or Reg by 1



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	ROL BL,1
D0	15+EA	rotate Mem8 by 1	ROL BYTESOMETHING,1
D1	2	rotate Reg16 by 1	ROL BX,1
D1	15+EA	rotate Mem16 by 1	ROL WORDSOMETHING,1

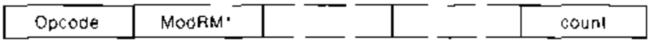
Memory or Reg by count in CL



*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	rotate Reg8 by CL	ROL BL,CL
D2	20+Ea+4/bit	rotate Mem8 by CL	ROL BYTESOMETHING,CL
D3	8+4/bit	rotate Reg16 by CL	ROL BX,CL
D3	20+EA+4/bit	rotate Mem16 by CL	ROL WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]



*—(Reg field = 000)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	ROL BL,5
C0	rotate Mem8 by Immed8	ROL BYTESOMETHING,5
C1	rotate Reg16 by Immed8	ROL BX,5
C1	rotate Mem16 by Immed8	ROL WORDSOMETHING,5

Operation

```
if variable-bit-rotate then count=CL or count=Immed8;
else count=1;
do until count=0
  CF ← high-order-bit of operand;
  operand ← operand * 2 + CF;
  count ← count - 1;
```

ROL

```
end do,
if not variable-bit-rotate then do;
   if high-order-bit of operand <> CF then OF ← 1;
   else OF ← 0;
end if;
```

Flags

```
O D I T S Z A P C
X - - - - - - - X
```

Description

The register or memory operand is rotated left according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. The high order bit of the operand is copied directly to the low order bit during the rotate, as well as to CF. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the resulting carry flag. (That is, if the high and low order bits of the result are not the same.) Only CF and OF are affected.

ROR—Rotate Right

For 186 clocks, see Appendix H.

Format

Memory or Reg by 1

Opcode	ModRM*		

*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
D0	2	rotate Reg8 by 1	ROR BL,1
D0	15 + EA	rotate Mem8 by 1	ROR BYTESOMETHING,1
D1	2	rotate Reg16 by 1	ROR BX,1
D1	15 + EA	rotate Mem16 by 1	ROR WORDSOMETHING,1

Memory or Reg by count in CL

Opcode	ModRM*		

*—(Reg field = 001)

Opcode	Clocks	Operation	Coding Example
D2	8 + 4/bit	rotate Reg8 by CL	ROR BL,CL
D2	20 + EA + 4/bit	rotate Mem8 by CL	ROR BYTESOMETHING,CL
D3	8 + 4/bit	rotate Reg16 by CL	ROR BX,CL
D3	20 + EA + 4/bit	rotate Mem16 by CL	ROR WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]

Opcode	ModRM*			count

*—(Reg field = 001)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	ROR BL,5
C0	rotate Mem8 by Immed8	ROR BYTESOMETHING,5
C1	rotate Reg16 by Immed8	ROR BX,5
C1	rotate Mem16 by Immed8	ROR WORDSOMETHING,5

Operation

```
if variable-bit-rotate then count=CL or count=Immed8;
else count = 1;
do until count = 0
  tempcf ← CF;
  CF ← low-order-bit of operand;
  operand ← operand / 2;
  high-order-bit of operand ← CF;
```

ROR

```
   count ← count - 1;
end do;
if not variable-bit-rotate then do;
   if high-order-bit of operand <> CF then OF ←1;
   else OF ←0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	-	-	-	-	X

Description

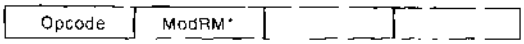
The register or memory operand is rotated right according to the shift count, which may be either a fixed count of 1 or a variable count that has been loaded into the CL register. The low bit of the operand is copied directly to the high bit during the rotate, as well as to the CF. If the shift count is 1, the overflow flag is set if the high bit of the rotated operand differs from the un-rotated high bit. Only CF and OF are affected.

SAL/SHL—Arithmetic/Logical Left Shift

For 186 clocks, see Appendix H.

Format

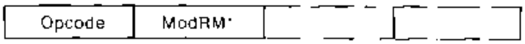
Memory or Reg by 1



*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SAL BL,1
D0	15+EA	shift Mem8 by 1	SHL BYTESOMETHING,1
D1	2	shift Reg16 by 1	SHL BX,1
D1	15+EA	shift Mem16 by 1	SAL WORDSOMETHING,1

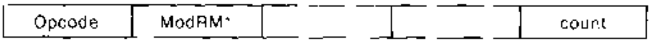
Memory or Reg by count in CL



*—(Reg field = 100)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	shift Reg8 by CL	SHL BL,CL
D2	20+EA+4/bit	shift Mem8 by CL	SAL BYTESOMETHING,CL
D3	8+4/bit	shift Reg16 by CL	SAL BX,CL
D3	20+EA+4/bit	shift Mem16 by CL	SHL WORDSOMETHING,CL

Mem or Reg by immediate count [iAPX 186]



*—(Reg field = 100)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	SHL BL,5
C0	rotate Mem8 by Immed8	SAL BYTESOMETHING,5
C1	rotate Reg16 by Immed8	SAL BX,5
C1	rotate Mem16 by Immed8	SHL WORDSOMETHING,5

Operation

```
if variable-bit-shift then count←CL or count←Immed8;
else count=1;
do until count=0
  CF ← high-order-bit of operand;
  operand ← operand * 2;
  count ← count - 1;
end do;
```

SAHF—Store AH in Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9E	4	copy AH to low byte of flags word	SAHF

Operation

AH → SF:ZF:X:AF:X:PF:X:CF

/* 'X' indicates non-specified bit value */

Flags

O	D	I	T	S	Z	A	P	C
-	-	-	-	X	X	X	X	X

Description

The Sign, Zero, Auxiliary carry, Parity, and Carry Flags are loaded from AH in the following format:

AH bit7 goes to SF
AH bit6 goes to ZF
AH bit4 goes to AF
AH bit2 goes to PF
AH bit0 goes to CF

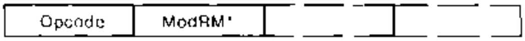
No other flags are altered.

SAR—Arithmetic Right Shift

For 186 clocks, see Appendix H.

Format

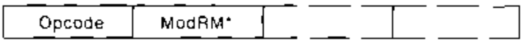
Memory or Reg by 1



*—(Reg field = 111)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SAR BL,1
D0	15+EA	shift Mem8 by 1	SAR BYTESOMETHING,1
D1	2	shift Reg16 by 1	SAR BX,1
D1	15+EA	shift Mem16 by 1	SAR WORDSOMETHING,1

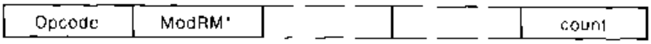
Memory or Reg by count in CL



*— (Reg field = 111)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	shift Reg8 by CL	SAR BL,CL
D2	20+EA+4/bit	shift Mem8 by CL	SAR BYTESOMETHING,CL
D3	8+4/bit	shift Reg16 by CL	SAR BX,CL
D3	20+EA+4/bit	shift Mem16 by CL	SAR WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]



*—(Reg field = 111)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	SAR BL,5
C0	rotate Mem8 by Immed8	SAR BYTESOMETHING,5
C1	rotate Reg16 by Immed8	SAR BX,5
C1	rotate Mem16 by Immed8	SAR WORDSOMETHING,5

Operation

```
if variable-bit-shift then count=CL or count=Immed8;
else count=1;
do until count=0
  CF ← low-order-bit of operand;
  operand ← operand / 2;    /* SIGNED DIVIDE */
  count ← count - 1;
end do;
```

SAL/SHL

```
if not variable-bit-shift then do;
   if high-order-bit of operand <> CF then OF ← 1;
   else OF ← 0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

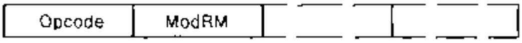
Description

SHL (shift logical left) and SAL (shift arithmetic left) shift the operand left by COUNT bits, shifting in low-order zero bits.

SBB—Integer Subtraction With Borrow

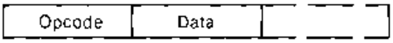
Format

Memory/Reg with Reg



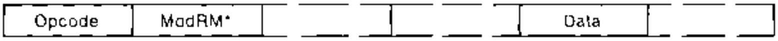
Opcode	Clocks	Operation	Coding Example
1A	3	Reg8 ← Reg8 - Reg8 - CF	SBB BL,CL
1A	9 + EA	Reg8 ← Reg8 - Mem8 - CF	SBB BL,BYTESOMETHING
1B	3	Reg16 ← Reg16 - Reg16 - CF	SBB BX,CX
1B	9 + EA	Reg16 ← Reg16 - Mem16 - CF	SBB BX,WORDSOMETHING
18	16 + EA	Mem8 ← Mem8 - Reg8 - CF	SBB BYTESOMETHING,BL
19	16 + EA	Mem16 ← Mem16 - Reg16 - CF	SBB WORDSOMETHING,BX

Immed from AX/AL



Opcode	Clocks	Operation	Coding Example
1C	4	AL ← AL - Immed8 - CF	SBB AL,5
1D	4	AX ← AX - Immed16 - CF	SBB AX,400H

Immed from Memory/Reg



*—(Reg field = 011)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 - Immed8 - CF	SBB BL,32
80	17 + EA	Mem8 ← Mem8 - Immed8 - CF	SBB BYTESOMETHING,32
81	4	Reg16 ← Reg16 - Immed16 - CF	SBB BX,1234H
81	17 + EA	Mem16 ← Mem16 - Immed16 - CF	SBB WORDSOMETHING,1234H
83	4	Reg16 ← Reg16 - Immed8 - CF	SBB BX,32
83	17 + EA	Mem16 ← Mem16 - Immed8 - CF (Immed8 is sign-extended before subtract)	SBB WORDSOMETHING,32

Operation

LeftOpnd ← LeftOpnd - RightOpnd - CF

SAR

```
if not variable-bit-shift then do:
   OF ← 0;
end if;
```

Flags

O	D	I	T	S	Z	A	P	C
X	–	–	–	X	X	U	X	X

Description

SAR (shift arithmetic right) shifts the operand right by COUNT bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

SHR—Logical Right Shift

Format

Memory or Reg by 1

Opcode	ModRM*		

*—(Reg field - 101)

Opcode	Clocks	Operation	Coding Example
D0	2	shift Reg8 by 1	SHR BL,1
D0	15+EA	shift Mem8 by 1	SHR BYTESOMETHING,1
D1	2	shift Reg16 by 1	SHR BX,1
D1	15+EA	shift Mem16 by 1	SHR WORDSOMETHING,1

Memory or Reg by count in CL

Opcode	ModRM*		

*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
D2	8+4/bit	shift Reg8 by CL	SHR BL,CL
D2	20+Ea+4/bit	shift Mem8 by CL	SHR BYTESOMETHING,CL
D3	8+4/bit	shift Reg16 by CL	SHR BX,CL
D3	20+EA+4/bit	shift Mem16 by CL	SHR WORDSOMETHING,CL

Mem or Reg by Immed8 [iAPX 186]

Opcode	ModRM*			count

*—(Reg field - 101)

Opcode	Operation	Coding Example
C0	rotate Reg8 by Immed8	SHR BL,5
C0	rotate Mem8 by Immed8	SHR BYTESOMETHING,5
C1	rotate Reg16 by Immed8	SHR BX,5
C1	rotate Mem16 by Immed8	SHR WORDSOMETHING,5

Operation

```
if variable-bit-shift then count=CL or count=Immed8;
else do;
  count=1;
  OF ← high-order-bit of operand.
end if;
do until count=0
  CF ← low-order-bit of operand;
  operand ← operand / 2;          /* UNSIGNED DIVIDE */
  count ← count-1;
end do;
```

SBB

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

The result of subtracting the right operand, then the original value of the carry flag, from the left operand replaces the left operand.

STC—Set Carry Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
F9	2	set the carry flag	STC

Operation

CF ← 1

Flags

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	1

Description

STC sets the carry flag, CF. No other flags are affected.

SHR

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	U	X	X

Description

SHR shifts the operand right by COUNT bits, shifting in high-order zero bits.

STI—Set Interrupt Enable Flag

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FB	2	set interrupt flag	STI

Operation

IF ← 1

Flags

O	D	I	T	S	Z	A	P	C
–	–	1	–	–	–	–	–	–

Description

STI sets the interrupt enable flag, IF. No other flags are affected.

STD—Set Direction Flags

Format

Opcode

Opcode	Clocks	Operation	Coding Example
FD	2	set direction flag	STD

Operation

DF ← 1

Flags

O	D	I	T	S	Z	A	P	C
–	1	–	–	–	–	–	–	–

Description

STD sets the direction flag, DF. No other flags are affected.

String

Flags

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

MOVS
MOVSB
MOVSW

Move the string pointed to by DS:SI into memory pointed to by ES:DI.

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

SCAS
SCASB
SCASW

Scan a string pointed to by ES:DI, comparing each element to AX or AL according to the type of string, and setting the flags to the result of such a comparison. Used with the conditional repeat-prefix (REPE....), this primitive can locate the next element matching AX/AL or next not-matching element.

Flags

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

LODS
LODSB
LODSW

Load each string element into AX/AL. This primitive would be used with the LOOP construct rather than the REP prefix, since some further processing on the data moved to AX/AL is almost surely necessary.

Flags

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

STOS
STOSB
STOSW

Store the AX or AL contents into the entire string.

The following operations are for iAPX 186:

INS — Store in memory pointed to by ES:DI the block of bytes or words read from the IO address in DX.

OUTS — Output to the IO address in DX the block of bytes/words in memory pointed to by DS:SI.

Each repetition of the string operation acknowledges pending interrupts, checks CX for zero (and stops repeating if 0), performs the string primitive operation once, adjusts any memory pointers used by the string operation by 1 for bytes and 2 for words (the adjustment is added if the FLAGS.DF is 0, otherwise subtracted), decrements CX (which does not affect the flags), and, in the case of SCAS and CMPS or their

String—String Operations

For 186 clocks, see Appendix H.

Format

Opcode

Opcode	Clocks	Operation	Coding Example
A6	22	flags ← (SI) - (DI)	CMPS BSTRING
A7	22	flags ← (SI) - (DI)	CMPS WSTRING
A4	18	(DI) ← (SI)	MOVS BSTRING1,BSTRING2
A5	18	(DI) ← (SI)	MOVS WSTRING1,WSTRING2
AE	15	flags ← (DI) - AX	SCAS BSTRING
AF	15	flags ← (DI) - AL	SCAS WSTRING
AC	12	AL ← (SI)	LODS BSTRING
AD	12	AX ← (SI)	LODS WSTRING
AA	11	(DI) ← AL	STOS BSTRING
AB	11	(DI) ← AX	STOS WSTRING
6E		(DI)←port(DX)	INS BSTRING, DX
6F		(DI)←port(DX:DX + 1)	INS WSTRING, DX
6C		port(DX)←(SI)	OUTS DX, BSTRING
6D		port(DX:DX + 1)←(SI)	OUTS DX, WSTRING

Operation

```
do until CX = 0;
  /* acknowledge any pending interrupts */
  perform string primitve once;
  CX ← CX - 1,          /* does not affect flags */
  if DF = 0 then add pointer adjustment to DS and/or ES
  else subtract pointer adjustment from DS and/or ES;
  if SCAS or CMPS, and repeat condition does not match ZF
  then undo,
end do;
```

Description

The string primitive operations are intended to be used primarily with the REP prefix. There are 7 primitives which, when so prefixed, perform the following operations:

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

CMPS
CMPSB
CMPSW — Compare the elements of two strings, one pointed to by ES:DI and the other by DS:SI.

SUB—Integer Subtraction

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
2A	3	Reg8 ← Reg8 - Reg8	SUB BL,CL
2A	9 + EA	Reg8 ← Reg8 - Mem8	SUB BL,BYTESOMETHING
2B	3	Reg16 ← Reg16 - Reg16	SUB BX,CX
2B	9 + EA	Reg16 ← Reg16 - Mem16	SUB BX,WORDSOMETHING
28	16 + EA	Mem8 ← Mem8 - Reg8	SUB BYTESOMETHING,BL
29	16 + EA	Mem16 ← Mem16 - Reg16	SUB WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
2C	4	AL ← AL - Immed8	SUB AL,5
2D	4	AX ← AX - Immed16	SUB AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 101)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 - Immed8	SUB BL,32
80	17 + EA	Mem8 ← Mem8 - Immed8	SUB BYTESOMETHING,32
81	4	Reg16 ← Reg16 - Immed16	SUB BX,1234H
81	17 + EA	Mem16 ← Mem16 - Immed16	SUB WORDSOMETHING,1234H
83	4	Reg16 ← Reg16 - Immed8	SUB BX,32
83	17 + EA	Mem16 ← Mem16 - Immed8 (Immed8 is sign-extended before subtract)	SUB WORDSOMETHING,32

Operation

LeftOpnd ← LeftOpnd - RightOpnd

Flags

O	D	I	T	S	Z	A	P	C
X	-	-	-	X	X	X	X	X

Description

The result of subtracting the right operand from the left operand replaces the left operand.

String

variants, checks the ZF for a match with the REP condition. As long as the REP condition matches, another repetition will be performed. For example, REPNE SCAS FOO will stop with ES:DI pointing to the next element of FOO which has not yet been scanned, and the last element scanned did not match the repeat condition 'Not Equal'- that is, the last element scanned matched the value in AX or AL, depending on whether FOO was a word or byte string. Repeat conditions 'NE' and 'NZ' match ZF=0, while 'E' and 'Z' match ZF=1.

Every string primitive has three variants. The mnemonics above, CMPS, MOVS, SCAS, LODS, INS, and OUTS, are generic and require one or more operands to be coded with them e.g. REP SCAS FOO or REP MOVS FEE,FIE These operands are used only to determine the size of a string element- byte or word- and do not determine the addresses of the strings used. The addresses used are determined solely by the contents of the register pairs ES:DI and DS:SI, as appropriate. Rather than coding operands for size specification, you may use the generic mnemonic with a 'W' or 'B' suffix- e.g. STOSB or CMPSW -and omit the operands entirely.

For Repeat String Operations, Clocks are:

Clocks	Coding Example	
9+17/rep	REP	MOVSB
9+22/rep	REPE	CMPSW
9+15/rep	REPNE	SCASB
9+13/rep	REP	LODSB
9+10/rep	REP	STOSW

WAIT—Wait While TEST pin not Asserted

Format

Opcode

Opcode	Clocks	Operation	Coding Example
9B	3 + 5n*	none	WAIT

*3 + 5n clocks where n is the number of times the TEST line is polled and found to be inactive.

Operation

None.

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The WAIT instruction causes the processor to enter a wait state if the signal on a TEST pin is not asserted. The wait state may be interrupted by an enabled external interrupt. When this occurs the saved code location is that of the WAIT instruction, so that upon return from the interrupting task the wait state is re-entered. The wait state is cleared and execution resumed when the TEST signal is asserted. Execution resumes without allowing external interrupts until after the execution of the next instruction. The instruction allows the processor to synchronize itself with external hardware.

TEST—Logical Compare

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
84	3	flags ← Reg8 AND Reg8	TEST BL,CL
84	9 + EA	flags ← Reg8 AND Mem8	TEST BL,BYTESOMETHING
85	3	flags ← Reg16 AND Reg16	TEST BX,CX
85	9 + EA	flags ← Reg16 AND Mem16	TEST BX,WORDSOMETHING

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
A8	4	flags ← AL AND Immed8	TEST AL,4
A9	4	flags ← AX AND Immed16	TEST AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 000)

Opcode	Clocks	Operation	Coding Example
F6	5	flags ← Reg8 AND Immed8	TEST BL,3FH
F6	11 + EA	flags ← Mem8 AND Immed8	TEST BYTESOMETHING,3FH
F7	5	flags ← Reg16 AND Immed16	TEST BX,3FFH
F7	11 + EA	flags ← Mem16 AND Immed16	TEST WORDSOMETHING,3FFH

Operation

flags ← LeftOpnd and RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The result of a bitwise logical AND of the two operands modifies the flags. Neither operand is modified.

XLAT/XLATB—Table Look-up Translation

Format

Opcode

Opcode	Clocks	Operation	Coding Example
D7	11	replace AL with table entry	XLAT ASCII_TABLE
D7	11		XLATB

Operation

AL ← table entry with effective address equal to BX + AL;

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

XLAT is intended for use as a table look-up instruction. You put the base address of the table in BX and a byte to be translated in AL. XLAT adds AL to the contents of BX and uses the result as an effective address. The byte at that EA is loaded into AL. BX is unchanged, and no flags are modified.

XCHG—Exchange Memory/Register With Register

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
86	4	Reg8 ←→ Reg8	XCHG BL,CL
86	17+EA	Mem8 ←→ Mem8	XCHG BYTESOMETHING,CL
87	4	Reg16 ←→ Reg16	XCHG BX,CX
87	17+EA	Mem16 ←→ Mem16	XCHG CX,WORDSOMETHING

Word Register with AX

Opcode + Reg

Opcode	Clocks	Operation	Coding Example
90 + Reg	3	AX ←→ Reg16	XCHG AX,BX

Operation

```
temp ← left operand;
left operand ← right operand;
right operand ← temp;
```

Flags

O	D	I	T	S	Z	A	P	C
–	–	–	–	–	–	–	–	–

Description

The two operands are exchanged. Segment registers are not legal operands. The order of the operands is immaterial. No flags are affected.

The 8087 Instruction Set

This section provides a summary discussion of those elements of the 8087 Numeric Processor that are of specific interest to the 8087 programmer. The following programmer accessible features of the architecture are included: floating-point stack; status, control and tag words; exception pointers; and data types. An elementary description of 8087 operation is provided to give a working understanding of 8086/8087/8088 coprocessing, 8087 numeric processing, exception handlers, and 8087 emulators.

Those users who wish detailed information on the 8087 architecture, operation, and/or those who wish to write their own exception handlers are referred to *The 8086 Family User's Manual, Numerics Supplement.*

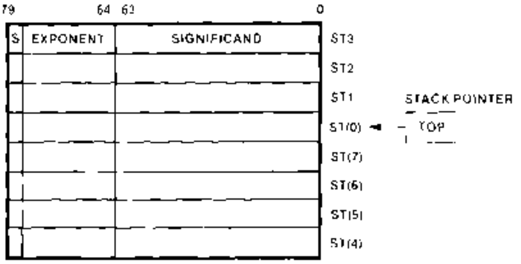
8087 Architectural Summary

The programmer accessible features of the 8087 Numeric Processor architecture consist of the eight floating-point stack elements; the seven words which constitute the 8087 environment (status word, control word, tag word, 2-word instruction address, and 2-word data address); and the seven data types accessible by the 8087.

Floating-Point Stack

The 8087 stack consists of eight elements divided into the fields shown in figure 6-1. The format of the fields corresponds with the temporary real data format used in all stack calculations and described under Data Types.

At a given point in time, the ST field in the status word identifies the current stack top element. This floating point stack element (rather than the status word field) is referred to in the rest of this chapter as ST. A load (push) operation, as in FLDLN2, decrements the stack pointer by 1 and loads a value (in this case $\log_e 2$) into the new stack top. An operation which pops the floating point stack increments the stack pointer by 1 (FADDP ST(i),ST adds the contents of the stack top to the stack element designated by (i), stores the result in ST(i) and increments the stack pointer by 1, making ST(1) the new stack top, ST(0).



121623-8

Figure 6-1. The 8087 Stack Fields

XOR—Logical Exclusive OR

Format

Memory/Reg with Reg

Opcode	ModRM		

Opcode	Clocks	Operation	Coding Example
32	3	Reg8 ← Reg8 XOR Reg8	XOR BL,CL
32	9 + EA	Reg8 ← Reg8 XOR Mem8	XOR BL,BYTESOMETHING
33	3	Reg16 ← Reg16 XOR Reg16	XOR BX,CX
33	9 + EA	Reg16 ← Reg16 XOR Mem16	XOR BX,WORDSOMETHING
30	16 + EA	Mem8 ← Mem8 XOR Reg8	XOR BYTESOMETHING,BL
31	16 + EA	Mem16 ← Mem16 XOR Reg16	XOR WORDSOMETHING,BX

Immed to AX/AL

Opcode	Data	

Opcode	Clocks	Operation	Coding Example
34	4	AL ← AL XOR Immed8	XOR AL,5
35	4	AX ← AX XOR Immed16	XOR AX,400H

Immed to Memory/Reg

Opcode	ModRM*			Data	

*—(Reg field = 110)

Opcode	Clocks	Operation	Coding Example
80	4	Reg8 ← Reg8 XOR Immed8	XOR BL,32
80	17 + EA	Mem8 ← Mem8 XOR Immed8	XOR BYTESOMETHING,32
81	4	Reg16 ← Reg16 XOR Immed16	XOR BX,1234H
81	17 + EA	Mem16 ← Mem16 XOR Immed16	XOR WORDSOMETHING,1234H

Operation

LeftOpnd ← LeftOpnd XOR RightOpnd
OF ← CF ← 0

Flags

O	D	I	T	S	Z	A	P	C
0	-	-	-	X	X	U	X	0

Description

The exclusive OR of two operands replaces the left operand. The carry and overflow flags are cleared.

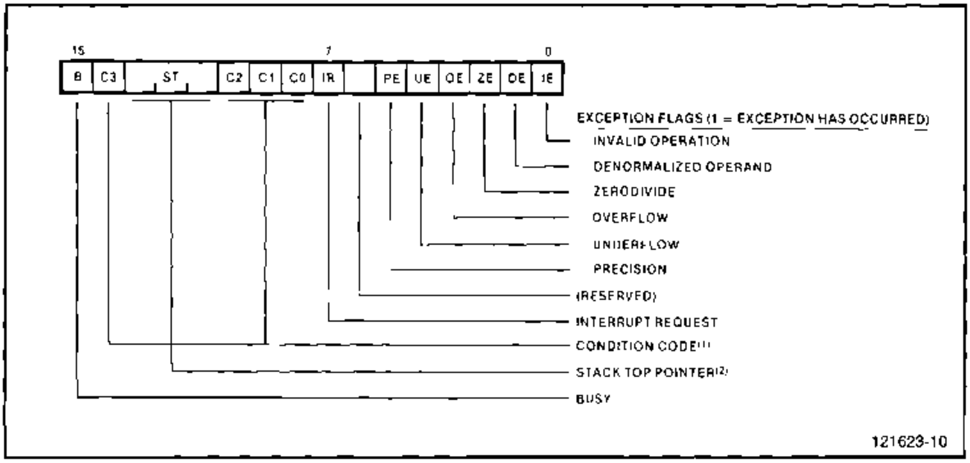


Figure 6-3. Status Word Format

ST values

000 = element 0 is stack top
001 = element 1 is stack top
.
.
.
111 = element 7 is stack top

Control Word

The control word consists of the exception masks, an interrupt enable mask, and control bits as shown in figure 6-4. During the execution of most instructions, the 8087 checks for six classes of exception conditions:

1. Invalid operations—programming errors such as trying to load a floating point stack element that is not empty, popping an operand from an element that is empty, using operands that cause indeterminate results (0/0, square root of a negative number, trying to store an unnormalized number which will not denormalize, etc.).
2. Overflow—usually the exponent of the true result is too large for the destination real format.
3. Underflow—the true exponent is too small to be represented in the result format.
4. Zerodivide—division of a finite non-zero operand by zero.
5. Denormalized—an instruction attempts to operate on a denormalized number.
6. Precision—for instructions that perform exact arithmetic, this exception means that some precision has been lost in reporting the results of an operation.

When one of these six conditions occurs, the corresponding flag in the status word is set to 1. The 8087 checks the appropriate mask in the Control Word to determine if it should process the exception with a default handling procedure on chip (mask = 1) or invoke a user written exception handler (mask = 0).

In the first case, the exception is said to be MASKED (from user software).

Elements of the floating point stack can be addressed either implicitly or explicitly:

FST ST(3)	Stores the contents of the stack top into element 3.
FADD	Adds the contents of the stack top to the contents of ST(1), stores the result in ST(1) and pops the stack. The result is now in the new stack top.

Note that floating-point stack indices outside of the range 0-7 are flagged as "out of range."

Environment

The 8087 environment consists of the seven words shown in figure 6-2.

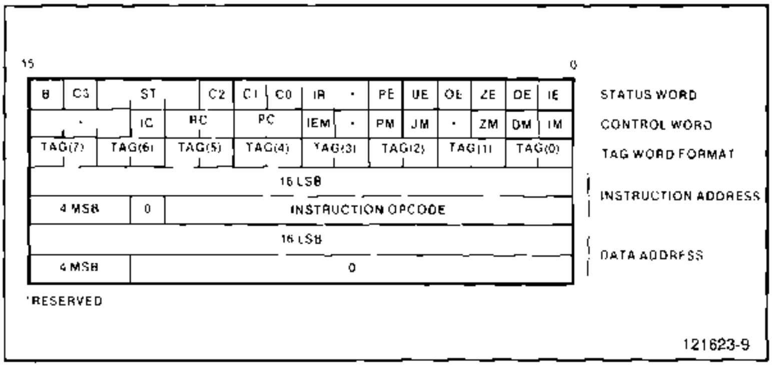


Figure 6-2. 8087 Environment

Status Word

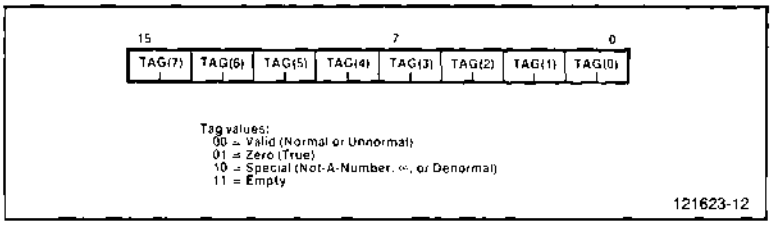
The status word reflects the overall condition of the 8087; it may be examined by storing it into memory with an 8087 instruction and then inspecting it with 8086/8088 CPU code. The status word is divided into the exception flag and status bit fields shown in figure 6-3. The busy field (bit 15) indicates whether the 8087 is executing an instruction (B=1) or is idle (B=0).

Several 8087 instructions (e.g., comparison instructions) result in modification of the condition code. The condition code is contained in bits 14 and 10-8 (C3-C0) of the status word. The condition code is used mainly for conditional branching. See the following instruction descriptions later in this chapter for condition code interpretations: FCOM, FCOMP, FCOMPP, FTST, FXAM and FPREM.

Bits 13-11 of the status word points to the 8087 stack element that is the current stack top (ST). Note that if ST=000B, a "push" operation which decrements ST, produces ST=111B; similarly, popping the stack with ST=111B yields ST=000B.

Bit 7 (IR) is the interrupt request field. The 8087 latches this bit to record a pending interrupt to the 8086/8088 CPU.

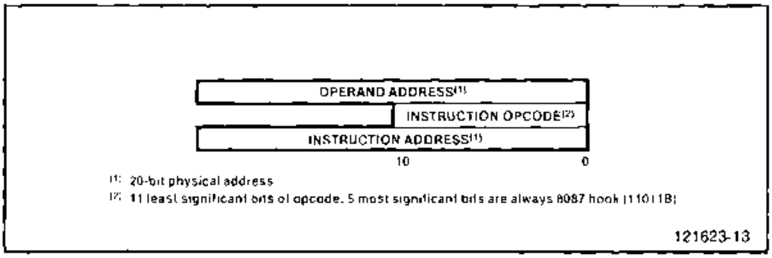
Bits 5-0 (PE, UE, OE, EE, DE, and IE) are set to indicate that the 8087 has detected an exception while executing an instruction.



Tag values:
00 = Valid (Normal or Unnormal)
01 = Zero (True)
10 = Special (Not-A-Number, ∞, or Denormal)
11 = Empty

121623-12

Figure 6-5. Tag Word Format



(1) 20-bit physical address
(2) 11 least significant bits of opcode. 5 most significant bits are always 8087 hook (11011B)

121623-13

Figure 6-6. Exception Pointers Format

Exception Pointers

The exception pointers shown in figure 6-6 are provided for user-written exception handlers. Whenever the 8087 executes an instruction, it saves the instruction address and the instruction opcode in the exception pointers. In addition, if the instruction references a memory operand, the address of the operand is retained also. An exception handler can be written to store these pointers in memory and obtain information concerning the instruction that caused the error.

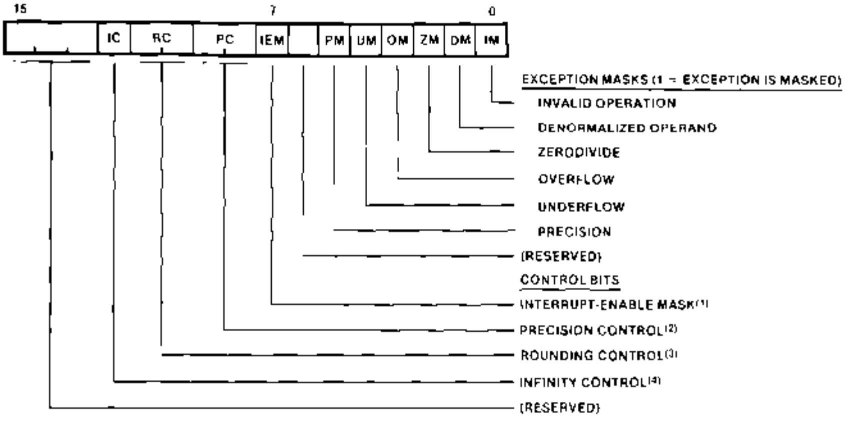
Data Types

The 8087 addresses seven different data types using all of the 8086 addressing modes. These data types and their valid ranges of value are shown in table 6-5.

Figure 6-7 describes how these formats are stored in memory (the sign is always located in the highest-addressed byte). In the figure, the most significant digits of all numbers (and field within numbers) are the leftmost digits.

Table 6-5. 8087 Data Types

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
WORD INTEGER	16	4-5	$-32768 \leq x \leq +32767$
SHORT INTEGER	32	9	$-2 \times 10^{9} \leq x \leq 2 \times 10^{9}$
LONG INTEGER	64	18	$-9 \times 10^{18} \leq x \leq +9 \times 10^{18}$
PACKED DECIMAL	80	18	$-99...99 \leq x \leq +99...99$ (18 digits)
SHORT REAL	32	6-7	$0, 1.2 \times 10^{-38} \leq \|x\| \leq 3.4 \times 10^{38}$
LONG REAL	64	15-16	$0, 2.3 \times 10^{-308} \leq \|x\| \leq 1.7 \times 10^{308}$
TEMPORARY REAL	80	19-20	$0, 3.4 \times 10^{-4932} \leq \|x\| \leq 1.1 \times 10^{4932}$



(1) Interrupt-Enable Mask:
0 = Interrupts Enabled
1 = Interrupts Disabled (Masked)

(2) Precision Control:
00 = 24 bits
01 = (reserved)
10 = 53 bits
11 = 64 bits

(3) Rounding Control
00 = Round to Nearest or Even
01 = Round Down (toward −∞)
10 = Round Up (toward +∞)
11 = Chop (Truncate Toward Zero)

(4) Infinity Control
0 = Projective
1 = Affine

121623-11

Figure 6-4. Control Word Format

The control bits have the following meanings:

PC:	Precision control—results are rounded to one of three precisions: Temporary Real (64 bits), Long Real (53 bits) or Short Real (24 bits).
RC:	Rounding Control—results are rounded in one of four directions: unbiased round to the nearest or even value, round toward +, round toward −, or round toward zero.
IC:	Infinity Control—there are two types of infinity arithmetic provided: affine and projective. The default means of closing a Number system is projective. See *The 8086 Family User's Manual, Numerics Supplement,* for a complete description.

Tag Word

The tag word, as shown in figure 6-5, contains tags describing the contents of the corresponding stack elements.

The short and long real formats exist only in memory. If a number in one of these formats is loaded into the stack, it is automatically converted to temporary real.

Special values are included to increase flexibility though not within the domain of normal floating point arithmetic. These special values are listed here, but the reader is referred to *The 8086 Family User's Manual, Numerics Supplement,* for descriptions. The special values include:

- Signed zero
- +∞ and −∞ representations
- Indefinite values
- NAN values (Not-A-Number)
- Denormals
- Unnormals

8087 Operation

Coprocessing

The 8087 and host CPU act as coprocessors. They share the same instruction stream and sometimes perform parallel executions. The 8086/8088 has a set of ESCAPE instructions that, in memory addressing mode, cause the 8086/8088 to calculate the address and read the contents of that address. The 8086/8088 ignores the word it reads and executes subsequent instructions. The 8087, however, monitors the same instruction stream and when it detects an ESCAPE it begins processing. The 8087 latches the opcode and, if there was an address calculated, the 8087 captures both the address and the datum read by the 8086/8088. The 8087 decodes the instruction to determine how many more words it needs from memory. It increments the address and fetches data until all required data is read. The 8087 then releases the bus and begins calculating while the 8086/8088 continues executing the instruction stream.

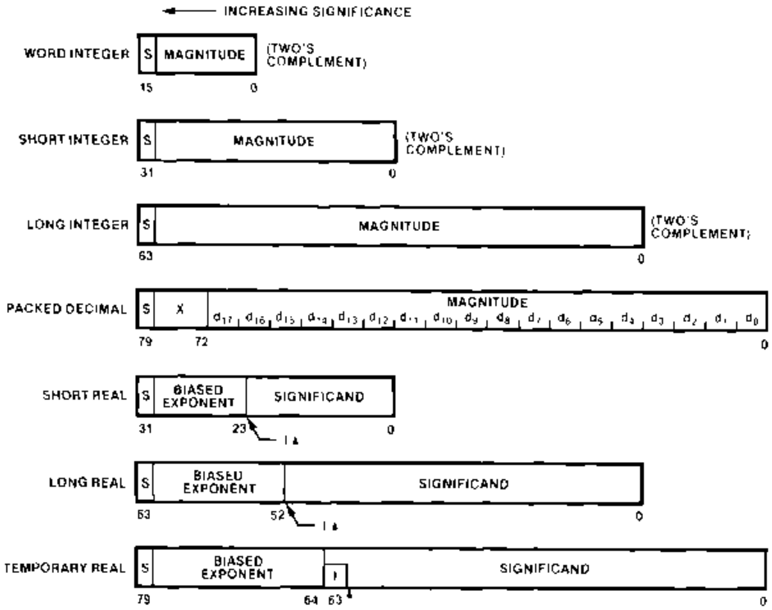
The 8086/8088 WAIT instruction allows software to synchronize the 8086/8088 to the 8087 so that the host processor does not execute the next instruction until the 8087 is finished with its current (if any) instruction. To accomplish this, the programmer should explicitly code the FWAIT instruction immediately before an 8086/8088 instruction that accesses a memory operand read or written by a previous 8087 instruction.

If an 8087 and a processor other than its host CPU can both update a variable, access to that variable should be controlled so that one processor at a time has exclusive rights to it. This can be done by using an 8086/8088 XCHG instruction prefixed by LOCK. When the 8087 no longer needs the variable, the 8086/8088 clears it and again makes it available for use.

The 8087 interrupt requests are made to the 8086/8088 as the result of detecting an exception. Interrupts are enabled or disabled by the Interrupt Enable Mask (IEM) in the Control Word. When IEM is set to 1, interrupts are masked (disabled). The interrupt request remains set until it is explicitly cleared. This can be done by the FNCLEX, FNSAVE, or FINIT instructions.

Numeric Processing

The 8087 has four rounding modes, selectable by the RC field in the control word. The rounding modes and their corresponding RC fields are shown in table 6-6.



NOTES
S = Sign bit (0 = positive, 1 = negative)
d_n = Decimal digit (two per byte)
X = Bits have no significance; 8087 ignores when loading, zeros when storing
▲ = Position of implicit binary point
I = Integer bit of significand; stored in temporary real, implicit in short and long real
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

121623-14

Figure 6-7. Data Formats

The three binary integer formats are identical except for length, which governs the range that can be accommodated in each format. The leftmost bit is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard two's complement notation (the binary integers are the only 8087 format to use two's complement). The quantity zero is represented with a positive sign (all bits 0). The 8087 word integer format is identical to the 16-bit signed integer data type of the 8086 and 8088.

Decimal integers are stored in packed decimal notation, with two decimal digits "packed" into each byte. Negative numbers are distinguished from positive ones only by the sign bit. All digits must be in the range 0H-9H.

The 8087 stores real numbers in a three-field binary format that resembles scientific notation. The number's significant digits are held in the SIGNIFICAND field, the EXPONENT field locates the binary point within the significant digits (determining the number's magnitude), and the SIGN field indicates whether the number is positive or negative. Negative numbers differ from positive numbers only in their sign bit.

Table 6-7. Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN**, return it; if both are NANS, return NAN with larger absolute value; if neither is NAN, return *indefinite*.	Request interrupt.
Zerodivide	Return ∞ signed with "exclusive or" of operand signs.	Request interrupt.
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then re-evaluate for exceptions.	Request interrupt.
Overflow	Return properly signed ∞	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent,* store result, request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt

* On overflow, 24,576 decimal is *subtracted* from the true result's exponent; this forces the exponent back into range and permits a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is *added* to the true result's exponent.

** NAN is a member of a class of special values that exist in the real formats only. See the *The 8086 Family User's Manual, Numerics Supplement*.

8087 Emulators

Numeric processing capability is not restricted to 8087 users. Intel offers two 8086/8088 software products which provide 8087 functionality. E8087 emulates the full 8087 instruction set for assembly language programs. PE8087 furnishes numeric support for PL/M-86 software. Use of the 8087 Emulators necessitates modification of the instruction formats presented in this chapter.

ASM86, the Intel 8086/8087/8088 assembler, produces special object code for 8087 instructions. Floating point instructions are identified in such a way that they may be linked to the 8087 Emulators. Refer to your operating system's 8086/8087/8088 assembler operating instructions manual for a short description of this change and link procedure.

Organization of the 8087 Instruction Set

Data Transfer Instructions

These instructions are summarized in table 6-8. They move operands among stack elements or between the stack top and memory. Any of the seven data types can be converted to temporary real and loaded (pushed) onto the stack in a single operation; they can be stored in memory in the same manner. The data transfer instructions automatically update the 8087 tag word to reflect the stack contents following the instruction.

Table 6-6. Rounding Modes

RC Field	Rounding Mode	Rounding Action
00	Round to nearest	Closer to *b* of *a* or *c*; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward −∞)	*a*
10	Round up (toward +∞)	*c*
11	Chop (toward 0)	Smaller in magnitude of *a* or *c*

Note: $a < b < c$; *a* and *c* are representable, *b* is not.

Rounding occurs in arithmetic and store operations when the format of the destination cannot exactly represent the true result. This can happen when a precise temporary real number is stored in a shorter real format or in an integer format. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded. "Round to the nearest significant bit" is the default mode and is suitable for most applications. Other modes and applications are described in *The 8086 Family User's Manual, Numerics Supplement.*

The precision of results can be calculated to 64, 53, or 24 bits as selected by the PC field of the control word. The default setting is 64 bits. This setting is best suited for most applications.

The 8087's system of real numbers may be closed by either of two models of infinity. The IC field in the control word is set for either projective or affine closure. The default is projective, which is recommended for most computations. Both closure forms and their uses are described in *The 8086 Family User's Manual, Numerics Supplement.*

The 8087 can represent data and final results of calculations in the range $\pm 2.3 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$ (double precision). Compared to most computers, including large mainframes, the 8087 provides a very good approximation of the real number system. It is important to remember, however, that it is not an exact representation, and that arithmetic on real numbers is inherently approximate.

Conversely, and equally important, the 8087 does perform exact arithmetic on its integer subset of the reals. That is, an operation on two integers returns an exact integral result, provided that the true result is an integer and is in range.

The 8087 detects the six types of exceptions shown in table 6-7. The programmer has a choice of using the 8087 on-chip fault-handling capability by masking exceptions in the Control Word, or writing software exception handlers and unmasking exceptions in the control word. Table 6-3 shows the 8087 response to each situation.

If the exception is unmasked, its detection results in the generation of an interrupt. When an interrupt is generated, the interrupt procedure (exception handler) has available the exception flags, a pointer to the instruction causing the interrupt and a pointer to the datum if memory was addressed. Each of the exceptions shown in table 6-7 has a sticky flag associated with it, which means that once the flag is set, it remains until reset by software. Several instructions can be used to clear the flag: FCLEX clears exceptions; FRSTOR or FLDENV overwrite flags.

Those users who wish to write their own exception handlers should consult *The 8086 Family User's Manual, Numerics Supplement* since they will vary widely from one application to the next.

The stack element form is a generalization of the classical stack form; the programmer specifies the stack top as one operand and any stack element on the stack as the other operand. Coding the stack top as the destination provides a convenient way to make use of a constant held elsewhere in the stack. The converse coding (ST is the source operand) allows, for example, adding the top into a stack element used as an accumulator.

Often the operand in the stack top is needed for one operation but then is of no further use in the computation. The stack element and pop form can be used to pick up the stack top as the source operand, and then discard it by popping the floating point stack. Coding operands of ST(1),ST with a stack element pop mnemonic is equivalent to a classical stack operation: the top is popped and the result is left at the new top.

Programmers no longer need to spend valuable time eliminating square roots from algorithms because processors run too slowly. Other arithmetic instructions perform exact modulo division, round real numbers to integers, and scale values by powers of two.

The 8087's arithmetic instructions (addition, subtraction, multiplication, and division) allow the programmer to minimize memory references and to make optimum use of the 8087 floating-point stack.

Table 6-10 summarizes the available operation/operand forms that are provided for basic arithmetic. In addition to the four normal operations, two "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication.

- Operands may be located in stack elements or memory.
- Results may be deposited in a choice of stack elements.
- Operands may be a variety of 8087 data types: long real, short real, short integer or word integer, with automatic conversion to temporary real performed by the 8087.

Five instruction forms may be used across all six operations, as shown in table 6-10. The classical stack form may be used to make the 8087 operate like a classical stack machine. No operands are coded in this form, only the instruction mnemonic is coded. The 8087 picks the source operand from the stack top and the destination from the next stack element. It then performs the operation, pops the stack, and returns the result to the new stack top, effectively replacing the operands by the result.

Table 6-10. Basic Arithmetic Instructions and Operands

Instruction Form	Mnemonic Form	Operand Forms destination, source	ASM86 Example	
Classical stack	F*op*	{ST(1),ST}	FADD	
Stack element	F*op*	ST(i),ST or ST,ST(i)	FSUB	ST,ST(3)
Stack element and pop	F*op*P	ST(i),ST	FMULP	ST(2),ST
Real memory	F*op*	{ST,} short-real/long-real	FDIV	AZIMUTH
Integer memory	FI*op*	{ST,} word-integer/short-integer	FIDIV	N__PULSES

Notes: Braces { } surround *implicit* operands; these are not coded, and are shown here for information only.

op = ADD destination ← destination + source
SUB destination ← destination − source
SUBR destination ← source − destination
MUL destination ← destination · source
DIV destination ← destination ÷ source
DIVR destination ← source ÷ destination

Table 6-8. Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

Arithmetic Instructions

The arithmetic instruction set for the 8087 provides a great many variations on the basic add, subtract, multiply and divide operations, and a number of other useful functions. Table 6-9 gives a summary of these instructions.

Table 6-9. Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

The transcendental instructions assume that their operands are *valid* and *in-range*. The instruction descriptions in this section provide the range of each operation. To be considered valid, an operand to a transcendental must be normalized; denormals, unnormals, infinities and NANs are considered invalid. Zero operands are accepted by some functions and are considered out-of-range by others. If a transcendental operand is invalid or out-of-range, the instruction will produce an undefined result without signaling an exception. It is the programmer's responsibility to ensure that operands are valid and in-range before executing a transcendental. FPREM may be used to bring an operand into range for periodic functions.

Constant Instructions

Each of these instructions (table 6-13) loads (pushes) a commonly-used constant onto the stack. The values have full temporary real precision (64 bits) and are accurate to approximately 19 decimal digits. Since a temporary real constant occupies 10 memory bytes, the constant instructions, which are only two bytes long, save storage and improve execution speed, in addition to simplifying programming.

Table 6-13. Constant Instructions

FLDZ	Load + 0.0
FLD1	Load + 1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Processor Control Instructions

When CPU interrupts are enabled, as will normally be the case when an application task is running, the "wait" forms of these instructions should be used. Most of the instructions shown in table 6-14 are used in system-level activities rather than in computations. These activities include: initialization, exception handling, and task switching.

Alternate mnemonics are shown for several of the processor control instructions in table 6-14. This mnemonic, distinguished by a second character of "N", instructs the assembler *not* to prefix the instruction with a CPU WAIT instruction (instead, a CPU NOP precedes the instruction). This "no-wait" form is intended for use in critical code regions where a WAIT instruction might precipitate an endless wait. Thus, when CPU interrupts are disabled, and the 8087 can potentially generate an interrupt, the "no-wait" form should be used.

Except for FNSTENV and FNSAVE, all instructions which provide a no-wait mnemonic are self-synchronizing and can be executed back-to-back in any combination without intervening FWAITs. These instructions can be executed by one part of the 8087 while the other part is busy with a previously decoded instruction. To ensure that the processor control instruction executes after completion of any operation in progress, the "WAIT" form of that instruction should be used.

The two memory forms increase the flexibility of the 8087's arithmetic instructions. They permit a real number or a binary integer in memory to be used directly as a source operand. This is a very useful facility in situations where operands are not used frequently enough to justify holding them in the floating point stack. Note that various forms of data allocation may be used to define these operands; they may be elements in arrays, structures or other data organizations, as well as simple scalars.

The six functional groups of instructions are discussed further in the next paragraphs.

Comparison Instructions

Each of these instructions (table 6-11) analyzes the top stack element, often in relationship to another operand, and reports the result in the status word condition code. The basic operations are compare, test (compare with zero), and examine (report tag, sign, and normalization). Special forms of the compare operation are provided to optimize algorithms by allowing direct comparisons with binary integers and real numbers in memory, as well as popping the stack after a comparison.

The FSTSW (store status word) instruction may be used following a comparison to transfer the condition code to memory for inspection. See individual descriptions of the instructions listed in table 6-11 for interpretations of the condition code bits.

Note that instructions other than those in the comparison group may update the condition code. To ensure that the status word is not altered inadvertently, it should be stored immediately after the compare operation.

Table 6-11. Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Transcendental Instructions

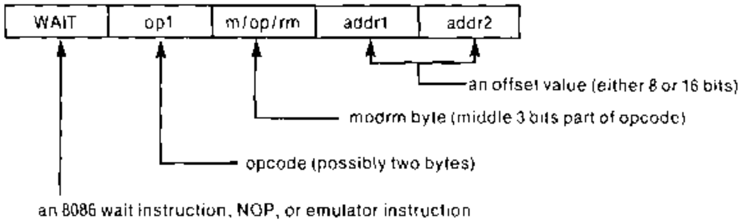
The instructions in this group are summarized in table 6-12. They perform the *core calculations* for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Prologue and epilogue software may be used to reduce arguments to the range accepted by the instructions and to adjust the result to correspond to the original arguments if necessary. The transcendentals operate on the top one or two stack elements, and they return their results to the stack.

Table 6-12. Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^X-1
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X+1)$

MNEMONIC—Sample 8086/8088 Instruction

Format


WAIT
op1
m/op/rm
addr1
addr2
an offset value (either 8 or 16 bits)
modrm byte (middle 3 bits part of opcode)
opcode (possibly two bytes)
an 8086 wait instruction, NOP, or emulator instruction

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
(the 8087 instruction coding)	(emulator instruction coding)	typical range	(machine operation)	MNEMONIC

Operation

(A description of the machine operation.)

Exceptions

I Z D O U P

(shows which exceptions could be set)

Table 6-14. Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

FABS—Absolute Value

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E1	CD 19 E1	14 10-17	ST ← |ST|	FABS

Operation

The absolute value instruction changes the element in the top of the stack to its absolute value by making its sign positive.

Exceptions

I	Z	D	O	U	P
X					

F2XM1—$2^x - 1$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F0	CD 19 F0	500 310-630	$ST \leftarrow 2^{ST} - 1$	F2XM1

Operation

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

Exceptions

I	Z	D	O	U	P	*
				X	X	

*Operands not checked.

Description

This instruction is designed to produce a very accurate result even when x is close to zero. To obtain $Y = 2^X$, add 1 to the result delivered by F2XM1.

The following formulas show how values other than 2 may be raised to a power of X.

$$10^x = 2^{x * \log_2 10}$$

$$e^x = 2^{x * \log_2 e}$$

$$Y^x - 2^{x * \log_2 y}$$

The 8087 has built-in instructions, described in this chapter, for loading the constants LOG_2 10 and LOG_2 e, and the FYL2X instruction may be used to calculate X * $\log_2 Y$.

FADDP/FADD—Add Real and Pop

Format

Stack top + Stack Element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE C1	CD 1E C1	90 75-105	ST(1) ← ST + ST(1) pop stack	FADD
9B DE C0 + i	CD 1E C0 + i	90 75-105	ST(i) ← ST + ST(i) pop stack	FADDP ST(2),ST

Operation

The add real and pop stack instruction adds the stack top to one of the stack elements, replacing the stack element with the sum, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FADD—Add Real

Format

Stack top + Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 C0 + i	CD 18 C0 + i	85 70-100	ST ← ST + ST(i)	FADD ST,ST(2)
9B DC C0 + i	CD 1C C0 + i	85 70-100	ST(i) ← ST + ST(i)	FADD ST(4),ST

Stack top + memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m0rm	CD 18 m0rm	105 + EA (90-120) + EA	ST ← ST + mem-op (short-real)	FADD COUNT
9B DC m0rm	CD 1C m0rm	110 + EA (95-125) + EA	ST ← ST + mem-op (long-real)	FADD MEAN

Operation

The add real instruction adds the source operand to the destination operand and places the result in the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FBSTP—Packed Decimal (BCD) Store and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DF m6rm	CD IF m6rm	530 + EA (520-540) + EA	mem-op ← ST pop stack	FBSTP FORECAST

Operation

The packed decimal store and pop stack instruction converts the contents of the stack top to a packed decimal integer, stores the result at the destination in memory, and pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X					

Note

FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then deleting least significant bits.

Users who are concerned about rounding may precede FBSTP with FRNDINT.

FBLD—Packed Decimal (BCD) Load

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DF m4rm	CD IF m4rm	300 + EA (290-310) + EA	push stack ST ← mem-op	FBLD YTD_SALES

Operation

The BCD load instruction converts the memory operand from packed decimal to temporary real and pushes the result onto the stack. The sign of source is preserved, including the case when the value is negative zero.

Exceptions

I	Z	D	O	U	P
X					

Note

The packed decimal digits of the source are assumed to be in the range 0-9H. The instruction does not check for invalid digits (A-FH) and the result of attempting to load an invalid encoding is undefined.

FCLEX/FNCLEX—Clear Exceptions

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E2	CD 1B E2	5 2-8	clear 8087 exceptions	FCLEX
90 DB E2	CD 1B E2	5 2-8	clear 8087 exceptions (no wait)	FNCLEX

Operation

This instruction clears all exception flags, the interrupt request flag and the busy flag in the status word. As a consequence, the 8087's INT and BUSY lines go inactive. The FCLEX form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

FNCLEX is used in critical areas of code where a WAIT instruction might result in a deadlock. FCLEX is used to insure that the processor control instruction executes only after completion of any operation in progress in the NOP.

FCHS—Change Sign

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E0	CD 19 E0	15 10-17	ST ← −ST	FCHS

Operation

The change sign instruction complements the sign on the stack top element.

Exceptions

I	Z	D	O	U	P
X					

FCOM

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87               ;STORE RESULT FROM FCOM
FWAIT                       ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1  ;MOVE STATUS BYTE TO AH
SAHF                        ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FCOM—Compare Real

Format

Compare Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 D1	CD 18 D1	45 40-50	ST ST(1)	FCOM
9B D8 D0 + i	CD 18 D0 + i	45 40-50	ST ST(i)	FCOM ST(2)

Compare Stack top and memory operands

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m2rm	CD 18 m2rm	65 + EA (60-70) + EA	ST - memop (short-real)	FCOM WAVELENGTH
9B DC m2rm	CD 1C m2rm	70 + EA (65-75) + EA	ST - memop (long real)	FCOM MEAN

Operation

The compare real instruction compares the stack top with the source operand. The source operand may be a stack element or short or long real memory operand. If no operand is coded, ST is compared with ST(1).

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Format

Compare Stack top and Stack element and pop

WAIT	op1	op2 + i

FCOMP

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                ;STORE RESULT FROM FCOM
FWAIT                        ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1   ;MOVE STATUS BYTE TO AH
SAHF                         ;LOAD INTO 8086 FLAGS REGISTER
```

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FCOMPP—Compare Real and Pop Twice

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE D9	CD 1E D9	50	ST - ST(1)	FCOMPP
		45-55	pop stack	
			pop stack	

Operation

The compare real and pop stack twice instruction compares the stack top with ST(1) and pops the floating point stack twice, discarding both operands. No operands may be explicitly coded with this instruction.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                 ;STORE RESULT FROM FCOM
FWAIT                         ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1    ;MOVE STATUS BYTE TO AH
SAHF                          ;LOAD INTO 8086 FLAGS REGISTER
```

FCOMPP

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ⩽ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ⩾ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FDECSTP—Decrement Stack Pointer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F6	CD 19 F6	9 6-12	stack pointer ← 2 stack pointer 1	FDECSTP

Operation

This instruction subtracts 1 from the stack top pointer in the status word. No tags or registers are altered, nor is any data transferred. Executing FDECSTP when the stack top pointer is 0, changes the pointer to 7.

Exceptions

I Z D O U P

FDISI/FNDISI—Disable Interrupts

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E1	CD 1B E1	5 2 B	Set 8087 interrupt mask	FDISI
90 DB E1	CD 1B E1	5 2-8	Set 8087 interrupt mask (no wait)	FNDISI

Operation

The instruction sets the interrupt enable mask in the control word and prevents the NDP from issuing an interrupt request. The FDISI form of this instruction is preceded by an assembler-generated WAIT.

Exceptions

I Z D O U P

Description

The NO WAIT form of the instruction (FNDISI) is intended for use in critical code regions where a WAIT instruction might induce an endless wait.

Note

If WAIT is decoded with pending exceptions, the 8087 generates an interrupt— masked or not.

FDIV—Divide Real

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 F0+i	CD 18 F0+i	198 193-203	ST ← ST/ST(i)	FDIV ST,ST(2)
9B DC F8+i	CD 1C F8+i	198 193-203	ST(i) ← ST(i)/ST	FDIV ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m6rm	CD 18 m6rm	220+EA (215-225)+EA	ST ← ST/mem-op (short-real)	FDIV DISTANCE
9B DC m6rm	CD 1C m6rm	225+EA (220-230)+EA	ST ← ST/mem-op (long-real)	FDIV GAMMA

Operation

The divide real instructions divide the destination by the source and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The divide real and pop stack instruction divides one of the stack elements by the stack top, replaces the stack element with the quotient, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FDIVP—Divide Real and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE F9	CD 1E F9	202 197-207	ST(1) ← ST(1)/ST pop stack	FDIV
9B DE F8 + i	CD 1E F8 + i	202 197-207	ST(i) ← ST(i)/ST pop stack	FDIVP ST(3),ST

Operation

The divide real instructions divide the destination by the source and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The divide real and pop stack instruction divides one of the stack elements by the stack top, replaces the stack element with the quotient, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FDIVR—Divide Real Reversed

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 F8+i	CD 18 F8+i	199 194-204	ST ← ST(i)/ST	FDIVR ST,ST(2)
9B DC F0 + i	CD 1C F0 + i	199 194-204	ST(i) ← ST/ST(i)	FDIVR ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m7rm	CD 18 m7rm	221 + EA (216-226) + EA	ST ← mem-op/ST (short-real)	FDIVR RATE
9B DC m7rm	CD 1C m7rm	226 + EA (221-231) + EA	ST ← mem-op/ST (long-real)	FDIVR SPEED

Operation

The divide real reversed instructions divide the source operand by the destination and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse divide and pop stack instruction divides the stack top by one of the stack elements and returns the quotient to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FDIVRP–Divide Real Reversed and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE F1	CD 1E F1	203 198-208	ST(1) ← ST/ST(1) pop stack	FDIVR
9B DE F0 + i	CD 1E F0 + i	203 198-208	ST(i) ← ST/ST(i)	FDIVRP ST(4),ST

Operation

The divide real reversed instructions divide the source operand by the destination and return the quotient to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse divide and pop stack instruction divides the stack top by one of the stack elements and returns the quotient to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FENI/FNENI—Enable Interrupts

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E0	CD 1B E0	5 2-8	clear 8087 interrupt mask	FENI
90 DB E0	CD 1B E0	5 2-8	clear 8087 interrupt mask (no wait)	FNENI

Operation

This instruction clears the interrupt enable mask in the control word, allowing the 8087 to generate interrupt requests. The FENI form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

The NO WAIT form of the instruction (FNENI), is intended for use in critical code regions where a WAIT instruction might induce an endless wait.

The WAIT form of this instruction (FENI), should be used in all non-critical code regions. This form insures that the processor control instruction executes after completion of any operation in progress in the NEU.

FFREE—Free Register

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD C0 + i	CD 1D C0 + i	11 9-16	TAG(i) masked empty	FFREE ST(1)

Operation

This instruction changes the destination stack element's tag to empty. The contents of this stack element are unaffected.

Exceptions

I Z D O U P

FIADD—Integer Add

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m0rm	CD 1A m0rm	125 + EA (108-143) + EA	ST ← ST + mem-op (short integer)	FIADD DISTANCE
9B DE m0rm	CD 1E m0rm	120 + EA (102-137) + EA	ST ← ST + mem-op (word integer)	FIADD PULSE

Operation

This instruction adds the integer memory source to the top of the stack and returns the sum to the destination at the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

FICOM—Integer Compare

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m2rm	CD 1A m2rm	85 + EA (78-91) + EA	ST – mem-op (short integer)	FICOM PASSES
9B DE m2rm	CD 1E m2rm	80 + EA (72-86) · EA	ST - mem-op (word integer)	FICOM CENTS

Operation

The integer compare instructions convert the memory operand (a word or short binary integer) to temporary real and compare it with the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST = source
1	1	1	ST ? source

Note

> NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87               ;STORE RESULT FROM FICOM
FWAIT                       ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1  ;MOVE STATUS BYTE TO AH
SAHF                        ;LOAD INTO 8086 FLAGS REGISTER
```

FICOM

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FICOMP—Integer Compare and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m3rm	CD 1A m3rm	87 - EA (80-93) + EA	ST – mem-op pop stack (short integer)	FICOMP LIMIT
9B DE m3rm	CD 1E m3rm	82 - EA (74-88) + EA	ST – mem-op pop stack (word integer)	FICOMP SAMPLE

Operation

The integer compare instructions convert the memory operand (a word or short binary integer) to temporary real and compare it with the top of the stack. FICOMP additionally discards the value in ST by popping the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Following the instruction, the condition codes in the 8087 status byte reflect the order of the operands as follows:

C3	C2	C0	ORDER
0	0	0	ST > source
0	0	1	ST < source
1	0	0	ST – source
1	1	1	ST ? source

Note

> NANs and ∞ (projective) cannot be compared and return C3 = C0 = 1 as shown above.

The following procedures can be used to store the status word from this instruction and test the compare result.

The condition code can be transferred from the 8087 status byte to memory, an 8086 register, or the 8086 flags register. For example, the code required to transfer the information to the flags register is:

```
FSTSW STAT_87                ;STORE RESULT FROM FICOMP
FWAIT                        ;WAIT FOR STORE
MOV AH, BYTE PTR STAT_87+1   ;MOVE STATUS BYTE TO AH
SAHF                         ;LOAD INTO 8086 FLAGS REGISTER
```

FICOMP

The 8086 instructions are now used to execute a conditional branch on the result of the compare as follows:

```
JB  - ;JUMP if ST < source OR ST ? source
JBE - ;JUMP IF ST ≤ source OR ST ? source
JA  - ;JUMP IF ST > source and NOT ST ? source
JAE - ;JUMP IF ST ≥ source and NOT ST ? source
JE  - ;JUMP IF ST = source or ST ? source
JNE - ;JUMP IF ST ≠ source and NOT ST ? source
```

FIDIV–Integer Divide

Format

WAIT	op1	*m/op/rm*	*addr1*	*addr2*

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m6rm	CD 1A m6rm	236 + EA (230-243) + EA	ST ← ST/mem-op (short integer)	FIDIV SURVEY
9B DE m6rm	CD 1E m6rm	230 + EA (224-238) + EA	ST ← ST/mem-op (word integer)	FIDIV ANGLE

Operation

The integer divide instruction divides the top of the stack by the integer memory operand and returns the quotient to the top of the stack.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FIDIVR–Integer Divide Reversed

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m7rm	CD 1A m7rm	237 - EA (231-245) - EA	ST ← mem-op/ST (short integer)	FIDIVR COORD
9B DE m7rm	CD 1E m7rm	230 + EA (225-239) + EA	ST ← mem-op/ST (word integer)	FIDIVR FREQUENCY

Operation

The reversed integer divide instruction divides the integer memory operand by the top of the stack and returns the quotient to the stack top.

Exceptions

I	Z	D	O	U	P
X	X	X	X	X	X

FILD—Integer Load

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m0rm	CD 1B m0rm	56 + EA (52-60) + EA	push stack ST ← mem-op (short integer)	FILD STANDOFF
9B DF m0rm	CD 1F m0rm	50 + EA (46-54) + EA	push stack ST ← mem-op (word integer)	FILD SEQUENCE
9B DF m5rm	CD 1F m5rm	64 + EA (60-68) + EA	push stack ST ← mem-op (long integer)	FILO RESPONSE

Operation

The integer load instruction converts the integer memory operand from its binary integer format (word, short, or long) to temporary real and pushes the result onto the stack. The new stack top is tagged zero if all bits in the source were zero, and is tagged valid otherwise.

Exceptions

I	Z	D	O	U	P
X					

FIMUL—Integer Multiply

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m1rm	CD 1A m1rm	136 + EA (130-144) + EA	ST ← ST * mem-op (short integer)	FIMUL BEARING
9B DE m1rm	CD 1E m1rm	130 + EA (124 138) + EA	ST ← ST * mem-op (word integer)	FIMUL POSITION

Operation

The integer multiply instruction multiplies the integer memory operand and the top of the stack and returns the product to the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

FINCSTP—Increment Stack Pointer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F7	CD 19 F7	9	stack pointer ←	FINCSTP
		6-12	stack pointer + 1	

Operation

The stack pointer increment instruction adds 1 to the stack top pointer in the status word. It does not alter tags or register contents, nor does it transfer data. It is not equivalent to popping the stack since it does not set the tag of the previous stack to empty. Incrementing a stack pointer of 7 changes it to 0.

Exceptions

I Z D O U P

FINIT/FNINIT—Initialize Processor

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB E3	CD 1B E3	5 2-8	initialize 8087	FINIT
90 DB E3	CD 1B E3	5 2-8	initialize 8087 (no wait)	FNINIT

Operation

The initialize processor instruction performs the functional equivalent of a hardware RESET, except that it does not affect the instruction fetch synchronization of the 8087 and its CPU. FINIT/FNINIT sets the control word to 03FFH, empties all floating point stack elements, and clears exception flags and busy interrupts. The FINIT form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Note

The system should call the INIT87 procedure in lieu of executing FINIT/FNINIT when the processor is first initialized, for compatability with the 8087 emulator.

FIST—Integer Store

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m2rm	CD 1B m2rm	88+EA (82-92)+EA	mem-op ← ST (short integer)	FIST COUNT
9B DF m2rm	CD 1F m2rm	86+EA (80-90)+EA	mem-op ← ST (word integer)	FIST FACTOR

Operation

The integer store instruction rounds the contents of the stack top to an integer (according to the RC field of the control word) and transfers the result to the memory destination. The destination may define a word or short integer variable. Negative zero is stored in the same encoding as positive zero: 0000...00.

Exceptions

I	Z	D	O	U	P
X					X

FISTP—Integer Store and Pop

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DB m3rm	CD 1B m3rm	90 + EA (84-94) + EA	mem-op ← ST pop stack (short integer)	FISTP CORRECTED
9B DF m3rm	CD 1F m3rm	88 + EA (82-92) + EA	mem-op ← ST pop stack (word integer)	FISTP ALPHA
9B DF m7rm	CD 1F m7rm	100 + EA (94-105) + EA	mem-op ← ST pop stack (long integer)	FISTP READINGS

Operation

The integer store and pop stack instruction rounds the contents of the stack top to an integer (according to the RC field of the control word) and transfers the result to the memory destination. The floating point stack is popped following the transfer. The destination may be any of the binary integer data types.

Exceptions

I	Z	D	O	U	P
X					X

FISUB—Integer Subtract

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m4rm	CD 1A m4rm	125 + EA (108-143) + EA	ST ← ST mem-op (short integer)	FISUB BASE
9B DE m4rm	CD 1E m4rm	120 + EA (102-137) + EA	ST ← ST - mem-op (word integer)	FISUB SIZE

Operation

This instruction subtracts the integer memory operand from the top of the stack and returns the difference to the top of the stack.

Exceptions

I	Z	D	O	U	P
X		X	X		X

FISUBR—Integer Subtract Reversed

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DA m5rm	CD 1A m5rm	125 + EA (109-144) + EA	ST ← mem-op − ST (short integer)	FISUBR FLOOR
9B DE m5rm	CD 1E m5rm	120 + EA (103-139) + EA	ST ← mem-op − ST (word integer)	FISUBR BALANCE

Operation

The integer subtract reversed instruction subtracts the stack top from the integer memory source and returns the difference to the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X		X

FLD—Load Real

Format

Stack element to Stack top

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 C0 + i	CD 19 C0 + i	20 17-22	T_1 ← ST(i) push stack ST ← T_1	FLD ST(2)

Memory operand to Stack top

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m0rm	CD 19 m0rm	43 + EA (38-56) + EA	push stack ST ← mem-op (short real)	FLD READING
9B DD m0rm	CD 1D m0rm	46 + EA (40-60) + EA	push stack ST ← mem-op (long real)	FLD TEMPERATURE
9B DB m5rm	CD 1B m5rm	57 + EA (53-65) + EA	push stack ST ← mem-op (temp real)	FLD SAVEREADING

Operation

The load real instruction pushes the source operand onto the top of the floating point stack. This is done by decrementing the stack pointer by one and then copying the contents of the source to the new stack top. The source may be a stack element on the stack (ST(i)), or any of the real data types in memory. Short and long real source operands are converted to temporary real automatically. Executing FLD ST(0) duplicates the old stack top in the new stack top.

Exceptions

I	Z	D	O	U	P
X		X			

FLDCW—Load Control Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m5rm	CD 19 m5rm	10 + EA (7-14) + EA	processor control word ← mem-op	FLDCW CONTROL

Operation

This instruction replaces the current processor control word with the word defined by the source operand.

Exceptions

I Z D O U P

Description

This instruction is typically used to establish, or change, the 8087's mode of operation.

Note

If an exception bit in the status word is set, loading a new control word that unmasks that exception and clears the interrupt enable mask will generate an immediate request before the next instruction is executed. When changing modes, the recommended procedure is to first clear any exceptions and then load the new control word.

FLDENV—Load Environment

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m4rm	CD 19 m4rm	40 + EA (35-45) + EA	8087 environment ← mem-op	FLDENV ENV_STORE

Operation

The load environment instruction reloads the 8087 environment from the memory area defined by the source operand. This data should have been written by a previous FSTENV/FNSTENV instruction.

Exceptions

I Z D O U P

Description

CPU instructions may immediately follow FLDENV, but no subsequent NDP instruction should be executed without an intervening FWAIT or assembler-generated WAIT.

Note

Loading an environment image that contains an unmasked exception causes an immediate interrupt request from 8087 (assuming IEM = 0 in the environment image).

FLDLG2—Load $\text{Log}_{10}2$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EC	CD 19 EC	21 18-24	push stack $ST \leftarrow \log_{10}2$	FLDLG2

Operation

The load log base 10 of 2 instruction pushes the value $\log_{10}2$ onto the top of the floating point stack. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDLN2—Load Log$_e$2

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 ED	CD 19 ED	20	push stack	FLDLN2
		17-23	ST ← log$_e$2	

Operation

The load log base e of 2 instruction pushes the value log$_e$2 onto the top of the floating point stack. This constant has temporary real precision of 64 bits with an accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDL2E—Load Log$_2$e

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EA	CD 19 EA	18	push stack	FLDL2E
		15-21	ST ← log$_2$e	

Operation

The load log base 2 of e instruction pushes the value log$_2$e onto the top of the floating point stack. This value has full temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

FLDL2T—Load $\text{Log}_2 10$

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E9	CD 19 E9	19 16-22	push stack ST ← $\log_2 10$	FLDL2T

Operation

The load log base 2 of 10 instruction pushes the constant $\log_2 10$ onto the stack. This constant has temporary real precision of 64 bits with accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDPI—Load π

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EB	CD 19 EB	19 16-22	push stack ST ← π	FLDPI

Operation

This instruction pushes π onto the top of the stack. The π value has full temporary real precision of 64 bits with an accuracy of approximately 19 decimal digits.

Exceptions

I	Z	D	O	U	P
X					

FLDZ—Load +0.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 EE	CD 19 EE	14	push stack	FLDZ
		11-17	ST ← 0.0	

Operation

The load zero instruction pushes the value +0.0 onto the top of the floating point stack. The constant has temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

FLD1—Load +1.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E8	CD 19 E8	18 15-21	push stack ST ← 1.0	FLD1

Operation

This instruction pushes the constant +1.0 onto the top of the floating point stack. This constant has full temporary real precision of 64 bits.

Exceptions

I	Z	D	O	U	P
X					

FMUL—Multiply Real

Format

Stack top and Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 C8+i	CD 18 C8+i	138* 130-145*	ST ← ST * ST(i)	FMUL ST,ST(3)
9B DC C8+i	CD 1C C8+i	138* 130-145*	ST(i) ← ST(i) * ST	FMUL ST(2),ST

*Clocks are 97
90-105 when one or both operands are short.

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m1rm	CD 18 m1rm	118+EA (110-125)+EA	ST ← ST * mem-op (short real)	FMUL SPEED
9B DC m1rm	CD 1C m1rm	161 + EA* (154-168) + EA*	ST ← ST * mem-op (long real)	FMUL HEIGHT

*Clocks are 120 + EA
(112-126) + EA when one or both operands are short

Operation

The multiply real instruction multiplies the destination operand by the source and returns the product to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FMULP—Multiply Real and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE C8 + i	CD 1E C8 + i	142* 134-148*	ST(i) ← ST(i) * ST pop stack	FMULP ST(2),ST

*Clocks are $\substack{100 \\ 94\text{-}108}$ when one or both operands are short.

Operation

The multiply real instruction multiplies the destination operand by the source and returns the product to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The multiply real and pop stack instruction multiplies one of the stack elements by the stack top, replaces the stack element with the product, and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FNOP—No operation

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 D0	CD 19 D0	13 10-16	ST ← ST	FNOP

Operation

This operation stores the stack top to the stack top and thus effectively performs no operation.

Exceptions

I Z D O U P

FPATAN—Partial Arctangent

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F3	CD 19 F3	650 250-800	$T_1 \leftarrow$ arctan (ST(1)/ST) pop stack ST $\leftarrow T_1$	FPATAN

Operation

The partial arctangent instruction computes the function Θ = ARCTAN (Y/X). X is taken from the top stack element and Y from ST(1). Y and X must observe the inequality $0 < Y < X < +\infty$. The instruction pops the floating point stack and returns Θ to the new stack top, overwriting the Y operand.

Exceptions

I	Z	D	O	U	P	*
				X	X	

*operands not checked

Description

This instruction assumes that the operands are valid and in-range. To be considered valid, an operand must be normalized. If an operand is either invalid or out-or-range, the instruction will produce an undefined result without signalling an exception.

FPREM—Partial Remainder

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F8	CD 19 F8	125 15-190	ST ← REPEAT (ST – ST(1))	FPREM

Operation

This instruction performs modulo division on the stack top by ST(1). FPREM produces an *EXACT* result; the precision exception does not occur. The sign of the remainder is the same as the sign of the original dividend.

Exceptions

I	Z	D	O	U	P
X		X		X	

Description

FPREM operates by performing successive subtractions. It can reduce a magnitude difference of up to 2^{64} in one execution. If FPREM produces a remainder that is less than the modulus (ST(1)), the function is complete and bit C2 of the status word condition code is cleared. If the function is incomplete. C2 is set to 1; the result in ST is then called the partial remainder.

Software can be used to inspect C2 by storing the status word following execution of FPREM and re-executing the instruction (using the partial remainder in ST as the dividend), until C2 is cleared. An alternate possibility is comparing ST to ST(1) to determine when the function is complete. If ST > ST(1), FPREM must be executed again. If ST = ST(1), the remainder is 0 and execution is complete. If ST < ST(1), execution is complete and the remainder is ST.

Note

> A context switch between the instructions in the remainder loop can be forced by a higher priority interrupting routine which needs the 8087.

One important use of FPREM is to reduce arguments (operands) of periodic transcendental functions to the range permitted by these instructions. For example, the FPTAN (tangent) instruction requires its argument to be less than $\pi/4$. Using $\pi/4$ as a modulus, FPREM will reduce an argument so that it is in the range of FPTAN. Because FPREM produces an exact result, the argument reduction does NOT introduce roundoff error into the calculations even if several iterations are required to bring the argument into range. The rounding of π produces a rounded period rather than a rounded argument.

FPREM also provides the least-significant three bits of the quotient generated by FPREM (in C_3, C_1, C_0). This is also important for transcendental argument reduction since it locates the original angle in the correct one of eight $\pi/4$ segments of the unit circle.

FPTAN—Partial Tangent

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example

9B D9 F2	CD 19 F2	450
		30-540
			Y/X ← TAN (ST)
			ST ← Y
			push stack
			ST ← X
				FPTAN

Operation

The partial tangent instruction computes the function Y/X = TAN(Θ). Θ is taken from the top stack element. The value of Θ must be within the range $0 <= \Theta < \pi/4$. The result of the operation is a ratio; y replaces Θ in the stack and X is pushed, becoming the new stack top. Θ is measured in radians.

Exceptions

I	Z	D	O	U	P	*
X					X	

*operands not checked

Description

The ratio result of FPTAN is designed to optimize the calculation of the other trigonometric functions.

This instruction assumes that the operand is valid and in-range; to be considered valid, an operand must be normalized. If the operand is invalid or out-of-range, the instruction will produce an undefined result without signalling an exception.

FRNDINT—Round to Integer

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FC	CD 19 FC	45 16-50	ST ← nearest integer (ST)	FRNDINT

Operation

This instruction rounds the top stack element to an integer.

Exceptions

I	Z	D	O	U	P
X					X

Description

Assume that ST contains the 8087 real number encoding of the decimal value 155.625. FRNDINT will change the value to 155 if the RC field of the control word is set to down or chop; or to 156 if it is set to up or nearest.

FRSTOR—Restore Saved State

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m4rm	CD 1D m4rm	202 + EA (197-207) + EA	8087 state ← mem-op	FRSTOR STATE SAVE

Operation

The restore state instruction reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction.

Exceptions

I Z D O U P

Note

CPU instructions may immediately follow FRSTOR, but no NDP instruction should be executed without an intervening FWAIT or an assembler-generated WAIT.

The 8087 resets to its new state at the conclusion of the FRSTOR. The 8087 will, for example, generate an immediate interrupt request if indicated by the exception and mask bits in the memory image.

FSAVE/FNSAVE—Save State

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m6rm	CD 1D m6rm	202 + EA (197-207) − EA	mem-op · 8087 state	FSAVE STATE_ SAVE
90 DD m6rm	CD 1D m6rm	202 + EA (197-207) · EA	mem-op − 8087 state (no wait)	FNSAVE STATE

Operation

The save state instruction writes the full 8087 state—environment plus register stack—to the memory location specified in the destination operand, and initializes the NDP. The FSAVE form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

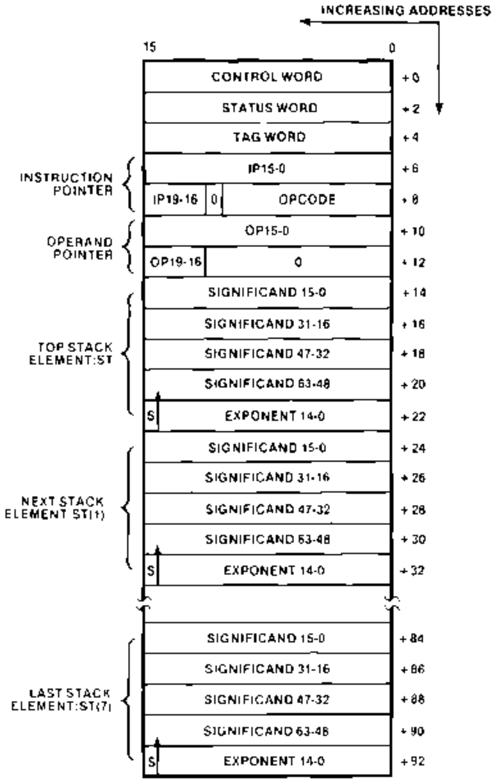
Figure 6-8 shows the 94-byte save area layout. Typically, FSAVE/FNSAVE will be coded to save this image on the CPU stack.

If an instruction is executing in the 8087 when FNSAVE is decoded, the CPU queues the save and delays its execution until the running instruction completes normally, or encounters an unmasked exception. The save image, therefore, reflects the state of the 8087 following completion of any running instruction. After writing the state image to memory, FSAVE/FNSAVE initializes the 8087 as if FINIT/FNINT had been executed.

FSAVE/FNSAVE is useful whenever a program wants to save the current state of the NDP and initialize it for a new routine. Three examples are:

1. An operating system needs to perform a context switch (suspend the task that has been running and give control to a new task);
2. An interrupt handler needs to use the 8087;
3. An application task wants to pass a "clean" 8087 stack to a sub-routine.

FSAVE/FNSAVE



NOTES
S = Sign
Bit 0 of each field is rightmost, least significant bit of corresponding register field
Bit 63 of significand is integer bit (assumed binary point is immediately to the right)

121623-15

Figure 6-8. FSAVE/FRSTOR Memory Layout

Note

FSAVE/FNSAVE, like FSTENV/FNSTENV, must be protected from any other 8087 instruction that might execute while the save is in progress. When FSAVE is coded, this can be insured by placing an explicit FWAIT in front of a subsequent no-wait mnemonic, if there is one. When FSAVE is executed with CPU interrupts disabled, an FWAIT should be executed before CPU interrupts are enabled or any subsequent 8087 instruction is executed. Because the FNSAVE initializes the NDP, there is no danger of the FWAIT causing an endless wait. Other CPU instructions may be executed between the FNSAVE and the FWAIT; this will reduce interrupt latency if the FNSAVE is queued in the 8087.

FSCALE—Scale

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FD	CD 19 FD	35 32-38	ST ← ST • $2^{ST(1)}$	FSCALE

Operation

This instruction interprets the value contained in ST(1) as an integer, and adds this value to the exponent of the number in ST. ST(1) must be in the range $-2^{15} \leqslant ST(1) < +2^{15}$ and ST(1) must be an integer.

Exceptions

I	Z	D	O	U	P
X			X	X	

Description

FSCALE is particularly useful for scaling the elements of a vector because it provides rapid multiplication or division by integral powers of 2.

Note

FSCALE assumes the scale factor in ST(1) is an integral value in the range $-2^{15} \leqslant x < 2^{15}$. If the value is not an integer, but is in-range and is greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude, i.e., it chops the value toward 0. If the value is out of range, or $0 < | x | < 1$, the instruction will produce an undefined result and will not signal an exception. The recommended practice is to load the scale factor from a word integer to ensure correct operation.

FSQRT—Square Root

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 FA	CD 19 FA	183 180-186	ST ← √ST	FSQRT

Operation

This instruction replaces the contents of the top of the stack with its square root. ST must be in the range −0 ≤ ST ≤ + ∞.

Exceptions

I	Z	D	O	U	P
X		X			X

FST—Store Real

Format

Stack top to Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD D0+i	CD 1D D0+i	18 15-22	ST(i) ← ST	FST ST(4)

Stack top to memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m2rm	CD 19 m2rm	87+EA (84-90)+EA	mem-op ← ST (short-real)	FST MEAN
9B DD m2rm	CD 1D m2rm	100+EA (96-104)+EA	mem-op ← ST (long-real)	FST READING

Operation

The store real instruction transfers the top of the stack to the destination, which may be another stack element or a short or long real memory operand. If the destination is short or long real, the significand is rounded to the width of the destination according to the RC field of the control word and the exponent is converted to the width and bias of the destination format.

Exceptions

I	Z	D	O	U	P
X			X	X	X

Note

If the stack top is tagged special (it contains ∞, a NAN, or a denormal), the stack top significand is not rounded. In this case, the least significant bits of the stack top are deleted to fit the destination. The exponent is treated in the same way. This preserves the value's identification as ∞, or a NAN (exponent of all ones), or a denormal (exponent all zeros) so that it can be properly loaded and tagged later in the program, if desired.

FSTCW/FNSTCW—Store Control Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m7rm	CD 19 m7rm	15 + EA (12-18) + EA	mem-op ← processor control word	FSTCW CONTROL
90 D9 m7rm	CD 19 m7rm	15 + EA (12-18) + EA	mem-op ← processor control word (no wait)	FNSTSW CONTROL

Operation

The store control word instructions write the current processor control word to the memory location defined by the destination. The FSTCW form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

When application tasks are running, the WAIT form of this instruction should be used. The NO WAIT form is provided for use in critical code regions where a WAIT instruction might induce an endless wait.

FSTENV/FNSTENV—Store Environment

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m6rm	CD 19 m6rm	45+EA (40-50)+EA	mem-op ← 8087 environment	FSTENV ENVIRON
90 D9 m6rm	CD 19 m6rm	45+EA (40-50)+EA	mem-op ← 8087 environment (no wait)	FNSTENV ENVIRON

Operation

This instruction writes the 8087 basic status (control word, status word, and tag word) and exception pointers to the memory location defined by the destination operand. The FSTENV form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

FSTENV/FNSTENV is often used by exception handlers because it provides access to the exception pointers which identify the offending instruction and operand.

FSTENV/FNSTENV typically saves the environment on the CPU stack. After the environment is saved, FSTENV/FNSTENV sets all exception masks in the processor; it does not affect the interrupt enable mask. Figure 6-9 shows the format of the environment data in memory. If FNSTENV is decoded while another instruction is executing concurrently in the NEU, the 8087 does not store the environment until the other instruction has completed. The data saved by this instruction, therefore, reflects the state of the 8087 AFTER any previously decoded instruction has been executed.

Note

FSTENV/FNSTENV must be allowed to complete before any other 8087 instruction is decoded. When FSTENV is coded, an assembler-generated WAIT should precede any subsequent 8087 instruction. When using FNSTENV, with CPU interrupts disabled, an explicit FWAIT should be executed before enabling CPU interrupts.

There is no risk of the FWAIT causing an endless wait. FNSTENV masks all exceptions so that interrupt requests from the 8087 are prevented.

FSTENV/FNSTENV

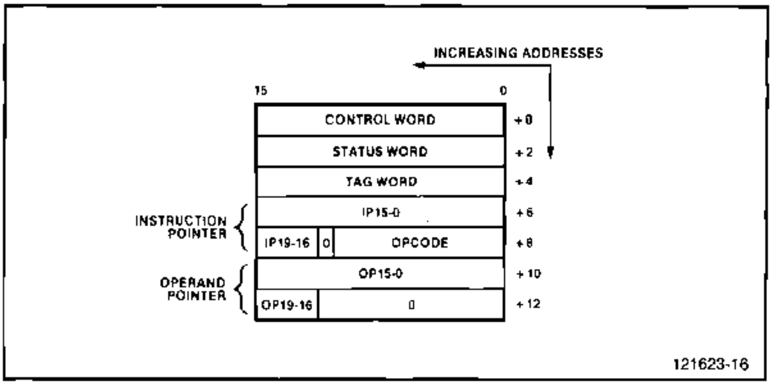


Figure 6-9. FSTENV and FLDENV Memory Layout

FSTP—Store Real and Pop

Format

Stack top to Stack element

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD D8 + i	CD 1D D8 + i	20 17-24	ST(i) ← ST pop stack	FSTP ST(2)

Stack top to memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 m3rm	CD 19 m3rm	89+EA (86-92)+EA	mem-op ← ST pop stack (short-real)	FSTP TOTAL
9B DD m3rm	CD 1D m3rm	102+EA (98-106)+EA	mem-op ← ST pop stack (long-real)	FSTP AVERAGE
9B DB m7rm	CD 1B m7rm	55+EA (52-58)+EA	mem-op ← ST pop stack (temp-real)	FSTP TEMP STORE

Operation

The store real and pop stack instruction transfers the top of the stack to the destination and then pops the stack. The destination may be another stack element, or memory operand (short-real, long-real, or temporary-real). If the destination is short or long real memory, the significand is rounded to the width of the destination according to the RC field of the control word and the exponent is converted to the width and bias of the destination format.

This instruction allows storing temporary real numbers into memory. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

Exceptions

I	Z	D	O	U	P
X			X	X	X

FSTSW/FNSTSW—Store Status Word

Format

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DD m7rm	CD 1D m7rm	15+EA (12-18)+EA	mem-op ← 8087 status word	FSTSW SAVE__STAT
90 DD m7rm	CD 1D m7rm	15+EA (12-18)+EA	mem-op ← 8087 status word (no wait)	FNSTSW SAVE_ STAT

Operation

The store status word instructions write the current value of the 8087 status word to the destination operand in memory. The FSTSW form of this instruction is preceded by an assembler-generated WAIT instruction.

Exceptions

I Z D O U P

Description

The three primary uses of this instruction are:

1. To implement conditional branching following a comparison or FPREM instruction (WAIT form).
2. To poll the 8087 to determine if it is busy (NO-WAIT form).
3. To invoke exception handlers in environments that do not use interrupts (WAIT form).

Note

If the WAIT form is used with an outstanding unmasked exception, deadlock will result.

FSUB—Subtract Real

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E0 + i	CD 18 E0 + i	85 70-100	ST ← ST − ST(i)	FSUB ST,ST(2)
9B DC E8 + i	CD 1C E8 + i	85 70-100	ST(i) ← ST(i) − ST	FSUB ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m4rm	CD 18 m4rm	105 + EA (90-120) + EA	ST ← ST − mem-op (short-real)	FSUB VALUE
9B DC m4rm	CD 1C m4rm	110 + EA (95-125) + EA	ST ← ST − mem-op (long-real)	FSUB BASE

Operation

The subtract real instruction subtracts the source operand from the destination and returns the difference to the destination. The source operand may be either the stack top, a stack element or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FSUBP—Subtract Real and Pop

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E8 + i	CD D8 E8 + i	90 75-105	ST(1) ← ST(1) – ST pop stack	FSUB
9B DE E8 + i	CD 1E E8 + i	90 75-105	ST(i) ← ST(i) – ST pop stack	FSUBP ST(2),ST

Operation

The subtract real instruction subtracts the source operand from the destination and returns the difference to the destination. The source operand may be either the stack top, a stack element or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The subtract real and pop stack instruction subtracts the stack top from one of the stack elements, replacing the stack element with the difference and then pops the floating point stack.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FSUBR—Subtract Real Reversed

Format

Stack top and Stack element

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 E8 + i	CD D8 E8 + i	87 70-100	ST ← ST(i) − ST	FSUBR ST,ST(i)
9B DC E0 + i	CD 1C E0 + i	87 70-100	ST(i) ← ST − ST(i)	FSUBR ST(3),ST

Stack top and memory operand

WAIT	op1	m/op/rm	addr1	addr2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D8 m5rm	CD 18 m5rm	105 + EA (90-120) + EA	ST ← mem-op − ST (short-real)	FSUBR INDEX
9B DC m5rm	CD 1C m5rm	110 + EA (95-125) + EA	ST ← mem-op − ST (long-real)	FSUBR VECTOR

Operation

The reverse subtract instruction subtracts the destination from the source and returns the difference to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

FSUBRP—Subtract Real Reversed and Pop

Format

WAIT	op1	op2+i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B DE E1	CD 1E E1	90 75-105	ST(1) ← ST – ST(1) pop stack	FSUBR
9B DE E0+i	CD 1E E0+i	90 75-105	ST(i) ← ST – ST(i) pop stack	FSUBRP ST(2),ST

Operation

The reverse subtract instruction subtracts the destination from the source and returns the difference to the destination. The source operand may be either the stack top, a stack element, or a short or long real operand in memory. When the source is the stack top, the destination is one of the stack elements. When the source is a stack element or memory operand, the destination is the stack top.

The reverse subtract and pop stack instruction subtracts one of the stack elements from the stack top and returns the difference to the stack element. The floating point stack is then popped.

Exceptions

I	Z	D	O	U	P
X		X	X	X	X

FTST—Test Stack Top Against +0.0

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E4	CD 19 E4	42 38-48	ST ← ST 0 0	FTST

Operation

The test instruction compares the element in the top of the floating point stack with zero and posts the result to the condition code.

Exceptions

I	Z	D	O	U	P
X		X			

Description

Condition Code Test Results

C3	C0	Result
0	0	ST is positive
0	1	ST is negative
1	0	ST is zero (+ or −)
1	1	ST is not comparable (i.e., it is a NAN or projective ∞)

FWAIT—(CPU) Wait while 8087 is busy

Format

WAIT

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B	90	3+5n 3+5n	8086 wait instruction	FWAIT

Operation

This instruction is an alternate mnemonic for the CPU WAIT instruction. FWAIT must be used instead of WAIT for 8087 emulator compatability is desired.

Exceptions

I Z D O U P

Description

The FWAIT mnemonic should be coded whenever the programmer wants to synchronize the CPU to the NDP. This means that further instruction decoding will be suspended until the NDP has completed the current instruction. This is useful if the CPU wants to inspect a value stored by the NDP (i.e., FIST should be followed by FWAIT to ensure that the value has been stored before attempting to examine it).

Note

Programmers should not code WAIT to synchronize the CPU and 8087. The routines that alter an object program for 8087 emulation change any FWAITs to NOPs but do not change any explicitly coded WAITs. The program will wait forever if a WAIT is encountered in emulated execution since there is no 8087 to drive the CPU's test pin active.

FXAM—Examine Stack Top

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 E5	CD 19 E5	17 12-23	set condition code	FXAM

Operation

The examine instruction reports the content of the top of the floating point stack as positive/negative and NAN/unnormal/denormal/normal/zero, or empty. The condition codes which can be generated are shown in table 6-15.

Exceptions

I Z D O U P

Description

Table 6-15 lists and interprets all of the condition code values that FXAM generates. Although four different encodings may be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all encodings. Bits C2 an C1 should be ignored when examining for empty.

Table 6-15. FXAM Condition Code Settings

	Condition Code			
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

FXCH—Exchange Registers

Format

WAIT	op1	op2 + i

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 C8	CD 19 C8	12 10-15	$T_1 \leftarrow ST(1)$ $ST(1) \leftarrow ST$ $ST \leftarrow T_1$	FXCH
9B D9 C8 + i	CD 19 C8 + i	12 10-15	$T_1 \leftarrow ST(i)$ $ST(i) \leftarrow ST$ $ST \leftarrow T_1$	FXCH ST(3)

Operation

The exchange instruction swaps the contents of a stack element and the stack top. If the stack element is not explicitly coded, ST(1) is used.

Exceptions

I	Z	D	O	U	P
X					

Description

Many 8087 instructions operate only on the stack top; FXCH provides an easy way to use these instructions on lower stack elements. For example, the following sequence takes the square root of the third element from the top.

```
FXCH    ST(3)
FSQRT
FXCH    ST(3)
```

FXTRACT—Extract Exponent and Significand

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F4	CD 19 F4	50 27-55	T_1 ← exponent (ST) T_2 ← significand (ST) ST ← T_1 push stack ST ← T_2	FXTRACT

Operation

The extract instruction "decomposes" the number in the stack top into two numbers that represent the actual value of the operand's exponent and significand fields. The "exponent" replaces the original operand on the stack and the "significand" is pushed onto the stack.

Exceptions

I	Z	D	O	U	P
X					

Description

FXTRACT is useful in conjunction with FBSTP for converting numbers in 8087 temporary real format to decimal representations (e.g., for printing or displaying). It can also be useful for debugging, since it allows the exponent and significand parts of a real number to be examined separately.

Note

Following execution of FXTRACT, ST (the new stack top), contains the value of the original significand expressed as a real number. The sign of this number is the same as the operand's; its exponent is 0 true (16,383 or 3FFFH biased), and its significand is identical to the original operand's. ST(1) contains the value of the original operand's true (unbiased) exponent expressed as a real number. If the original operand is zero, FXTRACT produces zeros in ST and ST(1) and BOTH are signed as the original operand.

Example

Assume ST contains a number whose true exponent is +4 (i.e., its exponent field contains 4003H). After executing FXTRACT, ST(1) will contain the real number +4.0; its sign will be positive, its exponent field will contains 4001+) (+2 true) and its significand field will contain 1Δ00 . . . 00B. In other words, the value in ST(1) will be 1.0x2^{2} = 4.

FXTRACT

If ST contains an operand whose true exponent is −7 (i.e., its exponent field contains 3FF8H), then FXTRACT will return an "exponent" of −7.0. After the instruction executes, ST(1)'s sign and exponent fields will contains C001H (negative sign, true exponent of 2) and its significand will be 1Δ1100 . . . 00B. The value in ST(1) will be $-1.11 x 2^2 = -7.0$.

In both cases, following FXTRACT, ST's sign and significand fields will be the same as the original operand's and its exponent field will contain 3FFFH, (0 true).

FYL2X—Y $\cdot$ Log$_2$X

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F1	CD 19 F1	950 900-1100	$ST_1 \leftarrow ST(1) \cdot \log_2(ST)$ pop stack $ST \leftarrow T_1$	FYL2X

Operation

This instruction calculates the function:

$$Z \leftarrow Y \cdot \log_2 X$$

X is taken from the stack top and Y from ST(1). The operands must be in the ranges $0 < X < \infty$ and $-\infty < Y < +\infty$. The instruction pops the stack and returns Z at the (new) stack top replacing the Y operand.

Exceptions

I	Z	D	O	U	P	*
					X	

*operands not checked

Note

This function optimizes the calculation of log to any base other than two since a multiplication is always required:

$$\log_n X = \frac{1}{\log_2 n} \cdot \log_2 X$$

FYL2XP1—Y * Log_2(X + 1)

Format

WAIT	op1	op2

8087 Encoding	Emulator Encoding	Execution Clocks Typical Range	Operation	Coding Example
9B D9 F9	CD 19 F9	850 700-1000	$T_1 \leftarrow ST + 1$ $T_2 \leftarrow ST(1) * \log_2 T_1$ pop stack $ST \leftarrow T_2$	FYL2XP1

Operation

This instruction calculates the function $Z = Y*LOG_2\ (X + 1)$. X is taken from the stack top and must be in the range $0 < \mid X \mid < (1- \sqrt{}2/2)$. Y is taken from ST(1) and must be in the range $-\infty\Delta < Y < \infty$. FYL2XP1 pops the floating point stack and returns Z at the new stack top, replacing Y.

Exceptions

I	Z	D	O	U	P	*
					X	

*operands not checked

Note

This instruction provides improved accuracy over FYL2X when computing the log of a number very close to 1. For example, when calculating 1 + E where E << 1, being able to input E rather than 1 + E to the function allows more significant digits to be retained.

The Macro Processing Language 7

Introduction

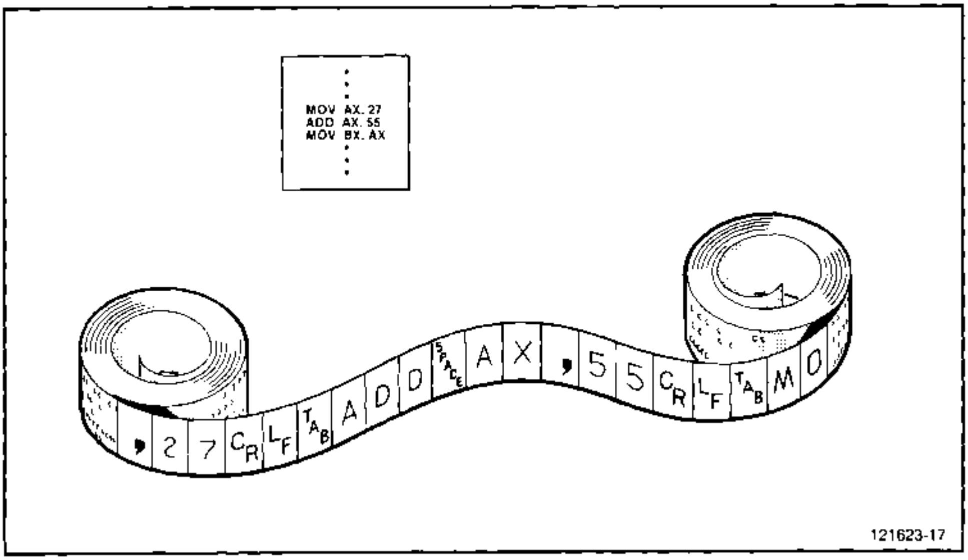
The Macro Processing Language (MPL) of the 8086/8087/8088 Macro Assembler is a string replacement facility. It permits you to write repeatedly used sections of code once and then insert that code at several places in your program. If several programmers are working on the same project, a library of macros in include files can be developed and shared by the entire team. Perhaps MPL's most valuable capability is conditional assembly. Compact configuration-dependent code is often critical to microprocessor software design, and conditional assembly of sections of code can help to achieve the most compact code possible.

This chapter documents MPL in three parts. The first section describes how to define and use your own macros. The second section defines the syntax and describes the operation of the macro processor's built-in functions. The final section of the chapter is devoted to advanced concepts in MPL.

The first two sections give enough information to begin using the macro processor. However, sometimes a more exact understanding of MPL's operation is needed. The advanced concepts section should fill those needs.

Macro Processor Overview

The macro processor views the source file in very different terms than does the assembler. To the assembler, the source file is a series of control lines, instruction lines, and directive lines. To the macro processor, the source file is a long string of characters. Figure 7-1 illustrates these two different views of the input file.



```
.
.
.
MOV AX,27
ADD AX,55
MOV BX,AX
.
.
.
```

Figure 7-1. Macro Processor versus Assembler—Two Different Views of a Source File

All macro processing of the source file is performed before your code is assembled. Because of this independence between the processing of macros and assembly of code, we must differentiate between macro-time and assembly-time. At macro-time, assembly language symbols—labels, predefined assembler symbols, EQU symbols, and the location counter are not known. The macro processor does not recognize the assembly language. Similarly, at assembly-time no information about macros is known.

The macro processor scans the source file looking for macro calls. A macro call is actually a request to the macro processor either to (re)define a user-defined macro or to replace a built-in or user-defined macro with its defined value.

This defined value or return value of a macro is the text that replaces the macro call. The return value of some macros is the null string. (The null string is a character string containing no characters.) In other words, when these macros are called, the calls are removed from the input stream. In their place, the assembler sees the return value kept.

Thus, when a macro call is encountered, the macro processor expands the call to its return value. The return value of a macro is then passed to the assembler and the macro processor continues. All characters that are not part of a macro call are passed to the assembler.

Creating and Calling Macros

The macro processor is a character string replacement facility. It searches the source file for a macro call, and then replaces the call with the macro's return value. The metacharacter (% is the default) signals a macro call. Until the macro processor finds a metacharacter, it does not process text. It simply passes the text from the source file to the rest of the assembler.

Since MPL only processes macro calls, it is necessary to call a macro in order to create other macros. The built-in function DEFINE creates macros. Built-in functions are a predefined part of the macro language, so they may be called without prior definition. The general syntax for DEFINE is:

%[*]DEFINE(*call-pattern*)[*local-symbol-list*](*macro-body*)

DEFINE is the most important MPL built-in function. This section of the chapter is devoted to describing this built-in function. Each of the symbols in the syntax above (*call-pattern*, *local-symbol-list*, and *macro-body*) are thoroughly described in the pages that follow. In some cases we have abbreviated this general syntax to emphasize certain concepts.

Creating Parameterless Macros

When you create a parameterless macro, there are two parts to a DEFINE call: the call pattern and the macro body. The call pattern defines the name used when the macro is called; the macro body defines the return value of the call.

The syntax of a parameterless macro definition is shown below:

%*DEFINE (*call-pattern*) (*macro-body*)

The '%' is the metacharacter that signals a macro call. The '*' is the literal character that is normally used when defining macros. The exact use of the literal character is discussed in the advanced concepts section.

When you define a parameterless macro, the *call-pattern* is a macro identifier that will follow the metacharacter in the source file. The rules for macro identifiers are:

- The identifier must begin with an alphabetic character (A,B,C,...,Z or a,b,c,...z).
- The remaining characters may be alphabetic, special (a question mark (?) or an underscore character ( )), or decimal digits (0,1,2,...,9).
- Only the first 31 characters of a macro identifier are significant. Upper and lower case characters are not distinguished in a macro identifier.

The *macro-body* is usually the return value of the macro call. However, the *macro-body* may contain calls to other macros. If so, the return value is actually the fully expanded macro body, including the return values of the calls to other macros. When you define a macro using the literal character, '*', shown above, macro calls contained in the body of the macro are not expanded until you call the macro. The macro call is reexpanded each time it is called.

Example 1. Nested Macro

```
%*DEFINE(ASTRING)(PHANT)
%*DEFINE(JUMBO)(ELE%ASTRING)
```

Call—`%JUMBO`

is expanded to—`ELEPHANT`

The syntax of DEFINE requires that left and right parentheses surround the *macro-body*. For this reason, you must have balanced parentheses within the macro body. (i.e., each left parenthesis must have a succeeding right parenthesis, and each right parenthesis must have a preceding left parenthesis.) We call character strings that meet these requirements *balanced-text*.

Example 2. Balanced and Unbalanced '( )'

Balanced strings—

```
(abc)
(a(b)c)
((ab(c)d)e)
```

Unbalanced strings—

```
(abc
(a(b)c
(ab(c)
```

To call a macro, you use the metacharacter followed by the *call-pattern* for the macro. (The literal character is generally not needed when you call a user-defined macro.) The macro processor will remove the call and insert the return value of the call. If the macro body contains any call to other macros, they will be replaced with their return values.

Example 3. Macro Calls

```
%*DEFINE(ME)(I LIKE)
%*DEFINE(WHAT(OBJECT))(MY %OBJECT)
```

Calls—

```
%ME               → I LIKE
%WHAT(BIKE)       → MY BIKE
%ME %WHAT(JOB).  → I LIKE MY JOB.
```

Once a macro has been created, it may be redefined by a second call to DEFINE.

Example 4. Redefinition of Macros

```
%*DEFINE(LINCOLN)(GETTYSBURG)

%LINCOLN WILL EXPAND TO → GETTYSBURG

%*DEFINE(LINCOLN)(ONE CENT)

%LINCOLN WILL EXPAND TO → ONE CENT
```

The three examples below show several macro definitions. Their return values are also shown.

> NOTE
>
> In order to postpone discussion of the use of local macro symbols for labels, location counter relative addressing (with '$') is used in these examples. This is done for simplicity, but is not generally recommended because different addressing modes produce different instruction sizes which will affect the location counter offset required.

Example 1:

Macro definition at the top of the program:

```
%*DEFINE(MOVE)(
MOV CX,100
LEA SI,TABLE1
LEA DI,TABLE2
REP MOVSW
)
```

Macro call as it appears in program:

```
PUSH CX
%MOVE
POP CX
```

The program after the macro processor makes the expansion:

```
PUSH CX
MOV CX,100
LEA SI,TABLE1
LEA SI,TABLE2
REP MOVSW
POP CX
```

Example 2:

Macro definition at the top of the program:

```
%*DEFINE (ADD5)(
MOV CX,100
MOV SI,0
MOV AX,TABLE2[SI]
ADD AX,5
MOV TABLE2[SI],AX
INC SI
INC SI
LOOPZ $-13
)
```

The macro call as it appears in the original program body:

```
PUSH AX
%ADD5
POP AX
```

The program after macro expansion:

```
PUSH AX
MOV CX,100
MOV SI,0
MOV AX,TABLE2[SI]
ADD AX,5
MOV TABLE2[SI],AX
INC SI
INC SI
LOOPZ $-13
POP AX
```

Example 3:

Macro definition at the top of the program:

```
%*DEFINE(MOV_AND_ADD)(
%MOVE
%ADD5
)
```

The macro call as it appears in the body of the program:

```
%MOVE_AND_ADD
```

The program after macro expansion:

```
MOV CX,100
LEA SI,TABLE1
LEA SI,TABLE2
REP MOVSW
MOV CX,100
MOV SI,0
MOV AX,TABLE2[SI]
ADD AX,5
MOV TABLE2[SI],AX
INC SI
INC SI
LOOPZ $-13
```

Creating Macros with Parameters

If the only thing the macro processor could do was simple string replacement, then it would not be very useful for most programming tasks. Each time we wanted to change even the simplest part of the macro's return value we would have to redefine the macro. Parameters in macro calls allow more general-purpose macros.

Parameters leave holes in a macro body that are filled in when you call the macro. This permits you to design a single macro that produces code for many typical programming operations.

The term parameter refers to both the formal parameters that are specified when the macro is defined (the holes), and the actual parameters or arguments that are specified when the macro is called (the fill-ins).

The syntax for defining macros with parameters is very similar to the syntax for macros without parameters. The *call-pattern* that we described earlier actually includes both the *macro-name* and an optional *parameter-list*. With this addition the syntax for the DEFINE built-in function becomes:

%*DEFINE(*macro-name*[(*parameter-list*)]) (*macro-body*)

NOTE

This is not the only format allowable but a specific case. The parentheses are not the only delimiters that can be used (see the Advanced MPL Concepts section).

The *macro-name* must be a valid macro identifier.

The *parameter-list* is a list of macro identifiers separated by macro delimiters, usually commas. These identifiers comprise the formal parameters used in the macro. The macro identifier for each parameter in the list must be unique.

The *macro-body* must be a *balanced-text* string. The locations of parameter replacement (the placeholders to be filled in by the actual parameters) are indicated by placing a parameter's name preceded by the metacharacter in the macro body. The parameters may be used any number of times and in any order within the macro body. (If a user-defined macro has the same macro identifier name as one of the parameters to the macro, the macro may not be called within the macro body since the name would be recognized as a parameter.)

The example below shows the definition of a macro with three parameters—SOURCE, DEST, and COUNT. The macro will produce code to copy any number of words from one part of memory to another.

```
%*DEFINE(MOVE_ADD_GEN(SOURCE,DEST,COUNT))(
                      MOV CX,%COUNT
                      MOV SI,0
                      MOV AX,%SOURCE[SI]
                      MOV %DEST[SI],AX
                      INC SI
                      INC SI
                      LOOPZ $-13
)
```

To call a macro with parameters you must use the metacharacter followed by the macro's name as with parameterless macros. However, a list of the actual parameters must follow. In the most simple case these actual parameters are enclosed with parentheses, and separted from each other by commas. The actual parameters must be *balanced-text* and may optionally contain calls to other macros. A simple call to the macro defined above might be:

```
%MOVE_ADD_GEN(INPUT, STORE, 100H)
```

The above macro call produces the following code:

```
MOV CX,100H
MOV SI,0
MOV AX,INPUT[SI]
MOV STORE[SI],AX
INC SI
INC SI
LOOPZ $-13
```

LOCAL Symbols in Macros

The LOOPZ instruction uses offset addressing ($-13). However, if the instructions in the macro MOVE _ADD_ GEN are modified, the offset address ($-13) may need to be changed. This is a disadvantage of using offset addressing. If we chose to use a label for the jump destination, macro modification would generally not affect the label. However, the macro could only be used once, since a second call to the macro would cause a conflict in label definitions at assembly time. We could make the label a parameter and specify a different symbol name each time we call the macro. A preferable way to ensure a unique label for each macro call is to put the label in a LOCAL list. The LOCAL list construct allows you to use macro identifiers to specify assembly-time symbols. Each use of a LOCAL symbol in a macro guarantees that the symbol will be replaced by a unique assembly-time symbol each time the macro is called.

The macro processor increments a counter once for each symbol used in the list every time your program calls a macro that uses the LOCAL construct. Symbols in the LOCAL list, when used in the macro body, receive a two to five digit suffix that is the hexadecimal value of the counter. The first time you call a macro that uses the LOCAL construct the suffix is '00'.

The syntax for the LOCAL construct in the DEFINE function is shown below. (This is the complete syntax for the built-in function DEFINE):

%*DEFINE(*macro-name*[*parameter-list*]) [LOCAL *local-list*] (*macro-body*)

The *local-list* is a list of valid macro identifiers separated by spaces. Since these macro identifiers are not parameters, the LOCAL construct in a macro has no affect on the syntax of a macro call.

The example below shows the MOVE _ADD_GEN macro definition that uses a LOCAL list:

```
%*DEFINE(MOVE_ADD_GEN(SOURCE,DEST,COUNT)) LOCAL LABEL (
                     MOV CX,%COUNT
                     MOV SI,0
            %LABEL:  MOV AX,%SOURCE[SI]
                     MOV %DEST[SI],AX
                     INC SI
                     INC SI
                     LOOPZ %LABEL
)
```

The following macro call:

```
%MOVE_ADD_GEN(DATA,FILE,67)
```

would produce this code if this is the eleventh call to a macro using a LOCAL list:

```
MOV CX,67
MOV SI,0
LABEL0A: MOV AX,DATA[SI]
MOV FILE[SI],AX
INC SI
INC SI
LOOPZ LABEL0A
```

Since macro identifiers follow the same rules as ASM86, you can use any macro identifier in a LOCAL list. However, if you use long identifier names, they should be restricted to 26 characters. Otherwise the label suffix may cause the identifiers to exceed 31 characters and the excess characters would be truncated.

The Macro Processor's Built-in Functions

The macro processor has several built-in or predefined macro functions. These built-in functions perform many useful operations that would be difficult or impossible to produce in a user-defined macro. An important difference between a user-defined macro and a built-in function is that user-defined macros may be redefined, while built-in functions cannot be redefined.

We have already seen one of these built-in functions, DEFINE. DEFINE creates user-defined macros. DEFINE does this by adding an entry in the macro processor's table of macro definitions. Each entry in the table includes the *call-pattern* for a macro, and its *macro body*. Entries for the built-in functions are present when the macro processor begins operation.

Other built-in functions perform numerical and logical expression evaluation, affect control flow of the macro processor, manipulate character strings, and perform console I/O (see Appendix D for a listing of the MPL built-in functions).

Comment, Escape, Bracket and METACHAR Built-in Functions

Comment Function

The Macro Processing Language can be very subtle, and the operation of macros written in a straightforward manner may not be immediately obvious. Therefore, it is often necessary to comment your macro definitions.

The macro processor's comment function has the following syntax:

%'*text*'

or

%'*text end-of-line*

The comment function always evaluates to the null string. Two terminating characters are recognized, the apostrophe and the *end-of-line* (line feed character, ASCII 0AH). The second form of the call allows you to spread macro definitions over several lines, while avoiding any unwanted *end-of-lines* in the return value. In either form of the comment function, the *text* or comment is not evaluated for macro calls.

The example below shows a commented macro definition:

```
    %*DEFINE(MOVE_ADD_GEN(SOURCE,DEST,COUNT)) LOCAL LABEL (
                     MOV CX,%COUNT  %'COUNT SHOULD BE A CONSTANT
                     MOV SI,0
             %LABEL %' %LABEL IS A LOCAL SYMBOL IT WILL HAVE A NUMBER ADDED
: MOV AX,%SOURCE[SI]  %'SOURCE MUST BE A WORD ADDRESS'
                     MOV %DEST[SI],AX %'DEST MUST ALSO BE A WORD ADDRESS'
                     INC SI
                     INC SI
                     LOOPZ %LABEL  %'THIS WILL HAVE THE SAME NUMBER ADDED
        %'AS THE %LABEL ABOVE'
)
```

Call to above macro:

```
%MOVE_ADD_GEN(DATA, STOR, 20H)
```

Return-value from above call:

```
        MOV CX,20H    MOV SI,0
LABEL07: MOV AX,DATA[SI]
        MOV STOR[SI],AX
        INC SI
        INC SI
        LOOPZ LABEL07
```

Notice that the comments that were terminated with *end-of-line* removed the *end-of-line* character along with the rest of the comment.

Note that the metacharacter is *not* recognized as flagging a call to the macro processor when it appears in the comment function.

Escape Function

Occasionally, it is necessary to prevent the macro processor from processing text. There are two built-in functions that perform this operation: the escape function and the bracket function.

The escape function interrupts the processor from its normal scanning of text. The syntax for this function is shown below:

%n text-n-characters-long

The metacharacter followed by a single decimal digit designates that the specified number of characters (maximum is 9) shall not be evaluated. The escape function is useful for inserting a metacharacter as text, adding a comma as part of an argument, or placing a single parenthesis in a character string that requires balanced parentheses.

Several examples of the escape function are shown below:

MACCALL is defined as follows:

```
%*DEFINE(MACCALL(ARG1, ARG2, ARG3))
(
      ;  %ARG1
      ;  %ARG2
      ;  %ARG3
)
```

```
          Before Macro Expansion                  After Macro Expansion
                                                   (actual parameters)

; COMPUTE 10%1% OF SUM             → ;COMPUTE 10% OF SUM
%MACCALL(JANUARY 23%1, 1980,       → ;JANUARY 23, 1980
         MARCH 15%1, 1980,         → ;MARCH 15, 1980
         APRIL 9%1, 1980)          → ;APRIL 9, 1980
%MACCALL(1%1) ADD INPUTS,          → ;1) ADD INPUTS
         2%1) DIVIDE BY INPUT      → ;2) DIVIDE BY INPUT
         3%1) GET INPUTS)          → ;3) GET INPUTS
```

Bracket Function

The other built-in function that inhibits the macro processor from expanding text is the bracket function. The syntax of the bracket function is shown below:

%(balanced-text)

The bracket function inhibits all macro processor expansion of the text contained within the parentheses except for the escape function, the comment function, and parameters which are still recognized. Since there is no restriction on the length of the text within the bracket function, it is often easier to use than the escape function. However, since balanced text is required and the metacharacter is interpreted, sometimes the bracket function does not do what you want and the escape function must be used.

Consider the following macro:

```
%*DEFINE(DW(LIST,NAME)) (
%NAME  DW %LIST
)
```

The macro above will add DW statements to the source file. It uses two parameters: one for the variable name and one for the DW expression list. Without the bracket or several escape functons we would not be able to use more than one expression in the list, since the first comma would be interpreted as the delimiter separating the macro parameters. The bracket function permits more than one expression in the LIST argument:

```
%DW(%(198H, 3DBH, 163BH),PHONE)→ PHONE DW 198H, 3DBH, 163H
```

In the example above the bracket function prevents the character string '198H, 3DBH, 163BH' from being evaluated as separate parameters.

METACHAR Function

The built-in function METACHAR allows you to redefine the *metacharacter* initially (%). Its syntax is shown below:

%METACHAR(*balanced-text*)

The following example changes the metacharacter from (%) to (&):

```
%METACHAR(&)
```

The balanced-text argument may be any number of characters long. However, only the first character in the string, i.e., the character immediately after the '(', is taken to be the new metcharacter. Extreme caution should be taken when using METACHAR, since it can have catastrophic effects. Consider the example below:

```
%METACHAR( & )
```

In this example METACHAR defines the *space* character to be the new metacharacter, since it is the first character in the *balanced-text* string!

Numbers and Expressions in MPL

Many of the built-in functions recognize and evaluate numerical expressions in their arguments. MPL uses the same rules for representing numbers as ASM86 (see Chapter 3):

- Numbers may be represented in base 2 (B suffix), base 8 (O or Q suffix), base 10 (D suffix or no suffix), and base 16 (H suffix).
- Internal representation of numbers is 17 bits (−0FFFFH to +0FFFFH). The processor does not recognize real or long integer numbers.
- The operators recognized by the macro processor and their order of precedence is shown in the list below (see Chapter 4 for discussion of these operators):
  1. ( ) (highest precedence)
  2. HIGH, LOW,
  3. *, /, MOD, SHL, SHR
  4. +, −(both unary and binary forms)
  5. EQ, NE, LE, LT, GE, GT
  6. NOT
  7. AND
  8. OR, XOR (lowest precedence)

The macro processor cannot access the assembler's symbol table. The values of labels, location counter, and EQU symbols are not known during macro-time expression evaluation. Any attempt to use assembly-time symbols in a macro-time expression generates an error. However, you can define macro-time symbols with the predefined macro SET.

SET Macro

SET assigns the value of the numeric expression to the identifier, *macro-id*, and stores the macro-id in the macro-time symbol table. *macro-id* must follow the same syntax conventions used for other macro identifiers. SET has the following syntax:

%SET(*macro-id*, *expression*)

The SET macro call affects the macro-time symbol table only; when SET is encountered in the source file, the macro processor replaces it with the null string. Symbols defined by SET can be redefined by a second SET call, or defined as a macro by a DEFINE call. In fact, if you ever assemble your source with the GEN control in effect you will see that SET uses the DEFINE built-in function to create the numeric symbols.

The following examples show several ways to use SET:

```
   Before Macro Expansion           After Macro Expansion

%SET(COUNT,0)                     → null string
%SET(OFFSET,16)                   → null string
MOV AX, %COUNT + %OFFSET          → MOV AX, 00H + 10H
MOV BX, %COUNT                    → MOV BX, 00H
```

SET can also be used to redefine symbols in the macro-time table:

```
%SET(COUNT,%COUNT + %OFFSET)      → null string
%SET(OFFSET,%OFFSET * 2)          → null string
MOV AX, %COUNT + %OFFSET          → MOV AX, 10H + 20H
MOV BX, %COUNT                    → MOV BX, 10H
```

SET is a predefined macro, not a built-in function; as such it may be redefined, however, you will then loose this function.

EVAL Function

The built-in function EVAL accepts an expression as its argument and returns the expression's value in hexadecimal. The syntax for EVAL is:

%EVAL(*expression*)

The *expression* argument must be a legal macro-time expression.

The *return-value* from EVAL follows ASM86's rules for representing hexadecimal numbers (see Chapter 3). EVAL always returns an expression with at least 3 characters even if the argument evaluates to a single digit. The leading character is always a decimal-digit (0,1,2,...,9). The remaining digits may be any hexadecimal digit (0,1,2,...E,F). The trailing character must always be the hexadecimal suffix (H). The following examples show the *return-value* from EVAL:

```
   Before Macro Expansion                   After Macro Expansion

MOV AX, %EVAL(1+1); move two to AX    → MOV AX, 02H; move
                                        two to AX
COUNT EQU %EVAL(33H + 15H + 0F00H)    → COUNT EQU 0F48H
ADD AX, %EVAL(10H-((13 + 6) * 2) + 7) → ADD AX, 0FFF1H
%SET(NUM1,44)                         → null string
%SET(NUM2,25H)                        → null string
MOV AX, %EVAL(%NUM1 LE %NUM2)         → MOV AX, 00H
```

Logical Expressions and String Comparisons in MPL

Several built-in functions return a logical value when they are called. Like relational operators that compare numbers and return true or false (−1H or 00H), respectively, these built-in functions compare character strings. If the function evaluates to 'TRUE,' then it returns the character string '−1H' (all ones). If the function evaluates to 'FALSE,' then it returns '00H' (zeros).

The built-in functions that return a logical value compare two *balanced-text* string arguments and return a logical value based on that comparison. The list of string comparison functions below shows the syntax and describes the type of comparison made for each. Both arguments to these functions may contain macro calls (the calls are expanded before the comparison is made).

%EQS(*arg1*,*arg2*) True if both arguments are identical; equal

%NES(*arg1*,*arg2*) True if arguments are different in any way; not equal

%LTS(*arg1*,*arg2*) True if first argument has a lower value than second argument; less than

%LES(*arg1*,*arg2*) True if first argument has a lower value than second argument or if both arguments are identical; less than or equal

%GTS(*arg1*,*arg2*) True if first argument has a higher value than second argument; greater than

%GES(*arg1*,*arg2*) True if first argument has a higher value than second argument, or if both arguments are identical; greater than or equal

Before these functions perform a comparison, both arguments are completely expanded. Then the ASCII value of the first character in the first string is compared to the ASCII value of the first character in the second string. If they differ, then the string with the higher ASCII value is considered to be greater. If the first characters are the same, then the process continues with the second character in each string, and so on. Only two strings of equal length that contain the same characters in the same order are equal.

Before Macro Expansion		After Macro Expansion	
%GTS(16D,11H)	→	−1H *true*	these macros compare strings not numerical values; ASCII '6' is greater than ASCII '1'
%EQS(ABC,ABC)	→	−1H *true*	the character strings are identical
%EQS(ABC, ABC)	→	00H *false*	the space after the comma is part of the second argument
%LTS(CBA,cba)	→	−1H *true*	the lower-case characters have a higher ASCII value than upper-case
%GES(ABCDEF,ABCDEF)	→	00H *false*	the space at the end of the second argument makes the second argument greater than the first

As with any other macro, the arguments to the string comparison macros can be other macros.

```
%*DEFINE(DOG) (CAT)
%*DEFINE(MOUSE) (%DOG)
%EQS(%DOG,%MOUSE)          -1H
                           true
```

Control Flow and Conditional Assemblies

Some built-in functions expect logical expressions in their arguments. Logical expressions follow the same rules as numeric expressions. The difference is in how the macro interprets the 17-bit value that the expression represents. Once the expression has been evaluated to a 17-bit value, MPL uses only the low-order bit to determine whether the expression is TRUE or FALSE. If the low-order bit is a one (the 17-bit numeric value is odd), the expression is TRUE. If the low-order bit is a zero (the 17-bit value is even), the expression is FALSE.

Typically, you will use either the relational operators (EQ, NE, LE, LT, GT, or GE) or the string comparison functions (EQS, NES, LES, LTS, GTS, or GES) to specify a logical value. Since these operators and functions always evaluate to -1H (all ones) or 00H (all zeros), you needn't worry about the single bit test. But remember, all numeric expressions are valid, and regardless of the value of the other 16 bits, only the least significant bit counts.

IF Function

The IF built-in function evaluates a logical expression, and based on that expression, expands or withholds it text arguments. The syntax for the IF macro is shown below:

%IF (*expression*) THEN (*balanced-text1*) [ELSE (*balanced-text2*)] FI

The IF function first evaluates the *expression*. If the low order bit is one, then *balanced-text1* is expanded; if the low order bit is zero and the optional ELSE clause is included in the call, then *balanced-text2* is expanded. If the low order bit is zero and the ELSE clause is not included, the IF call returns the null string. FI must be included to terminate the call.

IF calls can be nested; when they are, the ELSE clause refers to the most recent IF call that is still open (not terminated by FI). FI terminates the most recent IF call that is still open.

Several examples of IF calls are shown below:

This is a simple example of the IF call with no ELSE clause.

```
%IF (%GTS(0FFH,%VAR)) THEN (MOV AX, %VAR) F1
```

This is the simple form of the IF call with an ELSE clause.

```
%IF (%EQS(ADD AX, %OPERATION)) THEN (ADD BX, %R1) ELSE (ADD BX, %R2) FI
```

This is an example of several nested IF calls.

```
open first IF      %IF (%EQS(%OPER,ADD)) THEN ( ADD AX,DATUM
                            ) ELSE (
open second IF        %IF (%EQS(%OPER,SUB)) THEN ( SUB AX,DATUM
                               ) ELSE (
open third IF             %IF (%EQS(%OPER,MULT)) THEN ( MUL DATUM
                                   ) ELSE (DIV DATUM
close third IF            )FI
close second IF       )FI
close first IF     )FI
```

Example 5. Conditional Assembly

```
%SET (DEBUG, 1)
%IF (DEBUG) THEN (
MOV  AX, DEBUG_FLAG
OUT  AX, 2
)
MOV  BX, OFFSET ARRAY
SUB  BX, 1
       .
       .
       .
```

will expand to:

```
MOV  AX, DEBUG_FLAG
OUT  AX, 2
MOV  BX, OFFSET ARRAY
SUB  BX, 1
```

You could change the %SET to

```
%SET (DEBUG, 0)
```

to turn off the 'debug' code.

WHILE Function

The IF macro is useful for implementing one kind of conditional assembly—including or excluding lines of code in the source file. However, in many cases this is not enough. Often you may wish to perform macro operations until a certain condition is met. The built-in function WHILE provides this facility.

The syntax of the WHILE macro is shown below:

%WHILE (*expression*)   (*balanced-text*)

The WHILE function evaluates the *expresson* . If the least significant bit is one, then the *balanced-text* is expanded; otherwise, it is not. Once the *balanced-text* has been expanded, the logical argument is retested and if the least significant bit is still one, then the *balanced-text* is again expanded. This continues until the logical argument proves false (the least significant bit is 0).

Since the macro continues processing until *expression* is false, the *balanced-text* should modify *expression*, or else WHILE may never terminate.

A call to the built-in function EXIT will always terminate a WHILE macro. EXIT is described below.

The following examples show two common uses of the WHILE macro:

```
%SET(COUNTER,5)
%WHILE(%COUNTER GT 0)
(INC BX
 %SET(COUNTER, %COUNTER - 1)
)
%WHILE(%COUNT LT OFFH) ( HLT
                        %SET(COUNT, %COUNT+1) )
```

These examples use the SET macro and a macro-time symbol to count the iterations of the WHILE macro.

REPEAT Function

MPL offers another built-in function that will perform the counting loop automatically. The built-in function REPEAT expands its *balanced-text* a specified number of times. The general form of the call to REPEAT is shown below:

%REPEAT (*expression*) (*balanced-text*)

Unlike the IF and WHILE macros, REPEAT uses the *expression* for a numerical value that specifies the number of times the *balanced-text* will be expanded. The expression is evaluated once when the macro is first called, then the specified number of iterations is performed.

The examples below will perform the same text insertion as the WHILE examples above.

```
%REPEAT (5) (INC BX
)

%REPEAT (OFFH - COUNT) (HLT
)
```

Note that the line feeds preceding the right paren in each of the above examples are necessary for correct assembly.

EXIT Function

The EXIT built-in function terminates expansion of the most recently called REPEAT, WHILE or user-defined macro. It is most commonly used to avoid infinite loops (e.g., a WHILE *expression* that never becomes false, or a recursive user-defined macro that never terminates). It allows several exit points in the same macro.

The syntax for EXIT is:

%EXIT

Two examples of how you might use the EXIT macro follow:

This use of EXIT terminates a recursive macro when an odd number of bytes have been added.

```
     %*DEFINE(MEM_ADD_MEM(SOURCE,DESTIN,BYTES)) (
                 MOV AL,%SOURCE
                 ADD AL,%DESTIN
                 MOV %DESTIN,AL
                 IF (%BYTES EQ 1) THEN (%EXIT)FI
                 MOV AL, %SOURCE + 1
                 ADD AL, %DESTIN + 1
                 MOV %DESTIN + 1, AL
                 IF (%BYTES GT 2 ) THEN
(%MEM_ADD_MEM(%SOURCE+2,%DESTIN+2,%BYTES-2))FI
       )
```

The above example adds two pairs of bytes and stores the results in DESTIN. As long as there are more than two pairs of bytes to be added, the macro MEM_ADD MEM is expanded. That is, as long as BYTES is greater than 2, the expansion continues. When BYTES reaches a value of 1 (odd number of byte pairs) the macro is exited.

This EXIT is a simple jump out of a recursive loop.

```
%*DEFINE(UNTIL (CONDITION,BODY))
(%BODY
      %IF (%CONDITION) THEN (%EXIT)
      ELSE (%UNTIL(%CONDITION,%BODY)) FI
```

This example assumes that BODY is a macro that modifies CONDITION such that CONDITION eventually becomes true.

String Manipulation Built-in Functions

The purpose of the Macro Processor is to manipulate character strings. Therefore, there are several built-in functions that perform common character string manipulation functions.

LEN Function

The built-in function LEN takes a character string argument and returns the length of the character string in hexadecimal (the same format as EVAL). The character string argument to LEN is limited to 256 characters.

The syntax of the LEN macro call is shown below:

%LEN(*balanced-text*)

Several examples of calls to LEN and the hexadecimal numbers returned are shown below:

```
        Before Macro Expansion              After Macro Expansion

%LEN(ABCDEFGHIJKLMNOPQRSTUVWXYZ) → 1AH
%LEN(A,B,C)                      → 05H commas are counted
%LEN()                           → 00H
%*DEFINE(CHEESE) (MOUSE)
%*DEFINE(DOG) (CAT)
%LEN(%DOG %CHEESE)               → 09H
         ^the space after G is
          counted as part of the
          length
```

SUBSTR Function

The built-in function SUBSTR returns a substring of its text argument. The macro takes three arguments: a balanced character string to be divided and two numeric arguments. The syntax of the macro call to SUBSTR is shown below:

%SUBSTR(*balanced-text,expression1,expression2*)

balanced-text is described above.

expression1 specifies the starting character of the substring.

expression2 specifies the number of characters to be included in the substring.

If *expression1* is zero or greater than the length of the argument string, then SUBSTR returns the null string.

If *expression2* is zero, then SUBSTR returns the null string. If *expression2* is greater than the remaining length of the string, then all characters from the start character of the substring to the end of the string are included.

The examples below show several calls to SUBSTR and the value returned:

Before Macro Expansion	**After Macro Expansion**
`%SUBSTR(ABCDEFG,5,1)`	→ `E`
`%SUBSTR(ABCDEFG,5,100)`	→ `EFG`
`%SUBSTR(123(56)890,4,4)`	→ `(56)`
`%SUBSTR(ABCDEFG,8,1)`	→ `null`
`%SUBSTR(ABCDEFG,3,0)`	→ `null`

MATCH Function

The built-in function MATCH searches a character string for a delimiter character and assigns the substrings on either side of the *delimiter* to the identifiers. The syntax of the MATCH call is shown below:

%MATCH(*identifier1 delimiter identifier2*) (*balanced-text*)

identifier1 and *identifier2* are valid MPL identifiers.

delimiter is the first character to follow *identifier1*. Typically, a space or comma is used, but any character that is not a macro identifier character may be used. You can find a more complete description of delimiters in the Advanced Concepts section at the end of the chapter.

balanced-text is as described earlier in the chapter.

MATCH searches the *balanced-text* string for the specified *delimiter*. When the *delimiter* character is found, then all characters to the left of it are assigned to *identifier1* and all characters to the right are assigned to *identifier2*. If the *delimiter* is not found, the entire *balanced-text* string is assigned to *identifier1* and the null string is assigned to *identifier2*.

The following example shows a typical use of the MATCH macro.

```
%MATCH (NEXT,LIST) (10H, 20H, 30H)
       MOV SI, VAR_PTR
%WHILE (%LEN(%NEXT) NE 0) (
       MOV BX, %NEXT
       MOV AX, [BX+SI]
       ADD AX,22H
       MOV [BX+SI], AX
       %MATCH (NEXT,LIST)(%LIST)
)
```

Produces the following code:

```
                         MOV BX, 10H
   first iteration       MOV AX, [BX+SI]
   of WHILE              ADD AX,22H
                         MOV [BX+SI], AX
```

```
                     MOV BX, 20H
second iteration     MOV AX, [BX+SI]
of WHIIE             ADD AX,22H
                     MOV [BX+SI], AX

                     MOV BX, 30H
third iteration      MOV AX, [BX+SI]
of WHILE             ADD AX,22H
                     MOV [BX+SI], AX
```

Console I/O Built-in Functions

Four built-in functions perform console I/O. The first two, IN and OUT, are line oriented. IN outputs the characters '>>' as a prompt to the console, and returns the next line typed at the console including the line terminator. OUT outputs a string to the console; the return value of OUT is the null string. The syntax of both macros is shown below:

%IN

%OUT(*balanced-text*)

Several examples of how these macros can be used are shown below:

```
      %OUT(ENTER NUMBER OF PROCESSORS IN SYSTEM)
      %SET(PROC_COUNT,%IN)
      %OUT(ENTER THIS PROCESSOR'S ADDRESS)
ADDRESS EQU %IN
      %OUT(ENTER BAUD RATE)
      %SET(BAUD,%IN)
```

The following lines would be displayed at the console:

```
ENTER NUMBER OF PROCESSORS IN SYSTEM >user response
ENTER THIS PROCESSOR'S ADDRESS >user response
ENTER BAUD RATE >user response
```

The second two, CI and CO, are character oriented functions. CI returns a single character typed at the console. CI neither prompts for input nor echoes the character typed. CO outputs a single character to the console; the return value of CO is the null string. The syntax of the CI and CO macros is.

```
%CI
%CO(char)
```

The following example defines the macro NUMBER to be a string of three characters typed at the console, and echoes the characters as they are typed:

```
%DEFINE(NUMBER)()
%REPEAT(3)( %DEFINE(A)(%CI) %CO(%A)
            %DEFINE(NUMBER)(%NUMBER%A) )
```

Advanced MPL Concepts

For most programming problems, the Macro Processing Language as described above is sufficient. However, in some cases a more complete description of the macro processor's function is necessary.

However, it is impossible to describe all of the subtleties of the macro processor in a single chapter. With the rules described in this section, you should be able to discern, with a few simple tests, the answer to any specific question about MPL.

Macro Delimiters

When we discussed the syntax for defining macros, the *parameter-list* was surrounded by parentheses, and parameters were separated by commas. Because we used these delimiters to define a macro, a call to the macro required that these same delimiters be used. When we discussed the MATCH function, we mentioned that a space could be used as a delimiter. In fact the macro processor permits almost any character or group of characters to be used as a delimiter.

Regardless of the type of delimiter used to define a macro, once it has been defined, only the delimiters used in the definition can be used in the macro call. Macros defined with parentheses and commas require parentheses and commas in the macro call. Macros defined with spaces (or any other delimiter) require that specific delimiter when called.

Macro delimiters can be divided into three classes: implied blank delimiters, identifier (or id) delimiters and literal delimiters.

Implied Blank Delimiters

Implied blank delimiters are easy to use and contribute readability and flexibility to macro calls and definitions. An implied blank delimiter is one or more spaces, tabs or new lines (a carriage-return/linefeed pair or just a linefeed) in any order. To define a macro that uses the implied blank delimiter, simply place one or more spaces, tabs or new lines surrounding the parameter list and separating the formal parameters.

When you call the macro defined with the implied blank delimiter, each delimiter will match a series of spaces, tabs, or new lines. Each parameter in the call begins with the first non-blank character, and ends with the next blank character.

An example of a macro defined using implied blank delimiters is:

```
%*DEFINE(SENTENCE SUBJECT VERB OBJECT) (THE %SUBJECT %VERB %OBJECT.
```

All of the following calls are valid for the above definition:

```
   Before Macro Expansion                After Macro Expansion

 %SENTENCE TIME IS RIPE        → THE TIME IS RIPE.
 %SENTENCE CATS
           EAT
           FISH                → THE CATS EAT FISH.

%SENTENCE
         PEOPLE
 LIKE                   FREEDOM → THE PEOPLE LIKE FREEDOM.
```

Identifier Delimiters

Identifier (id) delimiters are legal macro identifiers designated as delimiters. To define a macro that uses an id delimiter in its call pattern, you must prefix the delimiter with the commercial at symbol (@). You must separate the id delimiter from the macro identifiers (formal parameters or macro name) by a blank character.

When calling a macro defined with id delimiters, an implied blank delimiter is required to precede the id delimiter, but none is required to follow the id delimiter. The @ is not required.

An example of a macro defined with id delimiters is:

```
%*DEFINE(ADD P1 @TO P2 @AND P3) (
        MOV AX, %P1
        MOV BX, AX
        ADD AX, %P2
        MOV %P2, AX
        MOV AX, BX
        ADD AX, %P3
        MOV %P3, AX
)
```

The following call:

```
%ADD ATOM TO MOLECULE AND CRYSTAL
```

returns this code when expanded:

```
MOV AX, ATOM
MOV BX, AX
ADD AX, MOLECULE
MOV MOLECULE, AX
MOV AX, BX
ADD AX, CRYSTAL
MOV CRYSTAL, AX
```

The call could also have been written

```
%ADD ATOM TOMOLECULE ANDCRYSTAL
```

Literal Delimiters

The delimiters used when we documented user-defined macros (parentheses and commas) were literal delimiters. A literal delimiter can be any character except the metacharacter.

When you define a macro using a literal delimiter you must use exactly that delimiter when you call the macro. If you do not include the specified delimiter character as it appears in the definition, it will generate a macro error.

When defining a macro, you must literalize the delimiter string, if the delimiter you wish to use meets any of the following conditions:

- uses more than one character,
- uses a macro identifier character (A-Z, 0-9,   , or ?),
- uses a commercial at (@).
- uses a space, tab, carriage-return, or linefeed.

You can use the escape function (%*n*) or the bracket function (%()) to literalize the delimiter string. Several examples of definitions and calls using a variety of literal delimiters are shown below:

This is the simple form shown earlier:

Before Macro Expansion **After Macro Expansion**

```
%*DEFINE(MAC(A,B)) (%A %B)      → null string
%MAC(4,5)                       → 4 5
```

In the following example brackets are used instead of parentheses. The commercial at symbol separates parameters:

```
%*DEFINE(MOV[A%( @ )B]) (MOV[%A],%B)
%MOV[BX @ DI]                          → MOV[BX],DI
```

In the next two examples delimiters that could be id delimiters have been defined as literal delimiters (the differences are noted):

```
%*DEFINE(ADD (A%(AND)B))(ADD %A,%B)    → null string
%ADD( AX AND 5)                        → ADD AX, 5
```

To illustrate the differences between id delimiters and literal delimiters, consider the following macro definition and call. (A similar macro definition is discussed with id delimiters):

```
%*DEFINE(ADD P1 %(TO) P2 %(AND) P3 ) (
        MOV AX, %P1
        MOV BX, AX
        ADD AX, %P2
        MOV %P2, AX
        MOV AX, BX
        ADD AX, %P3
        MOV %P3, AX
)
```

The following call:

```
%ADD COUNT TO INCR AND FACTOR
```

returns this code when expanded

```
MOV AX, COUNT
MOV BX, AX
ADD AX, INCR
MOV INCR, AX
MOV AX, BX
ADD AX, FACTOR
MOV FACTOR, AX
```

If the parameters contain strings that match the delimiters, i.e., if %P1 is ATOM, you will get incorrect results.

Literal vs. Normal Mode

In normal mode the macro processor scans text looking for the metacharacter. When it finds one, it begins expanding the macro call. Parameters are substituted and macro calls are expanded. This is the usual operation of the macro processor, but sometimes it is necessary to modify this mode of operation. The most common use of the literal mode is to prevent macro expansion. The literal character in DEFINE prevents the expansion of macros in the *macro-body* until you call the macro.

When you place the literal character in a DEFINE call, the macro processor shifts to literal mode while expanding the call. The effect is similar to surrounding the macro body with the bracket function. The escape, comment, and bracket functions are expanded; but no further processing is performed. Any calls to other macros are not expanded.

If there are no parameters in the macro being defined, the DEFINE built-in function can be called without the literal character. If the macro uses parameters, the macro processor will attempt to evaluate the formal parameters in the *macro-body* as parameterless macro calls.

The following example illustrates the difference between defining a macro in literal mode and normal mode:

```
%SET(TOM,1)
%*DEFINE(AB) (%EVAL(%TOM))
%DEFINE(CD) (%EVAL(%TOM))
```

When AB and CD are defined, TOM is equal to 1. The macro body of AB has not been evaluated due to the literal character, but the macro body of CD has been completely evaluated, since the literal character is not used in the definition. Changing the value of TOM has no affect on CD, it changes the return value of AB as illustrated below:

Before Macro Expansion **After Macro Expansion**

```
%SET(TOM,2)
%AB                                  → 02H
%CD                                  → 01H
```

The macros themselves can be called with the literal character. The return value then is the unexpanded body:

```
%*CD                                 → 01H
%*AB                                 → %EVAL(%TOM)
```

The literalized calls to AB and CD show that CD evaluates to 01H, while AB contains a macro call to EVAL with %TOM as its parameter.

Algorithm for Evaluating Macro Calls

The algorithm the macro processor uses for evaluating the source file can be seen in 6 steps:

1. Scan the source until the metacharacter is found.
2. Isolate the call pattern. See note below.
3. If macro has parameters, expand each parameter from left to right (initiate step one on actual parameter) before expanding the next parameter.
4. Substitute actual parameters for formal parameters in macro body.
5. If the literal character is not used, initiate step one on macro body.
6. Insert the result into output stream.

NOTE

> When isolating the call pattern, the macro processor is actually scanning input for the specified delimiter. All text found between delimiters is considered the actual parameter. For this reason Id delimiters need not be terminated by spaces in a call.

The terms 'input stream' and 'output stream' are used, because the return value of one macro may be a parameter to another. On the first iteration, the input stream is the source file. On the final iteration, the output stream is passed to the assembler.

The examples below illustrate the macro processor's evaluation algorithm:

```
%SET(TOM,3)
%*DEFINE(STEVE)(%SET(TOM,%TOM-1) %TOM)
%*DEFINE(ADAM(A,B)) (
DB %A, %B, %A, %B, %A, %B
)
```

Here is a call ADAM in the normal mode with TOM as the first actual parameter and STEVE as the second actual parameter. The first parameter is completely expanded before the second parameter is expanded. After the call to ADAM has been completely expanded, TOM will have the value 02H.

Before Macro Expression **After Macro Expression**

```
%ADAM(%TOM,%STEVE)          → DB 03H, 02H, 03H, 02H, 03H, 02H
```

Now reverse the order of the two actual parameters. In this call to ADAM, STEVE is expanded first (and TOM is decremented) before the second parameter is evaluated. Both parameters have the same value.

```
%SET(TOM,3)
%ADAM(%STEVE,%TOM)          → DB 02H, 02H, 02H, 02H, 02H, 02H
```

Now we will literalize the call to STEVE when it appears as the first actual parameter. This prevents STEVE from being expanded until it is inserted in the macro body, then it is expanded for each replacement of the formal parameters. Tom is evaluated before the substitution in the macro body.

```
%SET(TOM,3)
%ADAM(%*STEVE,%TOM)         → DB 02H, 03H, 01H, 03H, 00H, 03H
```

Codemacros A

This chapter describes codemacros, which define 8086, 8087, and 8088 instructions. Codemacros should not be confused with macros, which are described in Chapter 7.

A codemacro is a preset body of code which you define, a skeleton in which most instructions and values are fixed. They are automatically assembled wherever the macro is invoked (used as an instruction), which saves your rewriting them every time that sequence is needed.

However, certain names used in the definition are NOT fixed. They are stand-ins, which are replaced by names or values that you supply in the same line that invokes the codemacro. These stand-ins are called "dummy" or "formal" parameters. They simply "hold the place" for the actual parameters to come. Formal parameters thus indicate where and how the actual parameters are to be used.

You invoke the codemacro by using its name as an instruction. For example:

```
        .
        .
        .
MOV   BX, WORD3
MAC1  PARAM1, PARAM2
ADD   AX, WORD4
        .
        .
        .
```

MAC1 above represents the use of some codemacro you defined earlier. It apparently requires 2 parameters, that is, the definition used 2 formals to be replaced by these actual parameters supplied above when you invoke the codemacro.

In fact, the MOV and ADD instructions above are codemacros. The assembler's entire instruction set is defined and implemented as a large number of codemacros. (The definitions are at the end of this Appendix). Once you understand how this is done, you may add instructions to those supplied as part of the assembler.

The type of macro used to implement this assembly language is called a codemacro to distinguish it from text macros described in Chapter 7. The latter are more familiar to programmers because previous assembly languages have included such a facility. Text macros are not discussed in this chapter. The presentation below will describe creating and using codemacros.

These codemacros are encoded at codemacro definition time into a very compact form, so that all defined codemacros may reside simultaneously in memory. Each definition specifies a certain combination of parameters and will match only those. Other combinations of parameters may be accommodated by redefining the codemacro. Multiple definitions of the same codemacro name are chained together; so that when the codemacro is called, each link of the chain can be checked for a match of operands.

Since the 8086 instruction set consists of codemacros, it is natural to refer to a codemacro being called as an "instruction," and to refer to its actual parameters as "operands."

For example, the language has an ADD instruction that works properly with any general register or memory location as a destination operand or as a source operand, and works with immediate-data operands. This is achieved by defining 11 codemacros to generate the 11 different machine instructions appropriate to these different cases and combinations. The correct one is used because the specification of its formal parameters is matched by the actual parameters supplied in your source code. The details of how this works are covered in this chapter.

The definition of a codemacro begins with a line specifying its name and a list of its formal parameters, if any:

```
    CODEMACRO name [formal_ list]
or
    CODEMACRO name PREFX
```

where formal _list is a list of formal parameters, each in the form

```
    form _name:specifier_ letter [modifier__letter] [range]
```

The square brackets indicate optional items; they are not actually used in the statement that you code. The single word CODEMACRO and the name are both required. The formal parameters are optional. If they are present, then each one must be followed by one of the specifier letters A, C, D, E, M, R, S, X. After the specifier letter comes an optional modifier letter: b, d, q, t, or w. There follows an optional range specifier, which consists of a pair of parentheses enclosing either one expression, or two expressions separated by a comma. The semantics of specifiers, modifiers, and ranges are described below.

When no formals are used, you may code the keyword PREFX, indicating the codemacro is to be used as a prefix to other instructions. This too is optional. Examples of prefixes in the 8086 instruction set are LOCK and REP.

The definition ends with a line as follows:

```
    ENDM
```

Between the first and last lines of a codemacro definition is the body of the codemacro, the actual bit patterns and formal parameters which will be assembled and replaced each time the macro is invoked. Only a few kinds of directive are allowed in codemacros. They are:

1. SEGFIX
2. NOSEGFIX
3. MODRM
4. RELB
5. RELW
6. DB
7. DW
8. DD
9. Record initialization
10. RFIX
11. RFIXM
12. RNFIX
13. RNFIXM
14. RWFIX

Each of these directives, along with the special expression operand PROCLEN, are explained further on in this chapter.

Some simple examples of codemacros:

```
Codemacro STC
DB 0F9H   ; this sets the carry flag (CF) to 1.
Endm

Codemacro PUSHF
DB 9CH    ; pushes all flags into top word on stack.
Endm

Codemacro ADD dst:Ab, src:Db
DB 04H
DB src
Endm
```

The first two examples simply allow a machine instruction to be invoked by the use of a name, which is usually more easily remembered ("mnemonic") than a string of numbers.

The third example is one of the 11 macros defining the ADD instruction, or more precisely, defines one of the 11 ADD instructions. (There are 11 in order to cover all the valid combinations of parameters.) It has two formal parameters, called "dst" and "src," for destination and source operands. These formals could be called anything; for example:

```
Codemacro ADD anything:Ab, other:Db
DB 04H
DB other
Endm
```

is the identical macro in function and format.

Specifiers

Every formal parameter must have a specifier letter, which indicates what type of operand is needed to match the formal parameter. There are eight possible specifier letters:

1. A meaning Accumulator, that is AX or AL.
2. C meaning Code, i.e., a label expression only.
3. D meaning Data, i.e., a number to be used as an immediate value.
4. E meaning Effective address, i.e., either an M (memory address) or an R (register).
5. F meaning a floating point stack element, i.e., ST or ST(i).
6. M meaning a memory address. This can be either a variable (with or without indexing) or a bracketed register expression.
7. R meaning a general Register only, not an address-expression, not a register in brackets, and not a segment register.
8. S meaning a Segment register only, either CS, DS, ES, or SS.
9. T meaning the floating point stack top, i.e., ST or ST(0).
10. X meaning a direct memory reference, a simple variable name with no indexing.

A more detailed discussion of which operands match which specifier letters appears in the instruction-matching section later in this chapter.

Modifiers

The optional modifier letter imposes a further requirement on the operand, relating either to the size of data being manipulated, or to the amount of code generated by the operand. The meaning of the modifier depends on the type of the operand:

- For variables, the modifier requires the operand to be of a certain TYPE: "b" for byte, "w" for word, "d" for dword, "q" for qword, "t" for tbyte.
- For labels, the modifier requires the object code generated to be of a certain amount: "b" for an 8-bit relative displacement on a NEAR label, "w" for NEAR labels which are outside the −128 to 127 short displacement range, and "d" for FAR labels.
- For numbers, the modifier requires the number to be of a certain size: "b" for −256 through 255, and "w" for other numbers between −65,536 and 65,535. The specifier-modifier pairs "Dd"; "Dq" and "Dt" are never matched.

Note that this manual uses upper-case letters for specifiers and lower-case letters for modifiers. This is a useful language convention to clarify the code. However it is not required—as in all source code outside of strings, the distinction between upper and lower case is ignored by the assembler.

Range Specifiers

If a range is specified, it can be a single expression or two expressions separated by a comma. Each expression must evaluate to a register or a pure number, i.e., not an address. Range specifiers are not allowed with floating point stack elements, that is, src=F or T. The list of number values corresponding to range registers is given in the instruction-matching section later in this chapter. The following shows the first lines (only) of three codemacros in the current language which use range specifiers:

1. Codemacro IN dst:Aw,port:Rw(DX)
2. Codemacro ROR dst:Ew,count:Rb(CL)
3. Codemacro ESC opcode:Db(0,63),adds:Eb

The first of these is one of the four codemacros for the IN (input) instruction. It says that if a register is to specify the port from which to input a word, only DX will match this codemacro. Any other register will fail to match, and the source line will be flagged as erroneous (e.g., IN AX,BX is in error).

The second is one of the four ROtate Right codemacros. It says the word rotated can be any word register except a segment register, or any word in memory. It is to be rotated right some number of bit positions ("count"), where "count" is specified as a byte register, and further specified to be CL. No other register will match (e.g., ROR AX, DL is in error).

The third says the "opcode" supplied as the first parameter to the ESC instruction must be a byte of immediate-data of value 0 to 63 inclusive.

Segfix

SEGFIX is a directive, included in some codemacro definitions, which instructs the assembler to determine whether a segment-override prefix byte is needed to access a given memory location. If the override byte is needed, it is output as the first byte of the instruction. If it is not needed, no action is taken.

The form of the directive is:

```
SEGFIX formal  name
```

where "formal_name" is the name of a formal parameter which represents the memory address. Because it is a memory address, the formal must have one of the specifiers E, M, or X.

In the absence of a segment-override prefix byte, the 8086 hardware uses either DS or SS. Which one depends on which base register, if any, was used. BP implies SS. BX implies DS. No base register also implies DS. (This, of course, includes the three possibilities of SI alone, DI alone, or no indexing at all.) The assembler must decide whether this hardware-implied segment register is actually the one that will reach the intended memory location.

The assembler examines the segment attribute of the memory-address expression provided as the actual parameter. This attribute could be a segment, a group, or a segment register.

- If it is a segment, the assembler determines whether that segment or a group containing that segment has been ASSUMEd into the hardware-implied segment register. If so, no override byte is needed. If not, the assembler checks the ASSUMEs of other segment registers, looking for the segment or a group containing it. If found, the override byte for that segment register is issued. If not found, an error is reported.
- If it is a group, the assembler takes the same action as for a segment, except that the possibility of an including group is ruled out: the group itself must be ASSUMEd into one of the segment registers. Otherwise an error is reported.
- If it is a segment register, the assembler sees if it is the hardware-implied segment register. If so, no override byte is issued. If not, the override byte for the specified segment register is issued.

Nosegfix

NOSEGFIX is used for certain operands in those instructions for which a prefix is illegal because the instruction cannot use any other segment register but ES for that operand. This applies only to the destination operand of these string instructions: CMPS, MOVS, SCAS, STOS.

The form of the directive is:

```
NOSEGFIX segreg, formal _name
```

where "segreg" is one of the four segment registers ES, CS, SS, DS, and "formal_ name" is the name of a memory-address formal parameter. As a memory address, the formal must have one of the specifiers E, M, or X.

The only action the assembler performs when it encounters a NOSEGFIX in assembling an instruction is to perform an error check—no object code is ever generated from this directive.

The assembler looks up the segment attribute of the actual parameter (memory-address) corresponding to "formal_name." If the attribute is a segment register, it must match "segreg." If the attribute is a group, it must be ASSUMEd into "segreg." If the attribute is a segment, it or a group containing it must be ASSUMEd into "segreg." If these tests fail and "formal_name" is thus determined not to be reachable from "segreg," an error is reported.

The only value for "segreg" actually used by the string instructions listed above is ES.

Modrm

This directive instructs the assembler to create the ModRM byte, which follows the opcode byte on many of the 8086's instructions. The byte is constructed to carry the following information:

1. The indexing-type or register number to be used in the instruction.
2. Which register is (also) to be used, or more information to select the instruction.

The MODRM byte carries the information in three fields:

The mod field occupies the two most significant bits of the byte, and combines with the r/m to form 32 possible values: 8 registers and 24 indexing modes.

The reg field occupies the next three bits following the mod field, and specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.

The r/m field occupies the three least significant bits of the byte. It can specify a register as the location of an operand, or form part of the addressing-mode encoding in combination with the mod field as described above.

The bit patterns corresponding to each indexing mode and register combination are given in Chapter 1 and Appendix B. They need not concern you when you are writing codemacros, since the assembler takes care of the encoding when you provide the operands.

The form of the directive is:

```
MODRM formal__or__number, formal__name
```

where "formal__or__number" is either the name of a formal parameter, or an absolute number; and "formal _name" is the name of another formal parameter.

"formal__or _number" represents the quantity which goes into the reg field of the ModRM byte. If it is a number, then that same value is always plugged into the field every time that codemacro definition is invoked. The number in this case is a continuation of the opcode identifying which instruction the hardware is to execute.

If it is a formal, then the corresponding operand (usually a register number) is plugged in.

"formal-name" represents an effective-address parameter. The assembler examines whether the operand supplied is a register, variable, or indexed variable, and constructs the mod and r/m fields which correctly represent the operand. If the operand calls for an 8-bit or 16-bit offset displacement, the assembler generates that as well.

An example of an 8086 instruction using ModRM:

```
Codemacro ADD dst:Rw, src:Ew
Segfix src
DB 3
MODRM dst, src
Endm
```

The specifiers Rw and Ew indicate that this codemacro will match only when the actual parameters in the invocation line are a full word general register destination, and a full word source, memory or general register.

Example 1:

```
ADD DX,  [BX] [SI] becomes

00000011  10010000
76543210  76543210
```

The first byte identifies this as an ADD of a memory word into a register. This particular byte covers only 1 of the 4 cases that are possible depending on the lowest 2 bits. If bit 1 (direction) is a 0, the ADD is FROM a register TO either a register or a memory location. If bit 1 is a 1, then the ADD is TO a register FROM a register or memory location. The least significant bit, bit 0, tells whether the data being ADDed is byte (0) or word (1).

The second byte is the MODRM byte, with DX encoded as 010 in bits 5, 4, 3, a mode of 10 in bits 7, 6, and an RM of 000 (see Chapter 1 or Appendix B for more detail).

If the source line had included a variable; for example:

```
ADD  DX, MEMWORD [BX] [SI]
```

then the offset of MEMWORD (low-order byte first, high byte last) would follow the MODRM byte.

Example 2:

```
ADD DX,  [DI]

00000011  10010101
76543210  76543210
```

As a different example, consider a destination of a word in memory and a source of immediate-data. The relevant codemacros are:

```
Codemacro ADD dst:Ew,src:Dw
Segfix dst
DB81H
MODRM 0,dst
DW src
Endm

Codemacro ADD dst:Ew,src:Db (-128, 127)
Segfix dst
DB 83H
MODRM 0,dst
DB src
Endm
```

The object code generated for the instruction and data are different in the 2 cases of a byte of data or a word of data.

Furthermore, the MODRM line for these instructions specifies a "formal__or__number" field of zero, i.e., 3 bits all zero, whereas the MODRM line for the two examples above specified a field of dst, which became 010 to represent DX.

Example 3:

```
ADD [DI], 513

10000001  10000101  00000001  00000010
```

Example 4:

```
    ADD BYTE PTR [BX] [SI], 4
    10000011  10000000  00000100
```

The immediate-data byte or word follows the MODRM byte.

Relb and Relw

These directives, used in calls and jumps, instruct the assembler to generate the displacement between the end of the instruction and the label which is supplied as an operand. This means RELB generates the 1 byte (and RELW the 2 byte) displacement, or distance in bytes, between the instruction pointer value (at the end of the codemacro) and the destination address.

The directives have the following form:

```
    RELB formal__name
```

or

```
    RELW formal__name
```

where "formal__name" is the name of a formal with a "C" (Code) specifier.

The assembler assumes that all RELB and RELW directives occur immediately after a single opcode byte in the codemacro (as in all the JUMP and CALL instructions in the 8086 instruction set). It needs this assumption to determine (during codemacro matching) where the displacement starts from, so that an operand can be identified as "Cb" or "Cw." Although the assembler allows you to define codemacros in which RELB and RELW occur elsewhere in the definition (e.g., a multi-instruction codemacro), you run the risk of making the wrong match when the codemacro is invoked. If a "b" is thus matched as "w," a wasted byte is generated; if a "w" is thus matched as a "b," an error is reported.

Examples of RELB and RELW as they appear in the 8086 instruction set are:

```
Codemacro JMP place:Cw
DB 0E9H
RELW place
Endm

Codemacro JE place:Cb
DB 74H
RELB place
Endm
```

These are direct jumps to labels in the CS segment. The specifier on the formal parameter of the first macro calls for a NEAR label in the current CS segment (Cd would mean FAR). This means a 16 bit offset, able to reach any byte in the immediate 64K bytes of address higher than the start of the segment. RELW computes the distance and provides it as a word to follow the 0E9H instruction byte.

If the offset of the target is 513, then this codemacro would generate the instruction:

```
    11101001  00000001  00000010
```

The distance begins at the end of that RELW word, i.e., if you were counting the bytes to that label, the first byte counted would be the one after the 3 bytes comprising this jump.

NOTE

> A match only occurs if the label was assembled under the same ASSUME CS:name as the jump. Only if there is a match is object code actually generated.

The second example is a conditional jump, executed only if its conditions are met. In this case, a Jump if Equal, the jump occurs if ZF=0. Conditional jumps are always self-relative and limited to destinations whose distance can fit in 1 byte. This means destinations no further ahead than 127 bytes and no further behind this instruction than -128 bytes.

If the target is 99 bytes ahead, then this codemacro would generate the instruction:

```
01110100   01100011
```

The distance counted begins with the byte after these 2 bytes above.

DB, DW, and DD

These directives are similar to the DB, DW, and DD directives which occur outside of codemacro definitions (see Chapter 3); however, there are some differences in the operands they accept.

The form of the directives is:

```
    DB cmac   expression
or
    DW cmac   expression
or
    DD cmac   expression
```

where cmac_ expression is either an expression without forward references which evaluates to an absolute number; a formal parameter name; or a formal parameter name with a dot-recordfield shift construct.

An absolute number means that the same value is to be assembled every time this codemacro definition is invoked. A formal parameter means that the corresponding actual operand is to be assembled. A dot-recordfield shift construct means that the actual operand is to be shifted and then plugged in, as discussed later in this chapter.

The operands to these codemacro initializations are restricted, in that lists and DUP counts are not allowed.

Note that the DQ and DT directive are not allowed inside codemacro definitions.

Record Initializations

The record initialization directive allows you to control bit fields smaller than one byte in codemacro definitions. The form of the directive is:

```
record__name [cmac__expression__list]
```

where record__name is the name of a previously-defined record (see Chapter 3), and cmac _expression__list is a list of cmac__expressions, separated by commas. (These particular square brackets are not used in writing the list; their meaning here is that the list is optional.) A cmac _expression is, as in the above section, either a number, a formal, or a shifted formal. In addition, null cmac. expressions are allowed in the list; in which case the default record field value as specified in the RECORD definition is used.

The directive instructs the assembler to put together a byte or a word (depending on the record), using the constant numbers and supplied operands as specified in the expression list. The values to be plugged in might not fit into the record fields; in that case, the least significant bits are used, and no error is reported. In addition, a record initialization is subject to the following limitation: the number of fields in the record definition plus the number of fields being initialized by absolute numbers (by default or given), plus the number of fields initialized by shifted formal parameters *cannot* exceed 14.

RFIX

RFIX is a directive which generates two bytes: an 8086/8088 WAIT instruction (1001 1011B) followed by the first byte of an 8086/8088 ESCAPE instruction (1101 XXXB). The form of the directive is:

```
    RFIX   formal__or_ number
```

where "formal__or__number" is either the name of a formal parameter with specifier D or an absolute number. The value of "formal__or . number" specifies the low-order three bits of the second byte generated.

As an example of the use of RFIX, consider the codemacro for the 8087 instruction FLD1:

```
Codemacro  FLD1
      RFIX  001B
        DB  1110 1000B
      ENDM
```

The source statement instruction FLD generates:

```
    1001 1011 11011001 1110 1000
```

The first byte is an 8086/8088 WAIT instruction. The second byte is the first byte of an 8086/8088 ESCAPE instruction. The low-order three bits of the second byte, followed by the third byte, identify this as an FLD1 instruction.

RFIXM

RFIXM is a directive which generates the same two bytes as RFIX, but also instructs the assembler to determine whether a segment-override byte is needed to access a given memory location. The form of the RFIXM directive is:

```
    RFIXM  formal .or__number, formal._name
```

where "formal__or__number" is either the name of a formal parameter with specifier D or an absolute number, and 'formal__name' is the name of a formal parameter which represents a memory address; that is, its specifier must be 'E', 'M' or 'X'.

If the memory address uses the default segment register, no segment-override byte is needed and RFIXM generates the same two bytes as RFIX.

If the memory address requires a segment-override byte, RFIXM generates three bytes: an 8086/8088 WAIT instruction, a segment-override byte (001 reg 110B), and the first byte of an 8086/8088 ESCAPE instruction. Note that the segment-override byte is the second byte generated, not as SEGFIX would generate, the first.

(See the discussion of SEGFIX for information on how the assembler determines whether or not a segment-override byte is necessary.)

As an example of the use of RFIXM, consider one of the codemacros for the 8087 instruction FADD:

```
Codemacro   FADDmemop:Mq
 RFIXM      100B, memop
 ModRM      000B, memop
EndM
```

The source statement instruction FADD QUAR [BX] generates the following bytes:

```
1001 1011 00100110 11011100 00001010
```

The first byte is an 8086/8088 WAIT instruction. The second byte is the segment-override byte, specifying ES (reg=00). The third, fourth and fifth bytes identify the floating point instruction as FADD, with a memory operand pointed to by BX, with a displacement of 10. QUAR becomes a QWORD variable at offset 10 from a segment ASSUMED into the ES register only.

RNFIX

The RNFIX directive generates two bytes: an 8086 NOP instruction (1001 0000B) followed by the first byte of an 8086/8088 ESCAPE instruction (11011 XXXB). RNFIX functions like RFIX, except that a NOP instruction is the first byte generated, rather than a WAIT instruction. The format of the RNFIX directive is:

```
RNFIX  formal_ or  number
```

where "formal__or__number" is either the name of a formal parameter with specifier D or an absolute number. The value of "formal or _number" specifies the low-order three bits of the second byte generated.

As an example of the use of RNFIX, consider the codemacro for the 8086 instruction FNCLEX:

```
Codemacro  FNCLEX
 RNFIX     011B
 DB        111 000 10B
EndM
```

The source statement instruction FNCLEX generates the following three bytes:

```
1001 0000 11011011 11100010.
```

RNFIXM

RNFIXM is a directive which generates the same two bytes as RNFIX, but also instructs the assembler to determine whether a segment-override byte is needed to access a given memory location. RNFIXM functions like RFIXM, except that a NOP instruction is the first byte generated, rather than a WAIT instruction. The format of the RNFIXM directive is:

```
    RNFIXM  formal_ or_number,  formal_name
```

where "formal_or_number" is either the name of a formal parameter with specifier D or an absolute number and 'formal_name' is the name of a formal parameter which represents a memory address, that is, its specifier must be 'E', 'M' or 'X'.

If a segment-override byte (001 reg 110B) is needed to address "formal_name," it immediately follows the first byte generated, i.e., the NOP instruction.

As an example of the use of the RNFIXM directive, consider the codemacro for the 8087 instruction FNSAVE:

```
Codemacro   FNSAVE memop:M
 RNfixM     101B, memop
 ModRM      110B, memop
EndM
```

The source statement instruction FNSAVE WORD PTR SS:[BX] generates the following bytes:

```
    1001 0000 00110110 11011 1101 0011 0111
```

Note that the segment-override byte (0011 0110B) follows the NOP instruction (10010000).

RWFIX

The RWFIX directive generates an 8086/8088 WAIT instruction (1001 1011B). The format of this directive is:

```
    RWFIX
```

NOTE

The preceding descriptions assume that the generated code is to be linked with the 8087 chip library (8087.LIB). If the code is linked instead with the 8087 emulator library (E8087.LIB), an emulator instruction is generated instead of an 8087 instruction. The emulator instruction differs from the 8087 instruction in the first two bytes of code. The correct instruction may not be determined until the program is actually linked so the Assembler listing will always show the 8087 instructions.

Using the Dot Operator to Shift Parameters

A special construct allowed as the operand to a DB, DW, or DD, or as an element of the operand to a record initialization, is the shifted formal parameter. The form of this construct is:

```
    formal_name.record  field_name
```

where formal name is the name of a formal whose corresponding operand will be an absolute number; and record_ field _name is the name of a record field. The assembler evaluates this expression when the codemacro is invoked, by right-shifting the operand provided using the shift count defined by the record field.

The example in the 8086 instruction set where this feature is used is the ESC instruction, which permits communication with other devices using the same bus. Given an address, ESC puts that address on the bus; given a register operand, no address is put on the bus. This enables execution of commands from an external device both with or without an associated operand. These commands are represented in the ESC codemacro as numbers between 0 and 63 inclusive. The interpretation of the number is done by the external device.

```
R53  Record RF1:5, RF2:3
R233 Record RF6:2, mid3:3, RF7:3
Codemacro ESC opcode:Db(0,63), addr:E
Segfix  addr
R53     <11011B, opcode.mid3>
ModRM   opcode, addr
EndM
```

The R53 line in the body of the codemacro generates 8 bits as follows: the high-order 5 bits become 11011B, and the low-order 3 bits are filled with the actual parameter supplied as "opcode" shifted right by the shift count of mid3, namely 3.

Example:

Assume that you wish to use ESC with an "opcode" of 39 on an "addr" of MEMWORD, whose offset is 477H in ES, indexed by DI.

```
ESC 39, ES: MEMWORD [DI]
    SEGFIX addr becomes ES: = 0010 0110B
    39 = 00100111B
    opcode.MID3 = (000)00100
    R53<11011B, opcode.mid3> becomes 110 11100
    for [DI],MOD = 10,R/M = 101
```

MODRM opcode,addr puts "opcode" into bits 5, 4, 3 of the modrm byte, with bits 7, 6, 2, 1, 0 filled by the appropriate mod and R/M from "addr." Since opcode is 6 bits and the field is only 3 bits wide, only the low-order 3 are used, namely 111, and the high-order bits (100) are ignored.

Therefore MODRM opcode,addr becomes 1011 1101B followed by the offset of MEMWORD, 0111 0111 0000 0100.

Therefore the full object code for this ESC source line is:

```
0010 0110  (byte 1)
1101 1100  (byte 2)
1011 1101  (byte 3)
0111 0111  (byte 4)
0000 0100  (byte 5)
```

Note that opcode's 6 bits are split between the last 3 bits of byte 2 and bits 5, 4, 3 of byte 3.

PROCLEN

This special operand equals 0 if the current PROC is declared NEAR, and 0FFH if it is declared FAR. Code outside of PROC...ENDP blocks is considered NEAR. The RET codemacros use this operator in creating the correct machine instructions to return from a CALL to a NEAR or FAR procedure:

```
Codemacro RET
R413      <0CH,PROCLEN,3>
Endm
```

Instead of the more familiar DB or DW storage allocation commands, this codemacro makes use of a previously defined record. It is used here the same way a DB would be, but with the initialization given inside angle brackets to show that each field in the record gets its own initial value. You can tell there are at least 3 fields in the record (if this invocation validly matches the definition, i.e., is not an error) because 3 values are given, separated by commas.

Four such records are defined as one of the first acts of the assembler, to be used in defining its instruction set. They are listed in APPENDIX A along with the codemacros for ASM86:

```
R53    Record RF1:5, RF2:3
R323   Record RF3:3, RF4:2, RF5:3
R233   Record RF6:2, Mid3:3, RF7:3
R413   Record RF8:4, RF9:1, RF10:3
```

The last line above, R413, defines an 8 bit record of 3 fields: the high-order 4 bits (7, 6, 5, 4) called RF8, the next (bit 3) called RF9, and the low-order 3 (bits 2, 1, 0) called RF10. (When R413 is used as a storage allocation command, initial values for all fields must be specified within angle brackets because none were specified in the definition.)

In the codemacro for RET, the field RF8 is set to 0CH = 1100, and RF10 is set to 3 = 011. Field RF9, which becomes bit 3 of the allocated record byte, will be 0 if the current PROC (in which the RET appears) is typed NEAR, or it will be 1 if the PROC is typed FAR.

Note that PROCLEN is defined to give 8 bits, all zeros or all ones, but that R413 uses only one bit. The field size determines how many bits are used, and if more are supplied then the high-order bits are ignored beyond the field width.

Matching of Instructions to Codemacros

This section describes what might aptly be termed the heart of the 8086 assembly language. The careful ordering of the chain of codemacro definitions of a given instruction (for example, the ADD instruction) combines with the varied set of typing requirements on the operands to produce a single assembly language instruction mnemonic which represents many hardware instructions.

The algorithm for matching an instruction to a particular codemacro definition is as follows:

1. In pass 1, actual parameters are evaluated. Those containing forward references are treated as a special type, as described in each of the cases below.
2. If any of the actual parameters (when there are more than one) is a register expression without an associated type (e.g., [BX]), or if an implicit reference to the accumulator is made (e.g., "MOV,3"), then the other parameters are checked to see if at least one contains an unambiguous modifier type. Numbers

matching "b" do not suffice; but numbers matching "w," explicitly-given registers, and all typed variables do suffice to distinguish the modifier type. If no such parameter is found, the error message "INSUFFICIENT TYPE INFORMATION TO DETERMINE CORRECT INSTRUCTION" is issued, and no match is attempted. Note that a single, untyped, register expression parameter (as in FSTENV [BX]) is allowed.

3. The chain of codemacro definitions for a given instruction is searched for a match, beginning with the last one defined and working backwards. In order for a definition to match, the number of actual parameters must match the number of formals in the particular definition, and each actual must match the formal in specifier type, modifier (if given in the formal), and range (if given in the formal). The run-down of which actuals match which formals is as follows:

   a. SPECIFIERS.
      Forward references in pass 1 match C,D,E,M,X.
      AX and AL match A,E,R.
      Labels match C.
      Numbers match D.
      Non-indexed variables match E,M,X.
      Indexed variables and register expressions match E,M.
      Registers except segment registers match E,R.
      Segment registers CS,DS,ES,SS match S.
      Floating-point stack element
         (ST, ST(0), ..., ST(7)) match F.
      The floating-point stack top
         (ST, ST(0)) match T.

   b. MODIFIERS.
      The nature of modifier-matching depends on what the matched specifier is.
      For numbers: Numbers between −256 and 255 match "b" only. Other
         numbers match "w" only.
      For labels: NEAR labels with the SAME CS-assume which are in the range
      −126 to +129 from the beginning of the codemacro match "b" only.
         Other NEAR labels with the same CS assume match "w" only.
         NEAR labels with a different CS-assume match no modifier.
         FAR labels match "d".
      For variables: Type BYTE matches "b."
         Type WORD matches "w."
         Type DWORD matches "d."
         Type QWORD matches "q."
         Type TBYTE matches "t."
         Other numeric types match no modifier.
      Forward references match any modifier, except when typing information is attached, with BYTE PTR, SHORT, FAR PTR, etc.
      Index-register expressions without a type associated with them (e.g., [BX]) match either "b" or "w" when used with another operand of type "b" or "w" and matches no modifier for single-operand instructions.

   c. RANGES.
      Range specifiers are legal only for parameters which are numbers or registers (specifiers A, D, R, S). If one specifier follows the formal, the value of the actual must match; if two follow the formal, the value must fall within the inclusive range of the specifiers. For this matching, registers which are passed as actuals assume the following numeric values:

AL:	0
CL:	1
DL:	2
BL:	3
AH:	4
CH:	5
DH:	6

BH:	7
AX:	0
CX:	1
DX:	2
BX:	3
SP:	4
BP:	5
SI:	6
DI:	7
ES:	0
CS:	1
SS:	2
DS:	3

Forward references do not match the formal if there is a range specifier.

4. If a match is found, the number of bytes of object code generated is estimated. Forward-reference variables, unless explicitly overridden, are assumed not to need a segment override byte. ModRMs involving forward references are assumed to require 16-bit displacements, except if the reference has SHORT, in which case an 8-bit displacement is assumed.
5. In pass 2, the search through the codemacro chain starts all over again, starting at the end of the chain and working backwards just as in pass 1. The resolution of codemacro parameters which were forward references in pass 1 might cause a different codemacro to be matched in pass 2.
6. Object code generated by the instruction is issued in pass 2. If the number of bytes output exceeds the pass 1 estimate, an error message is issued and the extra bytes are withheld. The instruction is thus incomplete and the program should not be executed. If the number of bytes is less than the pass 1 estimate, the remaining space is padded with 90H's (NOP; i.e., no operation).

The ADD instruction (like many other instructions) provides an excellent example of codemacro matching. The 11 codemacro definitions of the ADD instruction cover the following cases:

DESTINATION	SOURCE	
1. BYTE MEMORY	IMMEDIATE BYTE	
2. WORD MEMORY	IMMEDIATE BYTE	(not between −128 and 127)
3. WORD MEMORY	IMMEDIATE BYTE	(from −128 to 127)
4. WORD MEMORY	IMMEDIATE WORD	
5. AL	IMMEDIATE BYTE	
6. AX	IMMEDIATE BYTE	
7. AX	IMMEDIATE WORD	
8. MEMORY BYTE OR BYTE-REGISTER	BYTE-REGISTER	
9. MEMORY WORD OR WORD-REGISTER	WORD-REGISTER	
10. BYTE-REGISTER	MEMORY BYTE OR BYTE-REGISTER	
11. WORD-REGISTER	MEMORY WORD OR WORD-REGISTER	

Each of the above English-language phrases is abbreviated in the codemacro definitions into a two-letter specifier-modifier combination. Once you are used to the abbreviations, the codemacros themselves are easier to scan and understand than the above English summary. Here are the first lines of each codemacro described above, in the same order, with an English reminder of its meaning, using EA to represent an effective address expression resolving to either a memory or register reference:

1. CodeMacro ADD dst:Eb, src:Db (TO EA byte FROM data byte)
2. CodeMacro ADD dst:Ew, src:Db (TO EA word FROM large data byte)

3. CodeMacro ADD dst:Ew, src:Db (−128,127) (TO EA word FROM signed data byte)
4. CodeMacro ADD dst:Ew, src:Dw (TO EA word FROM data word)
5. CodeMacro ADD dst:Ab, src:Db (TO AL FROM data byte)
6. CodeMacro ADD dst:Aw, src:Db (TO AX FROM data byte)
7. CodeMacro ADD dst:Aw, src:Dw (TO AX FROM data word)
8. CodeMacro ADD dst:Eb, src:Rb (TO EA byte FROM register byte)
9. CodeMacro ADD dst:Ew, src:Rw (TO EA word FROM register word)
10. CodeMacro ADD dst:Rb, src:Eb (TO register byte FROM EA byte)
11. CodeMacro ADD dst:Rw, src:Ew (TO register word FROM EA word)

The ordering of the codemacros is crucial. For example, the instruction "ADD AX,3" matches not only definition #6, but also definition #2, since as a register, AX qualifies as an Ew as well as an Aw. Since definition #6 produces less object code, it should be selected before definition #2. Hence, it is given later, so that when the assembler searches backwards from #11 up, it comes across #6 first.

Assuming that the following user symbols have been defined with the following attributes:

BYTE_VAR	byte variable
WORD_VAR	word variable
WORD_EXPR	memory-address expression
B_ARRAY	byte variable

The following assembler instructions would match the indicated codemacro definition line above:

```
ADD AX,250        → 6
ADD AX,350        → 7
ADD BX,WORD_EXPR  → 11
ADD BX,DX         → 11
ADD BYTE_VAR,AL   → 8
ADD BYTE_VAR,254  → 1
ADD WORD_VAR,CX   → 9
ADD DH,BARRAY[SI] → 10
ADD CL,BYTE_VAR   → 10
ADD AL,3          → 5
ADD WORD_VAR,35648→ 4
ADD WORD_VAR, OFFSET B_ARRAY → 4
ADD [BX][SI], AH  → 8
ADD [BP],CL       → 8
ADD DX,[DI]       → 11
ADD AX,[SI][BP]   → 11
ADD WORD_VAR,3    → 3
ADD WORD_VAR,255  → 2
```

NOTE

Each codemacro is limited to a maximum of 128 internal bytes, which is reached at approximately 60 bytes of generated object code.

Codemacros

```
; 8Ø86/186 and 8Ø87 Codemacro Definitions

R53    Record  RF1:5,RF2:3
R323   Record  RF3:3,RF4:2,RF5:3
R233   Record  RF6:2,Mid3:3,RF7:3
R413   Record  RF8:4,RF9:1,RF1Ø:3

; 8Ø86/186 Codemacros:

CodeMacro AAA
  DB  37H
EndM

CodeMacro AAD
  DW  ØAD5H
EndM

CodeMacro AAM
  DW  ØAD4H
EndM

CodeMacro AAS
  DB  3FH
EndM

CodeMacro Adc dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  2,dst
  DB  src
EndM

CodeMacro Adc dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  2,dst
  DW  src
EndM

CodeMacro Adc dst:Ew,src:Db(-128,127)
  Segfix  dst
  DB  83H
  ModRM  2,dst
  DB  src
EndM

CodeMacro Adc dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  2,dst
  DW  src
EndM

CodeMacro Adc dst:Ab,src:Db
  DB  14H
  DB  src
EndM

CodeMacro Adc dst:Aw,src:Db
  DB  15H
  DW  src
EndM

CodeMacro Adc dst:Aw,src:Dw
  DB  15H
  DW  src
EndM

CodeMacro Adc dst:Eb,src:Rb
  Segfix  dst
  DB  1ØH
  ModRM  src,dst
EndM
```

```
CodeMacro Adc dst:Ew,src:Rw
  Segfix  dst
  DB  11H
  ModRM  src,dst
EndM

CodeMacro Adc dst:Rb,src:Eb
  Segfix  src
  DB  12H
  ModRM  dst,src
EndM

CodeMacro Adc dst:Rw,src:Ew
  Segfix  src
  DB  13H
  ModRM  dst,src
EndM

CodeMacro Add dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  Ø,dst
  DB  src
EndM

CodeMacro Add dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  Ø,dst
  DW  src
EndM

CodeMacro Add dst:Ew,src:Db(-128,127)
  Segfix  dst
  DB  83H
  ModRM  Ø,dst
  DB  src
EndM

CodeMacro Add dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  Ø,dst
  DW  src
EndM

CodeMacro Add dst:Ab,src:Db
  DB  Ø4H
  DB  src
EndM

CodeMacro Add dst:Aw,src:Db
  DB  Ø5H
  DW  src
EndM

CodeMacro Add dst:Aw,src:Dw
  DB  Ø5H
  DW  src
EndM

CodeMacro Add dst:Eb,src:Rb
  Segfix  dst
  DB  Ø
  ModRM  src,dst
EndM
```

```
CodeMacro Add dst:Ew,src:Rw
  Segfix  dst
  DB  1
  ModRM  src,dst
EndM

CodeMacro Add dst:Rb,src:Eb
  Segfix  src
  DB  2
  ModRM  dst,src
EndM

CodeMacro Add dst:Rw,src:Ew
  Segfix  src
  DB  3
  ModRM  dst,src
EndM

CodeMacro And dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  4,dst
  DB  src
EndM

CodeMacro And dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  4,dst
  DW  src
EndM

CodeMacro And dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  4,dst
  DW  src
EndM

CodeMacro And dst:Ab,src:Db
  DB  24H
  DB  src
EndM

CodeMacro And dst:Aw,src:Db
  DB  25H
  DW  src
EndM

CodeMacro And dst:Aw,src:Dw
  DB  25H
  DW  src
EndM

CodeMacro And dst:Eb,src:Rb
  Segfix  dst
  DB  2ØH
  ModRM  src,dst
EndM

CodeMacro And dst:Ew,src:Rw
  Segfix  dst
  DB  21H
  ModRM  src,dst
EndM

CodeMacro And dst:Rb,src:Eb
  Segfix  src
  DB  22H
  ModRM  dst,src
EndM

CodeMacro And dst:Rw,src:Ew
  Segfix  src
  DB  23H
  ModRM  dst,src
EndM
```

```
  BOUND at end

CodeMacro Call addr:Ew
  Segfix  addr
  DB  ØFFH
  ModRM  2,addr
EndM

CodeMacro Call addr:Ed
  Segfix  addr
  DB  ØFFH
  ModRM  3,addr
EndM

CodeMacro Call addr:Cd
  DB  9AH
  DD  addr
EndM

CodeMacro Call addr:Cb
  DB  ØE8H
  RelW  addr
EndM

CodeMacro Call addr:Cw
  DB  ØE8H
  RelW  addr
EndM

CodeMacro CBW
  DB  98H
EndM

CodeMacro CLC
  DB  ØF8H
EndM

CodeMacro CLD
  DB  ØFCH
EndM

CodeMacro CLI
  DB  ØFAH
EndM

CodeMacro CMC
  DB  ØF5H
EndM

CodeMacro Cmp dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  7,dst
  DB  src
EndM

CodeMacro Cmp dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  7,dst
  DW  src
EndM

CodeMacro Cmp dst:Ew,src:Db(-128,127)
  Segfix  dst
  DB  83H
  ModRM  7,dst
  DB  src
EndM

CodeMacro Cmp dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  7,dst
  DW  src
EndM
```

```
CodeMacro Cmp dst:Ab,src:Db
  DB  3CH
  DB  src
EndM

CodeMacro Cmp dst:Aw,src:Db
  DB  3DH
  DW  src
EndM

CodeMacro Cmp dst:Aw,src:Dw
  DB  3DH
  DW  src
EndM

CodeMacro Cmp dst:Eb,src:Rb
  Segfix  dst
  DB  38H
  ModRM  src,dst
EndM

CodeMacro Cmp dst:Ew,src:Rw
  Segfix  dst
  DB  39H
  ModRM  src,dst
EndM

CodeMacro Cmp dst:Rb,src:Eb
  Segfix  src
  DB  3AH
  ModRM  dst,src
EndM

CodeMacro Cmp dst:Rw,src:Ew
  Segfix  src
  DB  3BH
  ModRM  dst,src
EndM

CodeMacro CmpS SI_ptr:Mb,DI_ptr:Mb
  NoSegfix ES,DI_ptr
  Segfix  SI_ptr
  DB  0A6H
EndM

CodeMacro CmpS SI_ptr:Mw,DI_ptr:Mw
  NoSegfix ES,DI_ptr
  Segfix  SI_ptr
  DB  0A7H
EndM

CodeMacro CmpSB
  DB  0A6H
EndM

CodeMacro CmpSW
  DB  0A7H
EndM

CodeMacro CWD
  DB  99H
EndM

CodeMacro DAA
  DB  027H
EndM

CodeMacro DAS
  DB  02FH
EndM

CodeMacro Dec dst:Eb
  Segfix  dst
  DB  0FEH
  ModRM  1,dst
EndM
```

```
CodeMacro Dec dst:Ew
  Segfix  dst
  DB  0FFH
  ModRM  1,dst
EndM

CodeMacro Dec dst:Rw
  R53  <01001B,dst>
EndM

CodeMacro Div divisor:Eb
  Segfix  divisor
  DB  0F6H
  ModRM  6,divisor
EndM

CodeMacro Div divisor:Ew
  Segfix  divisor
  DB  0F7H
  ModRM  6,divisor
EndM

; ENTER at end

CodeMacro Esc opcode:Db(0,63),addr:Eb
  Segfix  addr
  R53  <11011B,opcode.mid3>
  ModRM  opcode,addr
EndM

CodeMacro Esc opcode:Db(0,63),addr:Ew
  Segfix  addr
  R53  <11011B,opcode.mid3>
  ModRM  opcode,addr
EndM

CodeMacro Esc opcode:Db(0,63),addr:Ed
  Segfix  addr
  R53  <11011B,opcode.mid3>
  ModRM  opcode,addr
EndM

CodeMacro Hlt
  DB  0F4H
EndM

CodeMacro IDiv divisor:Eb
  Segfix  divisor
  DB  0F6H
  ModRM  7,divisor
EndM

CodeMacro IDiv divisor:Ew
  Segfix  divisor
  DB  0F7H
  ModRM  7,divisor
EndM

CodeMacro Imul mplier:Eb
  Segfix  mplier
  DB  0F6H
  ModRM  5,mplier
EndM

CodeMacro Imul mplier:Ew
  Segfix  mplier
  DB  0F7H
  ModRM  5,mplier
EndM

CodeMacro IMUL dst:RW,src1:EW,src2:DB
  Only186
  Segfix src1
  DB  69H
  ModRM  dst,src1
  DW  src2
EndM
```

```
CodeMacro IMUL dst:RW,src1:EW,src2:DB(-128,127)
  Only186
  Segfix src1
  DB  6BH
  ModRM  dst,src1
  DB  src2
EndM

CodeMacro IMUL dst:RW,src1:EW,src2:DW
  Only186
  Segfix src1
  DB  69H
  ModRM  dst,src1
  DW  src2
EndM

CodeMacro IMUL dst:RW,src2:DB
  Only186
  DB  69H
  ModRM  dst,dst
  DW  src2
EndM

CodeMacro IMUL dst:RW,src2:DB(-128,127)
  Only186
  DB  6BH
  ModRM  dst,dst
  DB  src2
EndM

CodeMacro IMUL dst:RW,src2:DW
  Only186
  DB  69H
  ModRM  dst,dst
  DW  src2
EndM

CodeMacro In dst:Ab,port:Db
  DB  0E4H
  DB  port
EndM

CodeMacro In dst:Aw,port:Db
  DB  0E5H
  DB  port
EndM

CodeMacro In dst:Ab,port:Rw(DX)
  DB  0ECH
EndM

CodeMacro In dst:Aw,port:Rw(DX)
  DB  0EDH
EndM

CodeMacro Inc dst:Eb
  Segfix  dst
  DB  0FEH
  ModRM  0,dst
EndM

CodeMacro Inc dst:Ew
  Segfix  dst
  DB  0FFH
  ModRM  0,dst
EndM

CodeMacro Inc dst:Rw
  R53  <01000B,dst>
EndM

; INS,INSB,INSW at end

CodeMacro Int itype:Db
  DB  0CDH
  DB  itype
EndM
```

```
CodeMacro Int itype:Db(3)
  DB  0CCH
EndM

CodeMacro IntO
  DB  0CEH
EndM

CodeMacro Iret
  DB  0CFH
EndM

CodeMacro JA place:Cb
  DB  77H
  RelB  place
EndM

CodeMacro JAE place:Cb
  DB  73H
  RelB  place
EndM

CodeMacro JB place:Cb
  DB  72H
  RelB  place
EndM

CodeMacro JBE place:Cb
  DB  76H
  RelB  place
EndM

  JC  Equ JB

CodeMacro JCXZ place:Cb
  DB  0E3H
  RelB  place
EndM

CodeMacro JE place:Cb
  DB  74H
  RelB  place
EndM

CodeMacro JG place:Cb
  DB  7FH
  RelB  place
EndM

CodeMacro JGE place:Cb
  DB  7DH
  RelB  place
EndM

CodeMacro JL place:Cb
  DB  7CH
  RelB  place
EndM

CodeMacro JLE place:Cb
  Db  7EH
  RelB  place
EndM

CodeMacro Jmp place:Ew
  Segfix  place
  DB  0FFH
  ModRM  4,place
EndM

CodeMacro Jmp place:Md
  Segfix  place
  DB  0FFH
  ModRM  5,place
EndM
```

```
CodeMacro Jmp place:Cd
  DB  0EAH
  DD  place
EndM

CodeMacro Jmp place:Cb
  DB  0EBH
  RelB  place
EndM

CodeMacro Jmp place:Cw
  DB  0E9H
  RelW  place
EndM

  JNA Equ JBE

  JNAE Equ JB

  JNB Equ JAE

  JNBE Equ JA

  JNC  Equ JNB

CodeMacro JNE place:Cb
  DB  75H
  RelB  place
EndM

  JNG Equ JLE

  JNGE Equ JL

  JNL Equ JGE

  JNLE Equ JG

CodeMacro JNO place:Cb
  DB  71H
  RelB  place
EndM

CodeMacro JNP place:Cb
  DB  7BH
  RelB  place
EndM

CodeMacro JNS place:Cb
  DB  79H
  RelB  place
EndM

  JNZ Equ JNE

CodeMacro JO place:Cb
  DB  70H
  RelB  place
EndM

CodeMacro JP place:Cb
  DB  7AH
  RelB  place
EndM

  JPE Equ JP

  JPO Equ JNP

CodeMacro JS place:Cb
  DB  78H
  RelB  place
EndM

  JZ Equ JE
```

```
CodeMacro LAHF
  DB  9FH
EndM

CodeMacro LDS dst:Rw,src:Ed
  Segfix  src
  DB  0C5H
  ModRM  dst,src
EndM

  LEAVE at end

CodeMacro LES dst:Rw,src:Ed
  Segfix  src
  DB  0C4H
  ModRM  dst,src
EndM

CodeMacro LEA dst:Rw,src:M
  DB  8DH
  ModRM  dst,src
EndM

CodeMacro Lock Prefx
  DB  0F0H
EndM

CodeMacro LodS SI_ptr:Mb
  Segfix  SI_ptr
  DB  0ACH
EndM

CodeMacro LodS SI_ptr:Mw
  Segfix  SI_ptr
  DB  0ADH
EndM

CodeMacro LodSB
  DB  0ACH
EndM

CodeMacro LodSW
  DB  0ADH
EndM

CodeMacro Loop place:Cb
  DB  0E2H
  RelB  place
EndM

CodeMacro LoopE place:Cb
  DB  0E1H
  RelB  place
EndM

CodeMacro LoopNE place:Cb
  DB  0E0H
  RelB  place
EndM

  LoopNZ Equ LoopNE

  LoopZ Equ LoopE

CodeMacro Mov dst:Eb,src:Db
  Segfix  dst
  DB  0C6H
  ModRM  0,dst
  DB  src
EndM

CodeMacro Mov dst:Ew,src:Db
  Segfix  dst
  DB  0C7H
  ModRM  0,dst
  DW  src
EndM
```

```
CodeMacro Mov dst:Ew,src:Dw
  Segfix  dst
  DB  ØC7H
  ModRM  Ø,dst
  DW  src
EndM

CodeMacro Mov dst:Rb,src:Db
  R53  <1Ø11ØB,dst>
  DB  src
EndM

CodeMacro Mov dst:Rw,src:Db
  R53  <1Ø111B,dst>
  DW  src
EndM

CodeMacro Mov dst:Rw,src:Dw
  R53  <1Ø111B,dst>
  DW  src
EndM

CodeMacro Mov dst:Eb,src:Rb
  Segfix  dst
  DB  88H
  ModRM  src,dst
EndM

CodeMacro Mov dst:Ew,src:Rw
  Segfix  dst
  DB  89H
  ModRM  src,dst
EndM

CodeMacro Mov dst:Rb,src:Eb
  Segfix  src
  DB  8AH
  ModRM  dst,src
EndM

CodeMacro Mov dst:Rw,src:Ew
  Segfix  src
  DB  8BH
  ModRM  dst,src
EndM

CodeMacro Mov dst:Ew,src:S
  Segfix  dst
  DB  Ø8CH
  ModRM  src,dst
EndM

CodeMacro Mov dst:S(ES),src:Ew
  Segfix  src
  DB  Ø8EH
  ModRM  dst,src
EndM

CodeMacro Mov dst:S(SS,DS),src:Ew
  Segfix  src
  DB  Ø8EH
  ModRM  dst,src
EndM

CodeMacro Mov dst:Ab,src:Xb
  Segfix  src
  DB  ØAØH
  DW  src
EndM

CodeMacro Mov dst:Aw,src:Xw
  Segfix  src
  DB  ØA1H
  DW  src
EndM
```

```
CodeMacro Mov dst:Xb,src:Ab
  Segfix  dst
  DB  ØA2H
  DW  dst
EndM

CodeMacro Mov dst:Xw,src:Aw
  Segfix  dst
  DB  ØA3H
  DW  dst
EndM

CodeMacro MovS SI_ptr:Mb,DI_ptr:Mb
  NoSegfix ES,SI_ptr
  Segfix  DI_ptr
  DB  ØA4H
EndM

CodeMacro MovS SI_ptr:Mw,DI_ptr:Mw
  NoSegfix ES,SI_ptr
  Segfix  DI_ptr
  DB  ØA5H
EndM

CodeMacro MovSB
  DB  ØA4H
EndM

CodeMacro MovSW
  DB  ØA5H
EndM

CodeMacro Mul mplier:Eb
  Segfix  mplier
  DB  ØF6H
  ModRM  4,mplier
EndM

CodeMacro Mul mplier:Ew
  Segfix  mplier
  DB  ØF7H
  ModRM  4,mplier
EndM

CodeMacro Neg dst:Eb
  Segfix  dst
  DB  ØF6H
  ModRM  3,dst
EndM

CodeMacro Neg dst:Ew
  Segfix  dst
  DB  ØF7H
  ModRM  3,dst
EndM

CodeMacro Nil
EndM

CodeMacro Nop
  DB  9ØH
EndM

CodeMacro Not dst:Eb
  Segfix  dst
  DB  ØF6H
  ModRM  2,dst
EndM

CodeMacro Not dst:Ew
  Segfix  dst
  DB  ØF7H
  ModRM  2,dst
EndM
```

```
CodeMacro OR dst:Eb,src:Db
  Segfix  dst
  DB  80H
  ModRM  1,dst
  DB  src
EndM

CodeMacro OR dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  1,dst
  DW  src
EndM

CodeMacro OR dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  1,dst
  DW  src
EndM

CodeMacro OR dst:Ab,src:Db
  DB  0CH
  DB  src
EndM

CodeMacro OR dst:Aw,src:Db
  DB  0DH
  DW  src
EndM

CodeMacro OR dst:Aw,src:Dw
  DB  0DH
  DW  src
EndM

CodeMacro OR dst:Eb,src:Rb
  Segfix  dst
  DB  8
  ModRM  src,dst
EndM

CodeMacro OR dst:Ew,src:Rw
  Segfix  dst
  DB  9
  ModRM  src,dst
EndM

CodeMacro OR dst:Rb,src:Eb
  Segfix  src
  DB  0AH
  ModRM  dst,src
EndM

CodeMacro OR dst:Rw,src:Ew
  Segfix  src
  DB  0BH
  ModRM  dst,src
EndM

CodeMacro Out port:Db,dst:Ab
  DB  0E6H
  DB  port
EndM

CodeMacro Out port:Db,dst:Aw
  DB  0E7H
  DB  port
EndM

CodeMacro Out port:Rw(DX),dst:Ab
  DB  0EEH
EndM

CodeMacro Out port:Rw(DX),dst:Aw
  DB  0EFH
EndM
```

```
; OUTS,OUTSB,OUTSW at end

CodeMacro Pop dst:Ew
  Segfix  dst
  DB  08FH
  ModRM  0,dst
EndM

CodeMacro Pop dst:S(ES)
  R323 <0,dst,7>
EndM

CodeMacro Pop dst:S(SS,DS)
  R323 <0,dst,7>
EndM

CodeMacro Pop dst:Rw
  R53  <01011B,dst>
EndM

; POPA at end


CodeMacro PopF
  DB  9DH
EndM


CodeMacro PUSH src:D
  Only186
  DB  68H
  DW  src
EndM


CodeMacro PUSH src:DB(-128,127)
  Only186
  DB  6AH
  DB  src
EndM


CodeMacro Push src:Ew
  Segfix  src
  DB  0FFH
  ModRM  6,src
EndM


CodeMacro Push src:S
  R323 <0,src,6>
EndM

CodeMacro Push src:Rw
  R53  <01010B,src>
EndM

; PUSHA moved to end

CodeMacro PushF
  DB  9CH
EndM

CodeMacro RCL dst:Eb,count:D(0,31)
  Only186
  Segfix dst
  DB  0C0H
  ModRM  2,dst
  DB  count
EndM

CodeMacro RCL dst:Eb,count:Db(1)
  Segfix  dst
  DB  0D0H
  ModRM  2,dst
EndM
```

```
CodeMacro RCL dst:Ew,count:D(0,31)
  Only186
  Segfix dst
  DB  0C1H
  ModRM  2,dst
  DB  count
EndM

CodeMacro RCL dst:Ew,count:Db(1)
  Segfix  dst
  DB  0D1H
  ModRM  2,dst
EndM

CodeMacro RCL dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  0D2H
  ModRM  2,dst
EndM

CodeMacro RCL dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  0D3H
  ModRM  2,dst
EndM

CodeMacro RCR dst:Eb,count:D(0,31)
  Only186
  Segfix dst
  DB  0C0H
  ModRM  3,dst
  DB  count
EndM

CodeMacro RCR dst:Eb,count:Db(1)
  Segfix  dst
  DB  0D0H
  ModRM  3,dst
EndM

CodeMacro RCR dst:EW,count:D(0,31)
  Only186
  Segfix dst
  DB  0C1H
  ModRM  3,dst
  DB  count
EndM

CodeMacro RCR dst:Ew,count:Db(1)
  Segfix  dst
  DB  0D1H
  ModRM  3,dst
EndM

CodeMacro RCR dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  0D2H
  ModRM  3,dst
EndM

CodeMacro RCR dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  0D3H
  ModRM  3,dst
EndM

CodeMacro Rep Prefx
  DB  0F3H
EndM

CodeMacro RepE Prefx
  DB  0F3H
EndM

CodeMacro RepNE Prefx
  DB  0F2H
EndM
```

```
  RepNZ Equ RepNE

  RepZ Equ RepE

 CodeMacro Ret src:Db
   R413  <0CH,Proclen,2>
   DW  src
 EndM

CodeMacro Ret src:Dw
  R413  <0CH,Proclen,2>
  DW  src
EndM

CodeMacro Ret
  R413  <0CH,Proclen,3>
EndM

CodeMacro ROL dst:Eb,count:D(0,31)
  Only186
  Segfix dst
  DB  0C0H
  ModRM  0,dst
  DB  count
EndM

CodeMacro ROL dst:Eb,count:Db(1)
  Segfix  dst
  DB  0D0H
  ModRM  0,dst
EndM

CodeMacro ROL dst:Ew,count:D(0,31)
  Only186
  Segfix dst
  DB  0C1H
  ModRM  0,dst
  DB  count
EndM

CodeMacro ROL dst:Ew,count:Db(1)
  Segfix  dst
  DB  0D1H
  ModRM  0,dst
EndM

CodeMacro ROL dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  0D2H
  ModRM  0,dst
EndM

CodeMacro ROL dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  0D3H
  ModRM  0,dst
EndM

CodeMacro ROR dst:Eb,count:D(0,31)
  Only186
  Segfix dst
  DB  0C0H
  ModRM  1,dst
  DB  count
EndM

CodeMacro ROR dst:Eb,count:Db(1)
  Segfix  dst
  DB  0D0H
  ModRM  1,dst
EndM
```

```
CodeMacro ROR dst:Ew,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØC1H
  ModRM  1,dst
  DB  count
EndM

CodeMacro ROR dst:Ew,count:Db(1)
  Segfix  dst
  DB  ØD1H
  ModRM  1,dst
EndM

CodeMacro ROR dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  ØD2H
  ModRM  1,dst
EndM

CodeMacro ROR dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  ØD3H
  ModRM  1,dst
EndM

CodeMacro SAHF
  DB  9EH
EndM

CodeMacro SAL dst:Eb,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØCØH
  ModRM  4,dst
  DB  count
EndM

CodeMacro SAL dst:Eb,count:Db(1)
  Segfix  dst
  DB  ØDØH
  ModRM  4,dst
EndM

CodeMacro SAL dst:Ew,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØC1H
  ModRM  4,dst
  DB  count
EndM

CodeMacro SAL dst:Ew,count:Db(1)
  Segfix  dst
  DB  ØD1H
  ModRM  4,dst
EndM

CodeMacro SAL dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  ØD2H
  ModRM  4,dst
EndM

CodeMacro SAL dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  ØD3H
  ModRM  4,dst
EndM

CodeMacro SAR dst:Eb,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØCØH
  ModRM  7,dst
  DB  count
EndM
```

```
CodeMacro SAR dst:Eb,count:Db(1)
  Segfix  dst
  DB  ØDØH
  ModRM  7,dst
EndM

CodeMacro SAR dst:Ew,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØC1H
  ModRM  7,dst
  DB  count
EndM

CodeMacro SAR dst:Ew,count:Db(1)
  Segfix  dst
  DB  ØD1H
  ModRM  7,dst
EndM

CodeMacro SAR dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  ØD2H
  ModRM  7,dst
EndM

CodeMacro SAR dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  ØD3H
  ModRM  7,dst
EndM

CodeMacro Sbb dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  3,dst
  DB  src
EndM

CodeMacro Sbb dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  3,dst
  DW  src
EndM

CodeMacro Sbb dst:Ew,src:Db(-128,127)
  Segfix  dst
  DB  83H
  ModRM  3,dst
  DB  src
EndM

CodeMacro Sbb dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  3,dst
  DW  src
EndM

CodeMacro Sbb dst:Ab,src:Db
  DB  1CH
  DB  src
EndM

CodeMacro Sbb dst:Aw,src:Db
  DB  1DH
  DW  src
EndM

CodeMacro Sbb dst:Aw,src:Dw
  DB  1DH
  DW  src
EndM
```

```
CodeMacro Sbb dst:Eb,src:Rb
  Segfix  dst
  DB  18H
  ModRM  src,dst
EndM

CodeMacro Sbb dst:Ew,src:Rw
  Segfix  dst
  DB  19H
  ModRM  src,dst
EndM

CodeMacro Sbb dst:Rb,src:Eb
  Segfix  src
  DB  1AH
  ModRM  dst,src
EndM

CodeMacro Sbb dst:Rw,src:Ew
  Segfix  src
  DB  1BH
  ModRM  dst,src
EndM

CodeMacro ScaS DI_ptr:Mb
  NoSegfix ES,DI_ptr
  DB  ØAEH
EndM

CodeMacro ScaS DI_ptr:Mw
  NoSegfix ES,DI_ptr
  DB  ØAFH
EndM

CodeMacro ScaSB
  DB  ØAEH
EndM

CodeMacro ScaSW
  DB  ØAFH
EndM

  SHL Equ SAL

CodeMacro SHR dst:Eb,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØCØH
  ModRM  5,dst
  DB  count
EndM

CodeMacro SHR dst:Eb,count:Db(1)
  Segfix  dst
  DB  ØDØH
  ModRM  5,dst
EndM

CodeMacro SHR dst:Ew,count:D(Ø,31)
  Only186
  Segfix dst
  DB  ØC1H
  ModRM  5,dst
  DB  count
EndM

CodeMacro SHR dst:Ew,count:Db(1)
  Segfix  dst
  DB  ØD1H
  ModRM  5,dst
EndM

CodeMacro SHR dst:Eb,count:Rb(CL)
  Segfix  dst
  DB  ØD2H
  ModRM  5,dst
EndM
```

```
CodeMacro SHR dst:Ew,count:Rb(CL)
  Segfix  dst
  DB  ØD3H
  ModRM  5,dst
EndM

CodeMacro STC
  DB  ØF9H
EndM

CodeMacro STD
  DB  ØFDH
EndM

CodeMacro STI
  DB  ØFBH
EndM

CodeMacro StoS DI_ptr:Mb
  NoSegfix ES,DI_ptr
  DB  ØAAH
EndM

CodeMacro StoS DI_ptr:Mw
  NoSegfix ES,DI_ptr
  DB  ØABH
EndM

CodeMacro StoSB
  DB  ØAAH
EndM

CodeMacro StoSW
  DB  ØABH
EndM

CodeMacro Sub dst:Eb,src:Db
  Segfix  dst
  DB  80H
  ModRM  5,dst
  DB  src
EndM

CodeMacro Sub dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  5,dst
  DW  src
EndM

CodeMacro Sub dst:Ew,src:Db(-128,127)
  Segfix  dst
  DB  83H
  ModRM  5,dst
  DB  src
EndM

CodeMacro Sub dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  5,dst
  DW  src
EndM

CodeMacro Sub dst:Ab,src:Db
  DB  2CH
  DB  src
EndM

CodeMacro Sub dst:Aw,src:Db
  DB  2DH
  DW  src
EndM
```

```
CodeMacro Sub dst:Aw,src:Dw
  DB  2DH
  DW  src
EndM

CodeMacro Sub dst:Eb,src:Rb
  Segfix  dst
  DB  28H
  ModRM  src,dst
EndM

CodeMacro Sub dst:Ew,src:Rw
  Segfix  dst
  DB  29H
  ModRM  src,dst
EndM

CodeMacro Sub dst:Rb,src:Eb
  Segfix  src
  DB  2AH
  ModRM  dst,src
EndM

CodeMacro Sub dst:Rw,src:Ew
  Segfix  src
  DB  2BH
  ModRM  dst,src
EndM

CodeMacro Test dst:Eb,src:Db
  Segfix  dst
  DB  ØF6H
  ModRM  Ø,dst
  DB  src
EndM

CodeMacro Test dst:Ew,src:Db
  Segfix  dst
  DB  ØF7H
  ModRM  Ø,dst
  DW  src
EndM

CodeMacro Test dst:Ew,src:Dw
  Segfix  dst
  DB  ØF7H
  ModRM  Ø,dst
  DW  src
EndM

CodeMacro Test dst:Ab,src:Db
  DB  ØABH
  DB  src
EndM

CodeMacro Test dst:Aw,src:Db
  DB  ØA9H
  DW  src
EndM

CodeMacro Test dst:Aw,src:Dw
  DB  ØA9H
  DW  src
EndM

CodeMacro Test dst:Eb,src:Rb
  Segfix  dst
  DB  84H
  ModRM  src,dst
EndM

CodeMacro Test dst:Ew,src:Rw
  Segfix  dst
  DB  85H
  ModRM  src,dst
EndM
```

```
CodeMacro Test dst:Rb,src:Eb
  Segfix  src
  DB  84H
  ModRM  dst,src
EndM

CodeMacro Test dst:Rw,src:Ew
  Segfix  src
  DB  85H
  ModRM  dst,src
EndM

CodeMacro Wait
  DB  Ø9BH
EndM

CodeMacro Xchg dst:Eb,src:Rb
  Segfix  dst
  DB  86H
  ModRM  src,dst
EndM

CodeMacro Xchg dst:Ew,src:Rw
  Segfix  dst
  DB  87H
  ModRM  src,dst
EndM

CodeMacro Xchg dst:Rb,src:Eb
  Segfix  src
  DB  86H
  ModRM  dst,src
EndM

CodeMacro Xchg dst:Rw,src:Ew
  Segfix  src
  DB  87H
  ModRM  dst,src
EndM

CodeMacro Xchg dst:Rw,src:Aw
  R53  <1ØØ1ØB,dst>
EndM

CodeMacro Xchg dst:Aw,src:Rw
  R53  <1ØØ1ØB,src>
EndM

CodeMacro Xlat table:Mb
  Segfix  table
  DB  ØD7H
EndM

CodeMacro XlatB
  DB  ØD7H
EndM

CodeMacro Xor dst:Eb,src:Db
  Segfix  dst
  DB  8ØH
  ModRM  6,dst
  DB  src
EndM

CodeMacro Xor dst:Ew,src:Db
  Segfix  dst
  DB  81H
  ModRM  6,dst
  DW  src
EndM

CodeMacro Xor dst:Ew,src:Dw
  Segfix  dst
  DB  81H
  ModRM  6,dst
  DW  src
EndM
```

```
CodeMacro Xor dst:Ab,src:Db
  DB  34H
  DB  src
EndM

CodeMacro Xor dst:Aw,src:Db
  DB  35H
  DW  src
EndM

CodeMacro Xor dst:Aw,src:Dw
  DB  35H
  DW  src
EndM

CodeMacro Xor dst:Eb,src:Rb
  Segfix  dst
  DB  30H
  ModRM  src,dst
EndM

CodeMacro Xor dst:Ew,src:Rw
  Segfix  dst
  DB  31H
  ModRM  src,dst
EndM

CodeMacro Xor dst:Rb,src:Eb
  Segfix  src
  DB  32H
  ModRM  dst,src
EndM

CodeMacro Xor dst:Rw,src:Ew
  Segfix  src
  DB  33H
  ModRM  dst,src
EndM


; 8087 Codemacros:

CodeMacro  F2XM1
  Rfix  001B
  DB  11110000B
EndM

CodeMacro  FABS
  Rfix  001B
  DB  11100001B
EndM

CodeMacro  FADD memop:Md
  RfixM  000B,memop
  ModRM  000B,memop
EndM

CodeMacro  FADD memop:Mq
  RfixM  100B,memop
  ModRM  000B,memop
EndM

CodeMacro  FADD  dst:T,src:F
  Rfix  000B
  R233  <11B,000B,src>
EndM

CodeMacro  FADD  dst:F,src:T
  Rfix  100B
  R233  <11B,000B,dst>
EndM

CodeMacro  FADD
  Rfix  110B
  DB  11000001B
EndM
```

```
CodeMacro  FADDP  dst:F,src:T
  Rfix  110B
  R233  <11B,000B,dst>
EndM

CodeMacro  FBLD memop:Mt
  RfixM  111B,memop
  ModRM  100B,memop
EndM

CodeMacro  FBSTP memop:Mt
  RfixM  111B,memop
  ModRM  110B,memop
EndM

CodeMacro  FCHS
  Rfix  001B
  DB  11100000B
EndM

CodeMacro  FCLEX
  Rfix  011B
  DB  11100010B
EndM

CodeMacro  FCOM memop:Md
  RfixM  000B,memop
  ModRM  010B,memop
EndM

CodeMacro  FCOM memop:Mq
  RfixM  100B,memop
  ModRM  010B,memop
EndM

CodeMacro  FCOM  fpst:F
  Rfix  000B
  R233  <11B,010B,fpst>
EndM

CodeMacro  FCOM
  Rfix  000B
  DB  11010001B
EndM

CodeMacro  FCOMP memop:Md
  RfixM  000B,memop
  ModRM  011B,memop
EndM

CodeMacro  FCOMP memop:Mq
  RfixM  100B,memop
  ModRM  011B,memop
EndM

CodeMacro  FCOMP  fpst:F
  Rfix  000B
  R233  <11B,011B,fpst>
EndM

CodeMacro  FCOMP
  Rfix  000B
  DB  11011001B
EndM

CodeMacro  FCOMPP
  Rfix  110B
  DB  11011001B
EndM

CodeMacro  FDECSTP
  Rfix  001B
  DB  11110110B
EndM
```

```
CodeMacro  FDISI
  Rfix  011B
  DB  11100001B
EndM

CodeMacro  FDIV memop:Md
  RfixM  000B,memop
  ModRM  110B,memop
EndM

CodeMacro  FDIV memop:Mq
  RfixM  100B,memop
  ModRM  110B,memop
EndM

CodeMacro  FDIV  dst:T,src:F
  Rfix  000B
  R233  <11B,110B,src>
EndM

CodeMacro  FDIV  dst:F,src:T
  Rfix  100B
  R233  <11B,111B,dst>
EndM

CodeMacro  FDIV
  Rfix  110B
  DB  11111001B
EndM

CodeMacro  FDIVP  dst:F,src:T
  Rfix  110B
  R233  <11B,111B,dst>
EndM

CodeMacro  FDIVR memop:Md
  RfixM  000B,memop
  ModRM  111B,memop
EndM

CodeMacro  FDIVR memop:Mq
  RfixM  100B,memop
  ModRM  111B,memop
EndM

CodeMacro  FDIVR  dst:T,src:F
  Rfix  000B
  R233  <11B,111B,src>
EndM

CodeMacro  FDIVR  dst:F,src:T
  Rfix  100B
  R233  <11B,110B,dst>
EndM

CodeMacro  FDIVR
  Rfix  110B
  DB  11110001B
EndM

CodeMacro  FDIVRP  dst:F,src:T
  Rfix  110B
  R233  <11B,110B,dst>
EndM

CodeMacro  FENI
  Rfix  011B
  DB  11100000B
EndM

CodeMacro  FFREE  fpst:F
  Rfix  101B
  R233  <11B,000B,fpst>
EndM
```

```
CodeMacro  FIADD memop:Mw
  RfixM  110B,memop
  ModRM  000B,memop
EndM

CodeMacro  FIADD memop:Md
  RfixM  010B,memop
  ModRM  000B,memop
EndM

CodeMacro  FICOM memop:Mw
  RfixM  110B,memop
  ModRM  010B,memop
EndM

CodeMacro  FICOM memop:Md
  RfixM  010B,memop
  ModRM  010B,memop
EndM

CodeMacro  FICOMP memop:Mw
  RfixM  110B,memop
  ModRM  011B,memop
EndM

CodeMacro  FICOMP memop:Md
  RfixM  010B,memop
  ModRM  011B,memop
EndM

CodeMacro  FIDIV memop:Mw
  RfixM  110B,memop
  ModRM  110B,memop
EndM

CodeMacro  FIDIV memop:Md
  RfixM  010B,memop
  ModRM  110B,memop
EndM

CodeMacro  FIDIVR memop:Mw
  RfixM  110B,memop
  ModRM  111B,memop
EndM

CodeMacro  FIDIVR memop:Md
  RfixM  010B,memop
  ModRM  111B,memop
EndM

CodeMacro  FILD memop:Mw
  RfixM  111B,memop
  ModRM  000B,memop
EndM

CodeMacro  FILD memop:Md
  RfixM  011B,memop
  ModRM  000B,memop
EndM

CodeMacro  FILD memop:Mq
  RfixM  111B,memop
  ModRM  101B,memop
EndM

CodeMacro  FIMUL memop:Mw
  RfixM  110B,memop
  ModRM  001B,memop
EndM

CodeMacro  FIMUL memop:Md
  RfixM  010B,memop
  ModRM  001B,memop
EndM
```

```
CodeMacro  FINCSTP
  Rfix  ØØ1B
  DB  11110111B
EndM

CodeMacro  FINIT
  Rfix  Ø11B
  DB  11100011B
EndM

CodeMacro  FIST memop:Mw
  RfixM  111B,memop
  ModRM  Ø1ØB,memop
EndM

CodeMacro  FIST memop:Md
  RfixM  Ø11B,memop
  ModRM  Ø1ØB,memop
EndM

CodeMacro  FISTP memop:Mw
  RfixM  111B,memop
  ModRM  Ø11B,memop
EndM

CodeMacro  FISTP memop:Md
  RfixM  Ø11B,memop
  ModRM  Ø11B,memop
EndM

CodeMacro  FISTP memop:Mq
  RfixM  111B,memop
  ModRM  111B,memop
EndM

CodeMacro  FISUB memop:Mw
  RfixM  11ØB,memop
  ModRM  1ØØB,memop
EndM

CodeMacro  FISUB memop:Md
  RfixM  Ø1ØB,memop
  ModRM  1ØØB,memop
EndM

CodeMacro  FISUBR memop:Mw
  RfixM  11ØB,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FISUBR memop:Md
  RfixM  Ø1ØB,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FLD memop:Md
  RfixM  ØØ1B,memop
  ModRM  ØØØB,memop
EndM

CodeMacro  FLD memop:Mq
  RfixM  1Ø1B,memop
  ModRM  ØØØB,memop
EndM

  CodeMacro  FLD memop:Mt
    RfixM  Ø11B,memop
    ModRM  1Ø1B,memop
  EndM

  CodeMacro  FLD  fpst:F
    Rfix  ØØ1B
    R233  <11B,ØØØB,fpst>

CodeMacro  FISTP memop:Mw
  RfixM  111B,memop
  ModRM  Ø11B,memop
EndM
```

```
CodeMacro  FISTP memop:Md
  RfixM  Ø11B,memop
  ModRM  Ø11B,memop
EndM

CodeMacro  FISTP memop:Mq
  RfixM  111B,memop
  ModRM  111B,memop
EndM

CodeMacro  FISUB memop:Mw
  RfixM  11ØB,memop
  ModRM  1ØØB,memop
EndM

CodeMacro  FISUB memop:Md
  RfixM  Ø1ØB,memop
  ModRM  1ØØB,memop
EndM

CodeMacro  FISUBR memop:Mw
  RfixM  11ØB,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FISUBR memop:Md
  RfixM  Ø1ØB,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FLD memop:Md
  RfixM  ØØ1B,memop
  ModRM  ØØØB,memop
EndM

CodeMacro  FLD memop:Mq
  RfixM  1Ø1B,memop
  ModRM  ØØØB,memop
EndM

CodeMacro  FLD memop:Mt
  RfixM  Ø11B,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FLD  fpst:F
  Rfix  ØØ1B
  R233  <11B,ØØØB,fpst>
EndM

CodeMacro  FLD1
  Rfix  ØØ1B
  DB  11101000B
EndM

CodeMacro  FLDCW memop:M
  RfixM  ØØ1B,memop
  ModRM  1Ø1B,memop
EndM

CodeMacro  FLDENV memop:M
  RfixM  ØØ1B,memop
  ModRM  1ØØB,memop
EndM

CodeMacro  FLDL2E
  Rfix  ØØ1B
  DB  111Ø1Ø1ØB
EndM

CodeMacro  FLDL2T
  Rfix  ØØ1B
  DB  111Ø1ØØ1B
EndM
```

```
CodeMacro  FLDLG2
  Rfix  001B
  DB  11101100B
EndM

CodeMacro  FLDLN2
  Rfix  001B
  DB  11101101B
EndM

CodeMacro  FLDPI
  Rfix  001B
  DB  11101011B
EndM

CodeMacro  FLDZ
  Rfix  001B
  DB  11101110B
EndM

CodeMacro  FMUL memop:Md
  RfixM  000B,memop
  ModRM  001B,memop
EndM

CodeMacro  FMUL memop:Mq
  RfixM  100B,memop
  ModRM  001B,memop
EndM

CodeMacro  FMUL  dst:T,src:F
  Rfix  000B
  R233  <11B,001B,src>
EndM

CodeMacro  FMUL  dst:F,src:T
  Rfix  100B
  R233  <11B,001B,dst>
EndM

CodeMacro  FMUL
  Rfix  110B
  DB  11001001B
EndM

CodeMacro  FMULP  dst:F,src:T
  Rfix  110B
  R233  <11B,001B,dst>
EndM

CodeMacro  FNCLEX
  RNfix  011B
  DB  11100010B
EndM

CodeMacro  FNDISI
  RNfix  011B
  DB  11100001B
EndM

CodeMacro  FNENI
  RNfix  011B
  DB  11100000B
EndM

CodeMacro  FNINIT
  RNfix  011B
  DB  11100011B
EndM

CodeMacro  FNOP
  Rfix  001B
  DB  11010000B
EndM
```

```
CodeMacro  FNSAVE memop:M
  RNfixM  101B,memop
  ModRM  110B,memop
EndM

CodeMacro  FNSTCW memop:M
  RNfixM  001B,memop
  ModRM  111B,memop
EndM

CodeMacro  FNSTENV memop:M
  RNfixM  001B,memop
  ModRM  110B,memop
EndM

CodeMacro  FNSTSW memop:M
  RNfixM  101B,memop
  ModRM  111B,memop
EndM

CodeMacro  FPATAN
  Rfix  001B
  DB  11110011B
EndM

CodeMacro  FPREM
  Rfix  001B
  DB  11111000B
EndM

CodeMacro  FPTAN
  Rfix  001B
  DB  11110010B
EndM

CodeMacro  FRNDINT
  Rfix  001B
  DB  11111100B
EndM

CodeMacro  FRSTOR memop:M
  RfixM  101B,memop
  ModRM  100B,memop
EndM

CodeMacro  FSAVE memop:M
  RfixM  101B,memop
  ModRM  110B,memop
EndM

CodeMacro  FSCALE
  Rfix  001B
  DB  11111101B
EndM

CodeMacro  FSQRT
  Rfix  001B
  DB  11111010B
EndM

CodeMacro  FST memop:Md
  RfixM  001B,memop
  ModRM  010B,memop
EndM

CodeMacro  FST memop:Mq
  RfixM  101B,memop
  ModRM  010B,memop
EndM

CodeMacro  FST  fost:F
  Rfix  101B
  R233  <11B,010B,fost>
EndM
```

```
CodeMacro  FSTCW memop:M
  RfixM  001B,memop
  ModRM  111B,memop
EndM

CodeMacro  FSTENV memop:M
  RfixM  001B,memop
  ModRM  110B,memop
EndM

CodeMacro  FSTP memop:Md
  RfixM  001B,memop
  ModRM  011B,memop
EndM

CodeMacro  FSTP memop:Mq
  RfixM  101B,memop
  ModRM  011B,memop
EndM

CodeMacro  FSTP memop:Mt
  RfixM  011B,memop
  ModRM  111B,memop
EndM

CodeMacro  FSTP  fpst:F
  Rfix  101B
  R233  <11B,011B,fpst>
EndM

CodeMacro  FSTSW memop:M
  RfixM  101B,memop
  ModRM  111B,memop
EndM

CodeMacro  FSUB memop:Md
  RfixM  000B,memop
  ModRM  100B,memop
EndM

CodeMacro  FSUB memop:Mq
  RfixM  100B,memop
  ModRM  100B,memop
EndM

CodeMacro  FSUB  dst:T,src:F
  Rfix  000B
  R233  <11B,100B,src>
EndM

CodeMacro  FSUB  dst:F,src:T
  Rfix  100B
  R233  <11B,101B,dst>
EndM

CodeMacro  FSUB
  Rfix  110B
  DB  11101001B
EndM

CodeMacro  FSUBP  dst:F,src:T
  Rfix  110B
  R233  <11B,101B,dst>
EndM

CodeMacro  FSUBR memop:Md
  RfixM  000B,memop
  ModRM  101B,memop
EndM

CodeMacro  FSUBR memop:Mq
  RfixM  100B,memop
  ModRM  101B,memop
EndM
```

```
CodeMacro  FSUBR  dst:T,src:F
  Rfix  000B
  R233  <11B,101B,src>
EndM

CodeMacro  FSUBR  dst:F,src:T
  Rfix  100B
  R233  <11B,100B,dst>
EndM

CodeMacro  FSUBR
  Rfix  110B
  DB  11100001B
EndM

CodeMacro  FSUBRP  dst:F,src:T
  Rfix  110B
  R233  <11B,100B,dst>
EndM

CodeMacro  FTST
  Rfix  001B
  DB  11100100B
EndM

CodeMacro  FWAIT
  RWfix
EndM

CodeMacro  FXAM
  Rfix  001B
  DB  11100101B
EndM

CodeMacro  FXCH  fpst:F
  Rfix  001B
  R233  <11B,001B,fpst>
EndM

CodeMacro  FXCH
  Rfix  001B
  DB  11001001B
EndM

CodeMacro  FXTRACT
  Rfix  001B
  DB  11110100B
EndM

CodeMacro  FYL2X
  Rfix  001B
  DB  11110001B
EndM

CodeMacro  FYL2XP1
  Rfix  001B
  DB  11111001B
EndM

CodeMacro  BOUND  indx:RW,bptr:MW
  Only186
  Segfix  bptr
  DB    62H
  ModRM  indx,bptr
EndM

CodeMacro  BOUND  indx:RW,bptr:MD
  Only186
  Segfix  bptr
  DB    62H
  ModRM  indx,bptr
EndM
```

```
CodeMacro  ENTER disp:D(Ø,ØFFFFH),level:D(Ø,255)
  Only186
  DB     ØC8H
  DW     disp
  DB     level
EndM

CodeMacro  INS  di_ptr:EB,port:RW(DX)
  Only186
  NoSegfix  ES:di_ptr
  DB     6CH
EndM

CodeMacro  INS  di_ptr:EW,port:RW(DX)
  Only186
  NoSegfix  ES:di_ptr
  DB     6DH
EndM

CodeMacro  INSB
  Only186
  DB     6CH
EndM

CodeMacro  INSW
  Only186
  DB     6DH
EndM

CodeMacro  LEAVE
  Only186
  DB     ØC9H
EndM

CodeMacro  OUTS  port:RW(DX),si_ptr:EB
  Only186
  Segfix  si_ptr
  DB     6EH
EndM
```

```
CodeMacro  OUTS  port:RW(DX),si_ptr:EW
  Only186
  Segfix  si_ptr
  DB     6FH
EndM

CodeMacro  OUTSB
  Only186
  DB     6EH
EndM

CodeMacro  OUTSW
  Only186
  DB     6FH
EndM

CodeMacro  POPA
  Only186
  DB     61H
EndM

CodeMacro  PUSHA
  Only186
  DB     6ØH
EndM

  Purge R53,R323,R233,R413
  Purge RF1,RF2,RF3,RF4,RF5
  Purge RF6,RF7,RF8,RF9
  Purge RF1Ø,Mid3

END
```

Flag Operations B

FLAG REGISTERS

Flags are used to distinguish or denote certain results of data manipulation. The 8086 provides the four basic mathematical operations (+, -, *, /) in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard two's complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer instructions in Chapter 6).

Adjustment operations are provided to allow arithmetic to be performed directly on unpacked decimal digits or on packed decimal representations, and the auxiliary flag (AF) facilitates these adjustments.

Flags also aid in interpreting certain operations which could destroy one of their operands. For example, a compare is actually a subtract operation; a zero result indicates that the operands are equal. Since it is unacceptable for the compare to destroy either of the operands, the processor includes several work registers reserved for its own use in such operations. The programmer cannot access these registers. They are used for internal data transfers and for holding temporary values in destructive operations, whose results are reflected in the flags.

Your program can test the setting of five of these flags (carry, sign, zero, overflow, and parity) using one of the conditional jump instructions. This allows you to alter the flow of program execution based on the outcome of a previous operation. the auxiliary carry flag is reserved for the use of the ASCII and decimal adjust instructions, as will be explained later in this section.

It is important for you to know which flags are set by a particular instruction. Assume, for example, that your program is to test the parity of an input byte and then execute one instruction sequence if parity is even, a different instruction sequence if parity is odd. Coding a JPE (jump if parity is even) or JPO (jump if parity is odd) instruction immediately following the IN (input) instruction would produce false results, since the IN instruction does not affect the condition flags. The jump conditionally executed by your program would reflect the outcome of some previous operation unrelated to the IN instructions.

For the operation to work correctly, you must include some instruction that alters the parity flag after the IN instruction, but before the jump instruction. For example, you can add zero to the input byte in the accumulator. This sets the parity flag without altering the data in the accumulator.

In other cases, you will want to set a flag though there may be a number of intervening instructions before you test it. In these cases, you must check the operation of the intervening instructions to be sure that they do not affect the desired flag.

The flags set by each instruction are detailed in the individual instructions in Chapter 6 of this manual.

Details of Flag Usage. Six flag registers are set or cleared by most arithmetic operations to reflect certain properties of the result of the operation. They follow these rules below, where "set" means set to 1 and "clear" means cler to 0. Further discussion of each of these flags follows the concise description.

CF is set if the operation resulted in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; otherwise CF is cleared.

AF is set if the operation resulted in a carry out of (from addition) or borrow into (from subtraction) the low-order four bits of the result; otherwise AF is cleared.

ZF is set if the result of the operation is zero; otherwise ZF is cleared.

SF is set if the high-order bit of the result is set; otherwise SF is cleared.

PF is set if the modulo 2 sum of the low-order eight bits of the result of the operation is 0 (even parity); otherwise PF is cleared (odd parity).

OF is set if the signed operation resulted in an overflow, i.e., the operation resulted in a carry into the high-order bit of the result but not a carry out of the high-order bit, or vice versa; otherwise OF is cleared.

Carry Flag. As its name implies, the carry flag is commonly used to indicate whether an addition causes a "carry" into the next higher order digit. (However, the increment and decrement instructions (INC, DEC) do not affect CF.) The carry flag is also used as a "borrow" flag in subtractions.

The logical AND, OR, and XOR instructions also affect CF. These instructions set or reset particular bits of their destination (register or memory). See the descriptions of the logic instruction in Chapter 6.

The rotate and shift instructions move the contents of the operand (registers or memory) one or more positions to the left or right. They treat the carry flag as though it were an extra bit of the operand. The original value in CF is only preserved by RCL and RCR. Otherwise it is simply replaced with the next bit rotated out of the source, i.e., the high-order bit if an RCL is used, the low-order bit if RCR.

Example:

Addition of two one-byte numbers can produce a carry out of the high-order bit:

Bit Number:	**7654**	**3210**
AEH -	1010	1110B
+ 74H -	0111	0100B
122H	0010	0010B - 22H ;carry flag - 1

An addition that causes a carry out of the high-order bit of the destination sets the flag to 1; an addition that does not cause a carry resets the flag to zero.

Sign Flag. The high-order bit of the result of operations on registers or memory can be interpreted as a sign. Instructions that affect the sign flag set the flag equal to this high-order bit. A zero indicates a positive value; a one indicates a negative value. This value is duplicated in the sign flag so that conditional jump instructions can test for positive and negative values. The high order bit for byte value is bit 7; for word values it is bit 15.

Zero Flag. Certain instructions set the zero flag to one. This indicates that the last operation to affect ZF resulted in all zeros in the destination (register or memory). If that result was other than zero, then ZF is reset to 0. A result that has a carry and a zero result sets both flags, as shown below:

```
  10100111
+01011001
 ___________________________
  00000000      Carry Flag = 1
                Zero Flag = 1
                meaning yes, zero
```

Parity Flag. Parity is determined by counting the number of one bits set in the low order 8 bits of the destination of the last operation to affect PF. Instructions that affect the parity flag set the flag to one for even parity and reset the flag to zero to indicate odd parity.

Auxiliary Carry Flag. The auxiliary carry flag indicates a carry out of bit 3 of the accumulator. You cannot test this flag directly in your program; it is present to enable the Decimal Adjust instructions to perform their function.

The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and XOR instructions.

C Reserved Words

DUAL FUNCTION KEYWORD/SYMBOLS

AND
NOT
OR
SHL
SHR
XOR

SYMBOLS

AAA
AAD
AAM
AAS
ADC
ADD
AH
AL
AX
BH
BL
BOUND
BP
BX
CALL
CBW
CH
CL
CLC
CLD
CLI
CMC
CMP
CMPS
CMPSB
CMPSW
CS
CWD
CX
DAA
DAS
DEC
DH
DI
DIV
DL
DS
DX

ENTER
ES
ESC
F2XM1
FABS
FADD
FADDP
FBLD
FBSTP
FCHS
FCLEX
FCOM
FCOMP
FCOMPP
FDECSTP
FDISI
FDIV
FDIVP
FDIVR
FDIVRP
FENI
FFREE
FIADD
FICOM
FICOMP
FIDIV
FIDIVR
FILD
FIMUL
FINCSTP
FINIT
FIST
FISTP
FISUB
FISUBR
FLD
FLD1
FLDCW

FLDENV
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FLDPI
FLDZ
FMUL
FMULP
FNCLEX
FNDISI
FNENI
FNINIT
FNOP
FNSAVE
FNSTCW
FNSTENV
FNSTSW
FPATAN
FPREM
FPTAN
FRNDINT
FRSTOR
FSAVE
FSCALE
FSQRT
FST
FSTCW
FSTENV
FSTP
FSTSW
FSUB
FSUBP
FSUBR
FSUBRP
FTST
FWAIT
FXAM

FXCH
FXTRACT
FYL2X
FYL2XP1
HLT
IDIV
IMUL
IN
INC
INS
INSB
INSW
INT
INTO
IRET
JA
JAE
JB
JBE
JC
JCXZ
JE
JG
JGE
JL
JLE
JMP
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNL
JNLE
JNO

JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
JZ
LAHF
LDS
LEA
LEAVE
LES
LOCK
LODS
LODSB
LODSW
LOOP
LOOPE
LOOPNE
LOOPNZ
LOOPZ
MOV
MOVS
MOVSB
MOVSW
MUL
NEG
NIL
NOP
OUT
OUTS
OUTSB
OUTSW
POP
POPA
POPF

PUSH
PUSHA
PUSHF
RCL
RCR
REP
REPE
REPNE
REPNZ
REPZ
RET
ROL
ROR
SAHF
SAL
SAR
SBB
SCAS
SCASB
SCASW
SI
SP
SS
ST
STC
STD
STI
STOS
STOSB
STOSW
SUB
TEST
WAIT
XCHG
XLAT
XLATB
??SEG

NON-CONFLICTING KEYWORDS

DA
DATE
DEBUG
EJ
EJECT
EP
ERRORPRINT
GEN
GENONLY
GO
IC

INCLUDE
LI
LIST
M1
MACRO
MEMORY
MOD186
MR
NODB
NODEBUG
NOEP

NOERRORPRINT
NOGE
NOGEN
NOLI
NOLIST
NOMACRO
NOMR
NOOBJECT
NOOJ
NOPAGING
NOPI

NOPR
NOPRINT
NOSB
NOSYMBOLS
NOTY
NOTYPE
NOXR
NOXREF
OBJECT
OJ
PAGELENGTH

PAGEWIDTH
PAGING
PI
PL
PR
PRINT
PW
RESTORE
RS
SA
SAVE

SB
STACK
SYMBOLS
TITLE
TT
TY
TYPE
WF
WORKFILES
XR
XREF

HANDS-OFF KEYWORDS

ABS
ASSUME
AT
BYTE
CODEMACRO
COMMON
DB
DD
DQ
DT
DUP
DW

DWORD
END
ENDM
ENDP
ENDS
EQ
EQU
EVEN
EXTRN
FAR
GE
GROUP

GT
HIGH
INPAGE
LABEL
LE
LENGTH
LOW
LT
MASK
MOD
MODRM
NAME

NE
NEAR
NOSEGFIX
NOTHING
OFFSET
ONLY186
ORG
PAGE
PARA
PREFX
PROC
PROCLEN

PTR
PUBLIC
PURGE
QWORD
RECORD
RELB
RELW
RFIX
RFIXM
RNFIX
RNFIXM
RWFIX

SEG
SEGFIX
SEGMENT
SHORT
SIZE
STRUC
TBYTE
THIS
TYPE
WIDTH
WORD
?

MPL Built-in Functions D

The following is a list of all MPL built-in functions.

%'_text end-of-line_ or %'_text_'

%(_balanced-text_)

%CI

%CO (_char_)

%*DEFINE(_macro-name_[_parameter-list_]) [LOCAL _local-list_] (_macro-body_)

%_n text-n-characters-long_

%EQS(_arg1,arg2_)

%EVAL(_expression_)

%EXIT

%GES(_arg1,arg2_)

%GTS(_arg1,arg2_)

%IF (_expression_) THEN (_balanced-text1_) [ELSE (_balanced-text2_)] FI

%IN

%LEN(_balanced-text_)

%LES(_arg1,arg2_)

%LTS(_arg1,arg2_)

%MATCH(_identifier1 delimiter identifier2_) (_balanced-text_)

%METACHAR(_balanced-text_)

%NES(_arg1,arg2_)

%OUT(_balanced-text_)

%REPEAT (_expression_) (_balanced-text_)

%SET(_macro-id,expression_)

%SUBSTR(_balanced-text,expression1,expression2_)

%WHILE (_expression_) (_balanced-text_)

Instructions in Hexadecimal Order

E

00	00000000	MOD REG R/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01	00000001	MOD REG R/M	ADD	EA,REG	WORD ADD (REG) TO EA
02	00000010	MOD REG R/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03	00000011	MOD REG R/M	ADD	REG,EA	WORD ADD (EA) TO REG
04	00000100		ADD	AL,DATA8	BYTE ADD DATA TO REG AL
05	00000101		ADD	AX,DATA16	WORD ADD DATA TO REG AX
06	00000110		PUSH	ES	PUSH (ES) ON STACK
07	00000111		POP	ES	POP STACK TO REG ES
08	00001000	MOD REG R/M	OR	EA,REG	BYTE OR (REG) TO EA
09	00001001	MOD REG R/M	OR	EA,REG	WORD OR (REG) TO EA
0A	00001010	MOD REG R/M	OR	REG,EA	BYTE OR (EA) TO REG
0B	00001011	MOD REG R/M	OR	REG,EA	WORD OR (EA) TO REG
0C	00001100		OR	AL,DATA8	BYTE OR DATA TO REG AL
0D	00001101		OR	AX,DATA16	WORD OR DATA TO REG AX
0E	00001110		PUSH	CS	PUSH (CS) ON STACK
0F	00001111		(not used)		
10	00010000	MOD REG R/M	ADC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11	00010001	MOD REG R/M	ADC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12	00010010	MOD REG R/M	ADC	REA,EA	BYTE ADD (EA) W/ CARRY TO REG
13	00010011	MOD REG R/M	ADC	REG,EA	WORD ADD (EA) W/ CARRY TO REG
14	00010100		ADC	AL,DATA8	BYTE ADD DATA W/CARRY TO REG AL
15	00010101		ADC	AX,DATA16	WORD ADD DATA W/ CARRY TO REG AX
16	00010110		PUSH	SS	PUSH (SS) ON STACK
17	00010111		POP	SS	POP STACK TO REG SS
18	00011000	MOD REG R/M	SBB	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19	00011001	MOD REG R/M	SBB	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A	00011010	MOD REG R/M	SBB	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B	00011011	MOD REG R/M	SBB	REG,EA	WORD SUB (EA) W/ BORROW FROM REG
1C	00011100		SBB	AL,DATA8	BYTE SUB DATA W/ BORROW FROM REG AL
1D	00011101		SBB	AX,DATA16	WORD SUB DATA W/ BORROW FROM REG AX
1E	00011110		PUSH	DS	PUSH (DS) ON STACK
1F	00011111		POP	DS	POP STACK TO REG DS
20	00100000	MOD REG R/M	AND	EA,REG	BYTE AND (REG) TO EA
21	00100001	MOD REG R/M	AND	EA,REG	WORD AND (REG) TO EA
22	00100010	MOD REG R/M	AND	REG,EA	BYTE AND (EA) TO REG
23	00100011	MOD REG R/M	AND	REG,EA	WORD AND (EA) TO REG
24	00100100		AND	AL,DATA8	BYTE AND DATA TO REG AL
25	00100101		AND	AX,DATA16	WORD AND DATA TO REG AX
26	00100110		ES:		SEGMENT OVERIDE W/ SEGMENT REG ES
27	00100111		DAA		DECIMAL ADJUST FOR ADD
28	00101000	MOD REG R/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29	00101001	MOD REG R/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A	00101010	MOD REG R/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B	00101011	MOD REG R/M	SUB	REG,EA	WORD SUBTRACT (EA) FROM REG
2C	00101100		SUB	AL,DATA8	BYTE SUBTRACT DATA FROM REG AL
2D	00101101		SUB	AX,DATA16	WORD SUBTRACT DATA FROM REG AX
2E	00101110		CS:		SEGMENT OVERIDE W/ SEGMENT REG CS
2F	00101111		DAS		DECIMAL ADJUST FOR SUBTRACT
30	00110000	MOD REG R/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31	00110001	MOD REG R/M	XOR	EA,REG	WORD XOR (REG) TO EA
32	00110010	MOD REG R/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33	00110011	MOD REG R/M	XOR	REG,EA	WORD XOR (EA) TO REG
34	00110100		XOR	AL,DATA8	BYTE XOR DATA TO REG AL
35	00110101		XOR	AX,DATA16	WORD XOR DATA TO REG AX
36	00110110		SS:		SEGMENT OVERIDE W/ SEGMENT REG SS
37	00110111		AAA		ASCII ADJUST FOR ADD
38	00111000	MOD REG R/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39	00111001	MOD REG R/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A	00111010	MOD REG R/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B	00111011	MOD REG R/M	CMP	REG,EA	WORD COMPARE (REG) WITH (EA)
3C	00111100		CMP	AL,DATA8	BYTE COMPARE DATA WITH (AL)
3D	00111101		CMP	AX,DATA16	WORD COMPARE DATA WITH (AX)
3E	00111110		DS:		SEGMENT OVERIDE W/ SEGMENT REG DS
3F	00111111		AAS		ASCII ADJUST FOR SUBTRACT
40	01000000		INC	AX	INCREMENT (AX)
41	01000001		INC	CX	INCREMENT (CX)

42	01000010				INC	DX	INCREMENT (DX)
43	01000011				INC	DX	INCREMENT (BX)
44	01000100				INC	SP	INCREMENT (SP)
45	01000101				INC	BP	INCREMENT (BP)
46	01000110				INC	SI	INCREMENT (SI)
47	01000111				INC	DI	INCREMENT (DI)
48	01001000				DEC	AX	DECREMENT (AX)
49	01001001				DEC	CX	DECREMENT (CX)
4A	01001010				DEC	DX	DECREMENT (DX)
4B	01001011				DEC	BX	DECREMENT (BX)
4C	01001100				DEC	SP	DECREMENT (SP)
4D	01001101				DEC	BP	DECREMENT (BP)
4E	01001110				DEC	SI	DECREMENT (SI)
4F	01001111				DEC	DI	DECREMENT (DI)
50	01010000				PUSH	AX	PUSH (AX) ON STACK
51	01010001				PUSH	CX	PUSH (CX) ON STACK
52	01010010				PUSH	DX	PUSH (DX) ON STACK
53	01010011				PUSH	BX	PUSH (BX) ON STACK
54	01010100				PUSH	SP	PUSH (SP) ON STACK
55	01010101				PUSH	BP	PUSH (BP) ON STACK
56	01010110				PUSH	SI	PUSH (SI) ON STACK
57	01010111				PUSH	DI	PUSH (DI) ON STACK
58	01011000				POP	AX	POP STACK TO REG AX
59	01011001				POP	CX	POP STACK TO REG CX
5A	01011010				POP	DX	POP STACK TO REG DX
5B	01011011				POP	BX	POP STACK TO REG BX
5C	01011100				POP	SP	POP STACK TO REG SP
5D	01011101				POP	BP	POP STACK TO REG BP
5E	01011110				POP	SI	POP STACK TO REG SI
5F	01011111				POP	DI	POP STACK TO REG DI
60	01100000				PUSHA		PUSH ALL DATA, INDEX, AND POINTER REGISTERS
61	01100001				POPA		POP ALL DATA, INDEX, AND POINTER REGISTERS
62	01100010	MOD	REG	R/M	BOUND	REG,EA	CHECK INDEX IN REG AGAINST BOUNDS AT EA
63	01100011				(not used)		
64	01100100				(not used)		
65	01100101				(not used)		
66	01100110				(not used)		
67	01100111				(not used)		
68	01101000				PUSH	DATA16	PUSH WORD DATA ON STACK
69	01101001	MOD	REG	R/M	IMUL	REG,EA,DATA16	MULTIPLY (EA) BY WORD DATA; SIGNED
6A	01101010				PUSH	DATA8	PUSH BYTE DATA ON STACK; SIGN-EXTEND
6B	01101011	MOD	REG	R/M	IMUL	REG,EA,DATA8	MULTIPLY (EA) BY BYTE DATA; SIGNED
6C	01101100				INS	DST8	BYTE INPUT, STRING OP
6D	01101101				INS	DST16	WORD INPUT, STRING OP
6E	01101110				OUTS	DST8	BYTE OUTPUT, STRING OP
6F	01101111				OUTS	DST16	WORD OUTPUT, STRING OP
70	01110000				JO	DISP8	JUMP ON OVERFLOW
71	01110001				JNO	DISP8	JUMP ON NOT OVERFLOW
72	01110010				JC/JB/JNAE	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73	01110011				JNC/JNB/JAE	DISP8	JUMP ON NOT BELOW/ABOVE OR EQUAL
74	01110100				JE/JZ	DISP8	JUMP ON EQUAL/ZERO
75	01110101				JNE/JNZ	DISP8	JUMP ON NOT EQUAL/NOT ZERO
76	01110110				JBE/JNA	DISP8	JUMP ON BELOW OR EQUAL/NOT ABOVE
77	01110111				JNBE/JA	DISP8	JUMP ON NOT BELOW OR EQUAL/ABOVE
78	01111000				JS	DISP8	JUMP ON SIGN
79	01111001				JNS	DISP8	JUMP ON NOT SIGN
7A	01111010				JP/JPE	DISP8	JUMP ON PARITY/PARITY EVEN
7B	01111011				JNP/JPO	DISP8	JUMP ON NOT PARITY/PARITY ODD
7C	01111100				JL/JNGE	DISP8	JUMP ON LESS/NOT GREATER OR EQUAL
7D	01111101				JNL/JGE	DISP8	JUMP ON NOT LESS/GREATER OR EQUAL
7E	01111110				JLE/JNG	DISP8	JUMP ON LESS OR EQUAL/NOT GREATER
7F	01111111				JNLE/JG	DISP8	JUMP ON NOT LESS OR EQUAL/GREATER
80	10000000	MOD	000	R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
80	10000000	MOD	001	R/M	OR	EA,DATA8	BYTE OR DATA TO EA
80	10000000	MOD	010	R/M	ADC	EA,DATA8	BYTE ADD DATA W/CARRY TO EA
80	10000000	MOD	011	R/M	SBB	EA,DATA8	BYTE SUB DATA W/BORROW FROM EA
80	10000000	MOD	100	R/M	AND	EA,DATA8	BYTE AND DATA TO EA
80	10000000	MOD	101	R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
80	10000000	MOD	110	R/M	XOR	EA,DATA8	BYTE XOR DATA TO EA
80	10000000	MOD	111	R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
81	10000001	MOD	000	R/M	ADD	EA,DATA16	WORD ADD DATA TO EA
81	10000001	MOD	001	R/M	OR	EA,DATA16	WORD OR DATA TO EA
81	10000001	MOD	010	R/M	ADC	EA,DATA16	WORD ADD DATA W/CARRY TO EA

81	10000001	MOD	011	R/M	SBB	EA,DATA16	WORD SUB DATA W/ BORROW FROM EA
81	10000001	MOD	100	R/M	AND	EA,DATA16	WORD AND DATA TO EA
81	10000001	MOD	101	R/M	SUB	EA,DATA16	WORD SUBTRACT DATA FROM EA
81	10000001	MOD	110	R/M	XOR	EA,DATA16	WORD XOR DATA TO EA
81	10000001	MOD	111	R/M	CMP	EA,DATA16	WORD COMPARE DATA WITH (EA)
82	10000010	MOD	000	R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
82	10000010	MOD	001	R/M	(not used)		
82	10000010	MOD	010	R/M	ADC	EA,DATA8	BYTE ADD DATA W/ CARRY TO EA
82	10000010	MOD	011	R/M	SBB	EA,DATA8	BYTE SUB DATA W/ BORROW FROM EA
82	10000010	MOD	100	R/M	(not used)		
82	10000010	MOD	101	R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
82	10000010	MOD	110	R/M	(not used)		
82	10000010	MOD	111	R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
83	10000011	MOD	000	R/M	ADD	EA,DATA8	WORD ADD DATA TO EA
83	10000011	MOD	001	R/M	(not used)		
83	10000011	MOD	010	R/M	ADC	EA,DATA8	WORD ADD DATA W/ CARRY TO EA
83	10000011	MOD	011	R/M	SBB	EA,DATA8	WORD SUB DATA W/ BORROW FROM EA
83	10000011	MOD	100	R/M	(not used)		
83	10000011	MOD	101	R/M	SUB	EA,DATA8	WORD SUBTRACT DATA FROM EA
83	10000011	MOD	110	R/M	(not used)		
83	10000011	MOD	111	R/M	CMP	EA,DATA8	WORD COMPARE DATA WITH (EA)
84	10000100	MOD	REG	R/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85	10000101	MOD	REG	R/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86	10000110	MOD	REG	R/M	XCHG	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87	10000111	MOD	REG	R/M	XCHG	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88	10001000	MOD	REG	R/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89	10001001	MOD	REG	R/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A	10001010	MOD	REG	R/M	MOV	REG,EA	BYTE MOVE (EA) TO REG
8B	10001011	MOD	REG	R/M	MOV	REG,EA	WORD MOVE (EA) TO REG
8C	10001100	MOD	0SR	R/M	MOV	EA,SR	WORD MOVE (SEGMENT REG SR) TO EA
8C	10001100	MOD	1--	R/M	(not used)		
8D	10001101	MOD	REG	R/M	LEA	REG,EA	LOAD EFFECTIVE ADDRESS OF EA TO REG
8E	10001110	MOD	0SR	R/M	MOV	SR,EA	WORD MOVE (EA) TO SEGMENT REG SR
8E	10001110	MOD	--	R/M	(not used)		
8F	10001111	MOD	000	R/M	POP	EA	POP STACK TO EA
8F	10001111	MOD	001	R/M	(not used)		
8F	10001111	MOD	010	R/M	(not used)		
8F	10001111	MOD	011	R/M	(not used)		
8F	10001111	MOD	100	R/M	(not used)		
8F	10001111	MOD	101	R/M	(not used)		
8F	10001111	MOD	110	R/M	(not used)		
8F	10001111	MOD	111	R/M	(not used)		
90	10010000				XCHG	AX,AX	EXCHANGE (AX) WITH (AX), (NOP)
91	10010001				XCHG	AX,CX	EXCHANGE (AX) WITH (CX)
92	10010010				XCHG	AX,DX	EXCHANGE (AX) WITH (DX)
93	10010011				XCHG	AX,BX	EXCHANGE (AX) WITH (BX)
94	10010100				XCHG	AX,SP	EXCHANGE (AX) WITH (SP)
95	10010101				XCHG	AX,BP	EXCHANGE (AX) WITH (BP)
96	10010110				XCHG	AX,SI	EXCHANGE (AX) WITH (SI)
97	10010111				XCHG	AX,DI	EXCHANGE (AX) WITH (DI)
98	10011000				CBW		BYTE CONVERT (AL) TO WORD (AX)
99	10011001				CWD		WORD CONVERT (AX) TO DOUBLE WORD
9A	10011010				CALL	DISP16,SEG16	DIRECT INTER SEGMENT CALL
9B	10011011				WAIT		WAIT FOR TEST SIGNAL
9C	10011100				PUSHF		PUSH FLAGS ON STACK
9D	10011101				POPF		POP STACK TO FLAGS
9E	10011110				SAHF		STORE (AH) INTO FLAGS
9F	10011111				LAHF		LOAD REG AH WITH FLAGS
A0	10100000				MOV	AL,ADDR16	BYTE MOVE (ADDR) TO REG AL
A1	10100001				MOV	AX,ADDR16	WORD MOVE (ADDR) TO REG AX
A2	10100010				MOV	ADDR16,AL	BYTE MOVE (AL) TO ADDR
A3	10100011				MOV	ADDR16,AX	WORD MOVE (AX) TO ADDR
A4	10100100				MOVS	DST8,SRC8	BYTE MOVE, STRING OP
A5	10100101				MOVS	DST16,SRC16	WORD MOVE, STRING OP
A6	10100110				CMPS	SIPTR.DIPTR	COMPARE BYTE, STRING OP
A7	10100111				CMPS	SIPTR,DIPTR	COMPARE WORD, STRING OP
A8	10101000				TEST	AL,DATA8	BYTE TEST (AL) WITH DATA
A9	10101001				TEST	AX,DATA16	WORD TEST (AX) WITH DATA
AA	10101010				STOS	DST8	BYTE STORE, STRING OP
AB	10101011				STOS	DST16	WORD STORE, STRING OP
AC	10101100				LODS	SRC8	BYTE LOAD, STRING OP
AD	10101101				LODS	SRC16	WORD LOAD, STRING OP
AE	10101110				SCAS	DIPTR8	BYTE SCAN, STRING OP

AF 10101111				SCAS	DIPTR16	WORD SCAN, STRING OP
B0 10110000				MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001				MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010				MOV	DL,DATA8	BYTE MOVE DATA TO REG DL
B3 10110011				MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100				MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101				MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110				MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111				MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000				MOV	AX,DATA16	WORD MOVE DATA TO REG AX
B9 10111001				MOV	CX,DATA16	WORD MOVE DATA TO REG CX
BA10111010				MOV	DX,DATA16	WORD MOVE DATA TO REG DX
BB10111011				MOV	BX,DATA16	WORD MOVE DATA TO REG BX
BC10111100				MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD10111101				MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE10111110				MOV	SI,DATA16	WORD MOVE DATA TO REG SI
BF10111111				MOV	DI,DATA16	WORD MOVE DATA TO REG DI
C0 11000000	MOD	000	R/M	ROL	EA,DATA8	BYTE ROTATE EA LEFT DATA8 BITS
C0 11000000	MOD	001	R/M	ROR	EA,DATA8	BYTE ROTATE EA RIGHT DATA8 BITS
C0 11000000	MOD	010	R/M	RCL	EA,DATA8	BYTE ROTATE EA LEFT THRU CARRY DATA8 BITS
C0 11000000	MOD	011	R/M	RCR	EA, DATA8	BYTE ROTATE EA RIGHT THRU CARRY DATA8 BITS
C0 11000000	MOD	100	R/M	SHL/SAL	EA,DATA8	BYTE SHIFT EA LEFT DATA8 BITS
C0 11000000	MOD	101	R/M	SHR	EA,DATA8	BYTE SHIFT EA RIGHT DATA8 BITS
C0 11000000	MOD	110	R/M	(not used)		
C0 11000000	MOD	111	R/M	SAR	EA,DATA8	BYTE SHIFT SIGNED EA RIGHT DATA8 BITS
C1 11000001	MOD	000	R/M	ROL	EA,DATA8	WORD ROTATE EA LEFT DATA8 BITS
C1 11000001	MOD	001	R/M	ROR	EA,DATA8	WORD ROTATE EA RIGHT DATA8 BITS
C1 11000001	MOD	010	R/M	RCL	EA,DATA8	WORD ROTATE EA LEFT THRU CARRY DATA8 BITS
C1 11000001	MOD	011	R/M	RCR	EA,DATA8	WORD ROTATE EA RIGHT THRU CARRY DATA8 BITS
C1 11000001	MOD	100	R/M	SHL/SAL	EA,DATA8	WORD SHIFT EA LEFT DATA8 BITS
C1 11000001	MOD	101	R/M	SHR	EA,DATA8	WORD SHIFT EA RIGHT DATA8 BITS
C1 11000001	MOD	110	R/M	(not used)		
C1 11000001	MOD	111	R/M	SAR	EA,DATA8	WORD SHIFT SIGNED EA RIGHT DATA8 BITS
C2 11000010				RET	DATA16	INTRA SEGMENT RETURN, ADD DATA TO REG SP
C3 11000011				RET		INTRA SEGMENT RETURN
C4 11000100	MOD	REG	R/M	LES	REG,EA	WORD LOAD REG AND SEGMENT REG ES
C5 11000101	MOD	REG	R/M	LDS	REG,EA	WORD LOAD REG AND SEGMENT REG DS
C6 11000110	MOD	000	R/M	MOV	EA,DATA8	BYTE MOVE DATA TO EA
C6 11000110	MOD	001	R/M	(not used)		
C6 11000110	MOD	010	R/M	(not used)		
C6 11000110	MOD	011	R/M	(not used)		
C6 11000110	MOD	100	R/M	(not used)		
C6 11000110	MOD	101	R/M	(not used)		
C6 11000110	MOD	110	R/M	(not used)		
C6 11000110	MOD	111	R/M	(not used)		
C7 11000111	MOD	000	R/M	MOV	EA,DATA16	WORD MOVE DATA TO EA
C7 11000111	MOD	001	R/M	(not used)		
C7 11000111	MOD	010	R/M	(not used)		
C7 11000111	MOD	011	R/M	(not used)		
C7 11000111	MOD	100	R/M	(not used)		
C7 11000111	MOD	101	R/M	(not used)		
C7 11000111	MOD	110	R/M	(not used)		
C7 11000111	MOD	111	R/M	(not used)		
C8 11001000				ENTER	DATA16,DATA8	PERFORM ENTER SEQUENCE
C9 11001001				LEAVE		PERFORM LEAVE SEQUENCE
CA11001010				RET	DATA16	INTER SEGMENT RETURN, ADD DATA TO REG SP
CB11001011				RET		INTER SEGMENT RETURN
CC11001100				INT	3	TYPE 3 INTERRUPT
CD11001101				INT	TYPE	TYPED INTERRUPT
CE11001110				INTO		INTERRUPT ON OVERFLOW
CF11001111				IRET		RETURN FROM INTERRUPT
D0 11010000	MOD	000	R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD	001	R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD	010	R/M	RCL	EA,1	BYTE ROTATE EA LEFT THRU CARRY 1 BIT
D0 11010000	MOD	011	R/M	RCR	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD	100	R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD	101	R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD	110	R/M	(not used)		
D0 11010000	MOD	111	R/M	SAR	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD	000	R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT

D1 11010001	MOD	001	R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1 11010001	MOD	010	R/M	RCL	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1 11010001	MOD	011	R/M	RCR	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1 11010001	MOD	100	R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1 11010001	MOD	101	R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1 11010001	MOD	110	R/M	(not used)		
D1 11010001	MOD	111	R/M	SAR	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT
D2 11010010	MOD	000	R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2 11010010	MOD	001	R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2 11010010	MOD	010	R/M	RCL	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2 11010010	MOD	011	R/M	RCR	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2 11010010	MOD	100	R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2 11010010	MOD	101	R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2 11010010	MOD	110	R/M	(not used)		
D2 11010010	MOD	111	R/M	SAR	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS
D3 11010011	MOD	000	R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3 11010011	MOD	001	R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS
D3 11010011	MOD	010	R/M	RCL	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3 11010011	MOD	011	R/M	RCR	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3 11010011	MOD	100	R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3 11010011	MOD	101	R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3 11010011	MOD	110	R/M	(not used)		
D3 11010011	MOD	111	R/M	SAR	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4 11010100	00001010			AAM		ASCII ADJUST FOR MULTIPLY
D5 11010101	00001010			AAD		ASCII ADJUST FOR DIVIDE
D6 11010110				(not used)		
D7 11010111				XLAT	TABLE	TRANSLATE USING (BX)
D8 11011---	MOD	---	R/M	ESC	EA	ESCAPE TO EXTERNAL DEVICE
D8 11011000	MOD	000	R/M	FADD	Short-real	ADD 4-BYTE EA TO ST
D8 11011000	MOD	001	R/M	FMUL	Short-real	MULTIPLY ST BY 4-BYTE EA
D8 11011000	MOD	010	R/M	FCOM	Short-real	COMPARE 4-BYTE EA WITH ST
D8 11011000	MOD	011	R/M	FCOMP	Short-real	COMPARE 4-BYTE EA WITH ST AND POP
D8 11011000	MOD	100	R/M	FSUB	Short-real	SUBTRACT 4 BYTE EA FROM ST
D8 11011000	MOD	101	R/M	FSUBR	Short real	SUBTRACT ST FROM 4-BYTE EA
D8 11011000	MOD	110	R/M	FDIV	Short-real	DIVIDE ST BY 4-BYTE EA
D8 11011000	MOD	111	R/M	FDIVR	Short-real	DIVIDE 4-BYTE EA BY ST
D8 11011000	1 1	000	(i)	FADD	ST, ST(i)	ADD ELEMENT TO ST
D8 11011000	1 1	001	(i)	FMUL	ST, ST(i)	MULTIPLY ST BY ELEMENT
D8 11011000	1 1	010	(i)	FCOM	ST(i)	COMPARE ST(i) WITH ST
D8 11011000	1 1	011	(i)	FCOMP	ST(i)	COMPARE ST(i) WITH ST AND POP
D8 11011000	1 1	100	(i)	FSUB	ST, ST(i)	SUBTRACT ELEMENT FROM ST
D8 11011000	1 1	101	(i)	FSUBR	ST, ST(i)	SUBTRACT ST FROM STACK ELEMENT
D8 11011000	1 1	110	(i)	FDIV	ST, ST(i)	DIVIDE ST BY ELEMENT
D8 11011000	1 1	111	(i)	FDIVR	ST, ST(i)	DIVIDE ST(i) BY ST
D9 11011001	MOD	000	R/M	FLD	Short-real	PUSH 4-BYTE EA TO ST
D9 11011001	MOD	001	R/M	(not used)		
D9 11011001	MOD	010	R/M	FST	Short-real	STORE 4-BYTE REAL TO EA
D9 11011001	MOD	011	R/M	FSTP	Short-real	STORE 4-BYTE REAL TO EA AND POP
D9 11011001	MOD	100	R/M	FLDENV	14 BYTES	LOAD 8087 ENVIRONMENT FROM EA
D9 11011001	MOD	101	R/M	FLDCW	2-BYTES	LOAD CONTROL WORD FROM EA
D9 11011001	MOD	110	R/M	FSTENV	14-BYTES	STORE 8087 ENVIRONMENT INTO EA
D9 11011001	MOD	111	R/M	FSTCW	2-BYTES	STORE CONTROL WORD INTO EA
D9 11011001	1 1	000	(i)	FLD	ST(i)	PUSH ST(i) ONTO ST
D9 11011001	1 1	001	(i)	FXCH	ST(i)	EXCHANGE ST AND ST(i)
D9 11011001	1 1	010	000	FNOP		STORE ST IN ST
D9 11011001	1 1	010	001	(not used)		
D9 11011001	1 1	010	01-	(not used)		
D9 11011001	1 1	010	1--	(not used)		
D9 11011001	1 1	011	(i)	*(1)		
D9 11011001	1 1	100	000	FCHS		CHANGE SIGN OF ST
D9 11011001	1 1	100	001	FABS		TAKE ABSOLUTE VALUE OF ST
D9 11011001	1 1	100	01-	(not used)		
D9 11011001	1 1	100	100	FTST		TEST ST AGAINST 0.0
D9 11011001	1 1	100	101	FXAM		EXAMINE ST AND REPORT CONDITION CODE
D9 11011001	1 1	100	11-	(not used)		
D9 11011001	1 1	101	000	FLD1		PUSH +1.0 TO ST
D9 11011001	1 1	101	001	FLDL2T		PUSH $\log_2 10$ TO ST
D9 11011001	1 1	101	010	FLDL2E		PUSH $\log_2 e$ TO ST
D9 11011001	1 1	101	011	FLDPI		PUSH Pi TO ST
D9 11011001	1 1	101	100	FLDLG2		PUSH $\log_{10} 2$ TO ST
D9 11011001	1 1	101	101	FLDLN2		PUSH $\log_e 2$ TO ST
D9 11011001	1 1	101	110	FLDZ		PUSH ZERO TO ST

D9 11011001	1 1	101	111	(not used)		
D9 11011001	1 1	110	000	F2XM1		CALCULATE $2^X - 1$
D9 11011001	1 1	110	001	FYL2X		CALCULATE FUNCTION $Y*\log_2 X$
D9 11011001	1 1	110	010	FPTAN		CALCULATE TAN OF θ AS A RATIO
D9 11011001	1 1	110	011	FPATAN		CALCULATE ARCTAN OF θ
D9 11011001	1 1	110	100	FXTRACT		EXTRACT EXPONENT AND SIGNIFICAND FROM ST VALUE
D9 11011001	1 1	110	101	(not used)		
D9 11011001	1 1	110	110	FDECSTP		DECREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1	110	111	FINCSTP		INCREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1	111	000	FPREM		MODULO DIVISION OF ST BY ST(1)
D9 11011001	1 1	110	001	FYL2XP1		CALCULATE VALUE OF $Y*\log_2 (X+1)$
D9 11011001	1 1	111	010	FSQRT		CALCULATE SQUARE ROOT OF ST
D9 11011001	1 1	111	011	(not used)		
D9 11011001	1 1	111	100	FRNDINT		ROUND ST TO INTEGER
D9 11011001	1 1	111	101	FSCALE		ADD ST(1) TO EXPONENT OF ST
D9 11011001	1 1	111	11-	(not used)		
DA 11011010	MOD	000	R/M	FIADD	Short-integer	ADD 4-BYTE INTEGER EA TO ST
DA 11011010	MOD	001	R/M	FIMUL	Short-integer	MULTIPLY ST BY 4-BYTE INTEGER EA
DA 11011010	MOD	010	R/M	FICOM	Short-integer	CONVERT 4-BYTE INTEGER EA, AND COMPARE WITH ST
DA 11011010	MOD	011	R/M	FICOMP	Short-integer	CONVERT 4-BYTE INTEGER EA, COMPARE WITH ST, POP
DA 11011010	MOD	100	R/M	FISUB	Short-integer	SUBTRACT 4-BYTE INTEGER EA FROM ST
DA 11011010	MOD	101	R/M	FISUBR	Short-integer	SUBTRACT ST FROM 4-BYTE INTEGER EA
DA 11011010	MOD	110	R/M	FIDIV	Short-integer	DIVIDE ST BY 4-BYTE INTEGER EA
DA 11011010	MOD	111	R/M	FIDIVR	Short-integer	DIVIDE 4-BYTE INTEGER EA BY ST
DA 11011010	1 1	--	---	(not used)		
DB 11011011	MOD	000	R/M	FILD	Short-integer	PUSH 4-BYTE INTEGER EA ONTO ST
DB 11011011	MOD	001	R/M	(not used)		
DB 11011011	MOD	010	R/M	FIST	Short integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA
DB 11011011	MOD	011	R/M	FISTP	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA, POP
DB 11011011	MOD	100	R/M	(not used)		
DB 11011011	MOD	101	R/M	FLD	Temp-real	PUSH 10-BYTE EA ONTO ST
DB 11011011	MOD	110	R/M	Reserved		
DB 11011011	MOD	111	R/M	FSTP	Temp-real	STORE ST INTO 10-BYTE EA, POP
DB 11011011	1 1	0--	---	Reserved		
DB 11011011	1 1	100	000	FENI		ENABLE INTERRUPT
DB 11011011	1 1	100	001	FDISI		DISABLE INTERRUPTS
DB 11011011	1 1	100	010	FCLEX		CLEAR EXCEPTIONS
DB 11011011	1 1	100	011	FINIT		INITIALIZE PROCESSOR
DB 11011011	1 1	100	1--	Reserved		
DB 11011011	1 1	101	---	Reserved		
DB 11011011	1 1	11-	---	Reserved		
DC 11011100	MOD	000	R/M	FADD	Long-real	ADD 8-BYTE EA TO ST
DC 11011100	MOD	001	R/M	FMUL	Long-real	MULTIPLY ST BY 8-BYTE EA
DC 11011100	MOD	010	R/M	FCOM	Long-real	COMPARE ST WITH 8-BYTE EA
DC 11011100	MOD	011	R/M	FCOMP	Long-real	COMPARE ST WITH 8-BYTE EA, POP STACK
DC 11011100	MOD	100	R/M	FSUB	Long-real	SUBTRACT 8-BYTE EA FROM ST
DC 11011100	MOD	101	R/M	FSUBR	Long-real	SUBTRACT ST FROM 8-BYTE EA
DC 11011100	MOD	110	R/M	FDIV	Long-real	DIVIDE ST BY 8-BYTE EA
DC 11011100	MOD	111	R/M	FDIVR	Long-real	DIVIDE 8-BYTE EA BY ST
DC 11011100	1 1	000	(i)	FADD	ST(i), ST	ADD ST TO ELEMENT
DC 11011100	1 1	001	(i)	FMUL	ST(i), ST	MULTIPLY ELEMENT BY ST
DC 11011100	1 1	010	(i)	*(2)		
DC 11011100	1 1	011	(i)	*(3)		
DC 11011100	1 1	100	(i)	FSUBR	ST(i), ST	SUBTRACT ST FROM ELEMENT
DC 11011100	1 1	101	(i)	FSUB	ST(i), ST	SUBTRACT ELEMENT FROM ST
DC 11011100	1 1	110	(i)	FDIVR	ST(i), ST	DIVIDE ST(i) BY ST
DC 11011100	1 1	111	(i)	FDIV	ST(i), ST	DIVIDE ST BY ST(i)
DD 11011101	MOD	000	R/M	FLD	Long-real	PUSH 8-BYTE EA ONTO ST
DD 11011101	MOD	001	R/M	Reserved		
DD 11011101	MOD	010	R/M	FST	Long-real	STORE ST INTO 8-BYTE EA
DD 11011101	MOD	011	R/M	FSTP	Long-real	STORE ST INTO 8-BYTE EA, POP
DD 11011101	MOD	100	R/M	FRSTOR	94-BYTES	RESTORE 8087 STATE FROM EA
DD 11011101	MOD	101	R/M	Reserved		
DD 11011101	MOD	110	R/M	FSAVE	94-BYES	SAVE 8087 STATE TO EA
DD 11011101	MOD	111	R/M	FSTSW	2-BYTES	STORE 8087 STATUS WORD TO 2-BYTE EA
DD 11011101	1 1	000	(i)	FFREE	ST(i)	SET STACK TAG TO "EMPTY"
DD 11011101	1 1	001	(i)	*(4)		
DD 11011101	1 1	010	(i)	FST	ST(i)	STORE ST INTO ST(i)
DD 11011101	1 1	011	(i)	FSTP	ST(i)	STORE ST INTO ST(i), POP
DD 11011101	1 1	1--	---	Reserved		
DE 11011110	MOD	000	R/M	FIADD	Word-integer	ADD 2-BYTE INTEGER EA TO ST
DE 11011110	MOD	001	R/M	FIMUL	Word-integer	MULTIPLY ST BY 2-BYTE INTEGER EA

DE 11011110	MOD	010	R/M	FICOM	Word-integer	COMPARE 2-BYTE EA INTEGER WITH ST
DE 11011110	MOD	011	R/M	FICOMP	Word-integer	COMPARE 2-BYTE INTEGER EA WITH ST POP
DE 11011110	MOD	100	R/M	FISUB	Word-integer	SUBTRACT 2-BYTE INTEGER EA FROM ST
DE 11011110	MOD	101	R/M	FISUBR	Word-integer	SUBTRACT ST FROM 2-BYTE INTEGER EA
DE 11011110	MOD	110	R/M	FIDIV	Word-integer	DIVIDE ST BY 2-BYTE INTEGER EA
DE 11011110	MOD	111	R/M	FIDIVR	Word-integer	DIVIDE 2-BYTE INTEGER EA BY ST
DE 11011110	1 1	000	(i)	FADDP	ST(i), ST	ADD ST TO ELEMENT, POP
DE 11011110	1 1	001	(i)	FMULP	ST(i), ST	MULTIPLY ST BY ELEMENT, POP
DE 11011110	1 1	010	---	*(5)		
DE 11011110	1 1	011	000	Reserved		
DE11011110	1 1		011	001	FCOMPP	COMPARE ST WITH ST(1), POP TWICE
DE 11011110	1 1	011	01-	Reserved		
DE 11011110	1 1	011	1--	Reserved		
DE 11011110	1 1	100	(i)	FSUBRP	ST(i), ST	SUBTRACT ST FROM ELEMENT, POP
DE 11011110	1 1	101	(i)	FSUBP	ST(i), ST	SUBTRACT ST(i) FROM ST, POP
DE 11011110	1 1	110	(i)	FDIVRP	ST(i), ST	DIVIDE STACK ELEMENT BY ST, POP
DE 11011110	1 1	111	(i)	FDIVP	ST(i), ST	DIVIDE ST BY STACK ELEMENT, POP
DF 11011111	MOD	000	R/M	FILD	Word-integer	CONVERT 2-BYTE EA AND PUSH ONTO STACK
DF 11011111	MOD	001	R/M	Reserved		
DF 11011111	MOD	010	R/M	FIST	Word-integer	ROUND ST AND STORE IN 2-BYTE INTEGER EA
DF 11011111	MOD	011	R/M	FISTP	Word-integer	ROUND ST, STORE IN 2-BYTE INTEGER EA, POP
DF 11011111	MOD	100	R/M	FBLD	Packed decimal	LOAD BCD TO ST
DF 11011111	MOD	101	R/M	FILD	Long-integer	CONVERT 8-BYTE INTEGER EA AND PUSH ONTO STACK
DF 11011111	MOD	110	R/M	FBSTP	Packed decimal	CONVERT ST, STORE IN 10-BYTE BCD EA, POP
DF 11011111	MOD	111	R/M	FISTP	Long-integer	ROUND ST, STORE IN 8-BYTE INTEGER EA, POP
DF 11011111	1 1	000	(i)	*(6)		
DF 11011111	1 1	001	(i)	*(7)		
DF 11011111	1 1	010	(i)	*(8)		
DF 11011111	1 1	011	(i)	*(9)		
DF 11011111	1 1	...	...	Reserved		
E0 11100000				LOOPNZ/LOOPNE	DISP8	LOOP (CX) TIMES WHILE NOT ZERO/NOT EQUAL
E1 11100001				LOOPZ/LOOPE	DISP8	LOOP (CX) TIMES WHILE ZERO/EQUAL
E2 11100010				LOOP	DISP8	LOOP (CX) TIMES
E3 11100011				JCXZ	DISP8	JUMP ON (CX)=0
E4 11100100				IN	AL,PORT	BYTE INPUT FROM PORT TO REG AL
E5 11100101				IN	AX,PORT	WORD INPUT FROM PORT TO REG AX
E6 11100110				OUT	PORT,AL	BYTE OUTPUT (AL) TO PORT
E7 11100111				OUT	PORT,AX	WORD OUTPUT (AX) TO PORT
E8 11101000				CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001				JMP	DISP16	DIRECT INTRA SEGMENT JUMP
EA 11101010				JMP	DISP16 SEG16	DIRECT INTER SEGMENT JUMP
EB 11101010				JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC 11101010				IN	AL,DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED 11101010				IN	AX,DX	WORD INPUT FROM PORT (DX) TO REG AX
EE 11101010				OUT	DX,AL	BYTE OUTPUT (AL) TO PORT (DX)
EF 11101010				OUT	DX,AX	WORD OUTPUT (AX) TO PORT (DX)
F0 11110000				LOCK		BUS LOCK PREFIX
F1 11110001				(not used)		
F2 11110010				REPNZ/REPNE		REPEAT WHILE (CX)≠0 AND (ZF)=0
F3 11110011				REPZ/REPE/REP		REPEAT WHILE (CX)≠0 AND (ZF)=1
F4 11110100				HLT		HALT
F5 11110101				CMC		COMPLEMENT CARRY FLAG
F6 11110110	MOD	000	R/M	TEST	EA DATA8	BYTE TEST (EA) WITH DATA
F6 11110110	MOD	001	R/M	(not used)		
F6 11110110	MOD	010	R/M	NOT	EA	BYTE INVERT EA
F6 11110110	MOD	011	R/M	NEG	EA	BYTE NEGATE EA
F6 11110110	MOD	100	R/M	MUL	EA	BYTE MULTIPLY BY (EA), UNSIGNED
F6 11110110	MOD	101	R/M	IMUL	EA	BYTE MULTIPLY BY (EA), SIGNED
F6 11110110	MOD	110	R/M	DIV	EA	BYTE DIVIDE BY (EA), UNSIGNED
F6 11110110	MOD	111	R/M	IDIV	EA	BYTE DIVIDE BY (EA), SIGNED
F7 11110111	MOD	000	R/M	TEST	EA,DATA16	WORD TEST (EA) WITH DATA
F7 11110111	MOD	001	R/M	(not used)		
F7 11110111	MOD	010	R/M	NOT	EA	WORD INVERT EA
F7 11110111	MOD	011	R/M	NEG	EA	WORD NEGATE EA
F7 11110111	MOD	100	R/M	MUL	EA	WORD MULTIPLY BY (EA), UNSIGNED
F7 11110111	MOD	101	R/M	IMUL	EA	WORD MULTIPLY BY (EA), SIGNED
F7 11110111	MOD	110	R/M	DIV	EA	WORD DIVIDE BY (EA), UNSIGNED
F7 11110111	MOD	111	R/M	IDIV	EA	WORD DIVIDE BY (EA), SIGNED
F8 11111000				CLC		CLEAR CARRY FLAG
F9 11111001				STC		SET CARRY FLAG
FA 11111010				CLI		CLEAR INTERRUPT FLAG
FB 11111011				STI		SET INTERRUPT FLAG

FC 11111100			CLD		CLEAR DIRECTION FLAG
FD 11111101			STD		SET DIRECTION FLAG
FE 11111110	MOD 000	R/M	INC	EA	BYTE INCREMENT EA
FE 11111110	MOD 001	R/M	DEC	EA	BYTE DECREMENT EA
FE 11111110	MOD 010	R/M	(not used)		
FE 11111110	MOD 011	R/M	(not used)		
FE 11111110	MOD 100	R/M	(not used)		
FE 11111110	MOD 101	R/M	(not used)		
FE 11111110	MOD 110	R/M	(not used)		
FE 11111110	MOD 111	R/M	(not used)		
FF 11111111	MOD 000	R/M	INC	EA	WORD INCREMENT EA
FF 11111111	MOD 001	R/M	DEC	EA	WORD DECREMENT EA
FF 11111111	MOD 010	R/M	CALL	EA	INDIRECT INTRA SEGMENT CALL
FF 11111111	MOD 011	R/M	CALL	EA	INDIRECT INTER SEGMENT CALL
FF 11111111	MOD 100	R/M	JMP	EA	INDIRECT INTRA SEGMENT JUMP
FF 11111111	MOD 101	R/M	JMP	EA	INDIRECT INTER SEGMENT JUMP
FF 11111111	MOD 110	R/M	PUSH	EA	PUSH (EA) ON STACK
FF 11111111	MOD 111	R/M	(not used)		

REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE:

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

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)

00	000	(BX) + (SI)	DS
00	001	(BX) + (DI)	DS
00	010	(BP) + (SI)	SS
00	011	(BP) + (DI)	SS
00	100	(SI)	DS
00	101	(DI)	DS
00	110	DISP16 (DIRECT ADDRESS)	DS
00	111	(BX)	DS
01	000	(BX) + (SI) + DISP8	DS
01	001	(BX) + (DI) + DISP8	DS
01	010	(BP) + (SI) + DISP8	SS
01	011	(BP) + (DI) + DISP8	SS
01	100	(SI) + DISP8	DS
01	101	(DI) + DISP8	DS
01	110	(BP) + DISP8	SS
01	111	(BX) + DISP8	DS
10	000	(BX) + (SI) + DISP16	DS
10	001	(BX) + (DI) + DISP16	DS
10	010	(BP) + (SI) + DISP16	SS
10	011	(BP) + (DI) + DISP16	SS
10	100	(SI) + DISP16	DS
10	101	(DI) + DISP16	DS
10	110	(BP) + DISP16	SS
10	111	(BX) + DISP16	DS
11	000	REG AX / AL	
11	001	REG CX / CL	
11	010	REG DX / DL	
11	011	REG BX / BL	
11	100	REG SP / AH	
11	101	REG BP / CH	
11	110	REG SI / DH	
11	111	REG DI / BH	

FLAGS REGISTER CONTAINS:

X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

*The marked encodings are NOT generated by the language translators. If however, the 8087 encounters one of these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)

(2) FCOM ST(i)

(3) FCOMP ST(i)

(4) FXCH ST(i)

(5) FCOMP ST(i)

(6) FFREE ST(i) and pop stack

(7) FXCH ST(i)

(8) FSTP ST(i)

(9) FSTP ST(i)

iAPX 86/88/186 INSTRUCTION SET MATRIX

Hi | Lo	0	1	2	3	4	5	6	7
0	ADD b,f,r/m	ADD w,f,r/m	ADD b,t,r/m	ADD w,t,r/m	ADD b,ia	ADD w,ia	PUSH ES	POP ES
1	ADC b,f,r/m	ADC w,f,r/m	ADC b,t,r/m	ADC w,t,r/m	ADC b,i	ADC w,i	PUSH SS	POP SS
2	AND b,f,r/m	AND w,f,r/m	AND b,t,r/m	AND w,t,r/m	AND b,i	AND w,i	SEG ES	DAA
3	XOR b,f,r/m	XOR w,f,r/m	XOR b,t,r/m	XOR w,t,r/m	XOR b,i	XOR w,i	SEG SS	AAA
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6	PUSHA	POPA	BOUND r,r/m					
7	JO	JNO	JB/ JNAE	JNB/ JAE	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JA
8	Immed b,r/m	Immed w,r/m	Immed b,r/m	Immed is,r/m	TEST b,r/m	TEST w,r/m	XCHG b,r/m	XCHG w,r/m
9	NOP	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI
A	MOV m → AL	MOV m → AX	MOV AL → m	MOV AX → m	MOVS b	MOVS w	CMPS b	CMPS w
B	MOV i → AL	MOV i → CL	MOV i → DL	MOV i → BL	MOV i → AH	MOV i → CH	MOV i → DH	MOV i → BH
C	Shift b,r/m,i	Shift w,r/m,i	RET (i+SP)	RET	LES	LDS	MOV b,i,r/m	MOV w,i,r/m
D	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	IN b	IN w	OUT b	OUT w
F	LOCK		REP	REP Z	HLT	CMC	Grp 1 b,r/m	Grp 1 w,r/m

Hi | Lo	8	9	A	B	C	D	E	F
0	OR b,f,r/m	OR w,f,r/m	OR b,t,r/m	OR w,t,r/m	OR b,i	OR w,i	PUSH CS	
1	SBB b,f,r/m	SBB w,f,r/m	SBB b,t,r/m	SBB w,t,r/m	SBB b,i	SBB w,i	PUSH DS	POP DS
2	SUB b,f,r/m	SUB w,f,r/m	SUB b,t,r/m	SUB w,t,r/m	SUB b,i	SUB w,i	SEG CS	DAS
3	CMP b,f,r/m	CMP w,f,r/m	CMP b,t,r/m	CMP w,t,r/m	CMP b,i	CMP w,i	SEG DS	AAS
4	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6	PUSH iw	IMUL i,w,r/m	PUSH is	IMUL is,r/m	INS b	INS w	OUTS b	OUTS w
7	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	MOV b,f,r/m	MOV w,f,r/m	MOV b,t,r/m	MOV w,t,r/m	MOV sr,f,r/m	LEA	MOV sr,t,r/m	POP r/m
9	CBW	CWD	CALL l,d	WAIT	PUSHF	POPF	SAHF	LAHF
A	TEST b,i	TEST w,i	STOS b	STOS w	LODS b	LODS w	SCAS b	SCAS w
B	MOV i → AX	MOV i → CX	MOV i → DX	MOV i → BX	MOV i → SP	MOV i → BP	MOV i → SI	MOV i → DI
C	ENTER iw,b	LEAVE	RET l,(i+SP)	RET l	INT Type 3	INT (Any)	INTO	IRET
D	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	CALL d	JMP d	JMP l,d	JMP si,d	IN v,b	IN v,w	OUT v,b	OUT v,w
F	CLC	STC	CLI	STI	CLD	STD	Grp 2 b,r/m	Grp 2 w,r/m

where

mod r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	SHL/SAL	SAR
Grp 1	TEST		NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2	INC	DEC	CALL id	CALL l,id	JMP id	JMP l,id	PUSH	

b	=	byte operation
d	=	direct
f	=	from CPU reg
i	=	immediate
ia	=	immed. to accum.
ib	=	immediate byte
id	=	indirect
is	=	immed. byte, sign ext.
iw	=	immediate word
l	=	long ie. intersegment
m	=	memory
r	=	register
r/m	=	EA is second byte
si	=	short intrasegment
sr	=	segment register
t	=	to CPU reg
v	=	variable
w	=	word operation
z	=	zero

- 186 only instruction

F

Example Macros

This appendix presents some example macros. These macros are designed to support the writing of ASM86 routines that will be linked to other modules for the SMALL model of computation (see *An Introduction to ASM86*). The intent here is not to show the full power of MPL. Instead, it is to demonstrate a practical use for macros in a common programming situation.

These macros could be built into an include file. If you were developing a large set of ASM86 modules, you could use this include file at the beginning of each of your modules to define a common interface between the modules (in this case SMALL). Similar sets of macros could be defined to support other models of computation.

```
;A SET OF MACROS TO SUPPORT THE SMALL MODEL OF COMPUTATION
;TO BE USED AS AN INCLUDE FILE

$NOLIST

;THIS MACRO WILL GENERATE A PUBLIC SEGMENT STATEMENT WITH
;A NAME AS A PARAMETER

%*DEFINE (SEG(NAME)) (%NAME  SEGMENT  PUBLIC  '%NAME')

;THESE MACROS ARE USED TO GENERATE THE SEGMENT DIRECTIVES
;FOR THE SMALL MODEL

;CODE SEGMENT

%*DEFINE (CSEG)  (%SEG(CODE))

%*DEFINE (CEND)  (CODE ENDS)

;DATA SEGMENT

%*DEFINE (DSEG)  (%SEG(DATA))

%*DEFINE (DEND)  (DATA ENDS)

;CONST SEGMENT

%*DEFINE (CONSEG)  (%SEG(CONST))

%*DEFINE (CONEND)  (CONST ENDS)

;MEMORY SEGMENT

%*DEFINE (MEMSEG)  (MEMORY  SEGMENT  MEMORY  'MEMORY')

%*DEFINE (MEMEND)  (MEMORY ENDS)

;THIS MACRO WILL DEFINE A STACK SEGMENT. THE NUMBER OF
;WORDS TO RESERVE FOR THE STACK IS PASSED AS A PARAMETER.

%*DEFINE (STACKSEG(LENGTH))  (STACK  SEGMENT  STACK 'STACK'

DW  %LENGTH  DUP  (?)
STACK ENDS)

;THE FOLLOWING MACRO WILL GENERATE THE CODE TO INITIALIZE
;A SEGMENT REGISTER. IT WILL USE THE AX REGISTER.

%*DEFINE (INIT(SEGREG, SEGBASE))  (MOVE AX, %SEGBASE
                       MOV  %SEGREG, AX)

;THE FOLLOWING MACROS GENERATE THE PROLOGS AND EPILOGS USED
;AT THE BEGINNING AND ENDINGS OF PROCS.

%*DEFINE (PROLOG)  (PUSH BP
                        MOV  BP, SP
)
;NO PARAMETERS

%*DEFINE (EPILOG)  (POP BP
                       RET
)
;PARAMETERS TO BE POPPED OFF THE STACK

%*DEFINE (EPI(PARMBYTECOUNT))  (POP  BP
                        RET %PARMBYTECOUNT
)

$LIST

;GROUP DECLARATIONS FOR THE SMALL MODEL

CGROUP  GROUP  CODE

DGROUP  GROUP  DATA, CONST, STACK, MEMORY

ASSUME  CS:CGROUP, DS:DGROUP, SS:DGROUP, ES:DGROUP

;END OF INCLUDE FILE
```

The following is an example source file that uses these macros.

```
;AN EXAMPLE SOURCE FILE USING THE SMALL MODEL MACRO
;INCLUDE FILE

$INCLUDE SMALL.LIB

%DSEG
    .
    .            ;some data
    .
%DEND

%CONSEG
    .
    .            ;constant definitions
    .
%CONEND

;reserve 10 words of stack

%STACKSEG(10)
```

```
%CSEG
    .
    .

APROC           PROC  NEAR
                %PROLOG
                .
                .           ;code goes here
                .
                %EPILOG
APROC           ENDP

XPROC           PROC  NEAR
                %PROLOG
                .
                .           ;code goes here
                .
                %EPI(6)     ;pop 6 bytes of parameters
XPROC           ENDP
    .
    .
    .
%CEND

END
```

The above source module would expand to the following form:

```
;AN EXAMPLE SOURCE FILE USING THE SMALL MODEL MACRO
;INCLUDE FILE

$INCLUDE SMALL.LIB

;A SET OF MACROS TO SUPPORT THE SMALL MODEL OF COMPUTATION
;TO BE USED AS AN INCLUDE FILE

$NOLIST

;GROUP DECLARATIONS FOR THE SMALL MODEL

CGROUP  GROUP  CODE

DGROUP  GROUP  DATA, CONST, STACK, MEMORY

ASSUME  CS:CGROUP, DS:DGROUP, SS:DGROUP, ES:DGROUP

;END OF INCLUDE FILE

DATA  SEGMENT  PUBLIC  'DATA'
    .
    .           ;some data
    .
DATA ENDS

CONST  SEGMENT  PUBLIC  'CONST'
    .
    .           ;constant definitions
    .
CONST ENDS
```

```
;reserve 10 words of stack

STACK  SEGMENT  STACK  'STACK'

DW  10  DUP  (?)

STACK  ENDS

CODE  SEGMENT  PUBLIC  'CODE'
    .
    .

APROC            PROC  NEAR
                 PUSH  BP
                 MOV  BP, SP
                 .
                 .           ;code goes here
                 .
                 POP  BP
                 RET
APROC            ENDP

XPROC            PROC  NEAR
                 PUSH  BP
                 MOV  BP, SP
                 .
                 .           ;code goes here
                 .
                 POP BP      ;pop 6 bytes of parameters
                 RET  6
XPROC            ENDP
    .
    .
CODE ENDS

END
```

Example Programs G

In this Appendix, several sample programs are presented, each with several solutions.

The first two examples illustrate transferring control to one of eight routines, depending on which bit of the accumulator has been set to 1 (by earlier instructions, not shown).

Examples 3, 4, and 5 discuss additional methods of passing data and parameters to procedures, illustrating the use of both the registers and the stack for passing parameters. Examples 6 and 7 cover multibyte addition and subtraction. Interrupt procedures and timing loops are described in examples 8 and 9. Examples 10-13 illustrate input/output control.

The 8086 code examples given here are not optimal, and the presentation is not an attempt at an exhaustive and complete overview of the language. These examples are presented more as a gradual method of building familiarity, perhaps suggestive of further improvements, rather than as ideal, finished models. Some instruction usage is not introduced until the need for it has been suggested by the discussion of prior code.

Examples 1 and 2

Consider a program that executes one of eight routines depending on which bit of the accumulator is set:

Jump to routine 1 if the accumulator holds 00000001
Jump to routine 2 if the accumulator holds 00000010
Jump to routine 3 if the accumulator holds 00000100
Jump to routine 4 if the accumulator holds 00001000
Jump to routine 5 if the accumulator holds 00010000
Jump to routine 6 if the accumulator holds 00100000
Jump to routine 7 if the accumulator holds 01000000
Jump to routine 8 if the accumulator holds 10000000

MAIN PROGRAM BRANCH TABLE PROGRAM JUMP ROUTINES

(normal procedure return sequence not provided by branch table program)

Example 1 below is a routine which transfers control to one of the eight possible procedures depending on which bit of the accumulator is 1.

It moves the low-order bit of the accumulator into a flag register to find the one signalling the correct routine, and then transfers based on that flag. This routine uses seven instructions, including a test to prevent an infinite loop and an indirect transfer via register BX.

Example 2 achieves the same transfer using a different technique for selecting the appropriate address. It shifts the high-order bit of AL, and uses register SI as an index into the branch table.

Each example contains comments, and is followed by a brief explanation.

Example 1:

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

BRANCH_ADDRESSES ENDS

        PROCEDURE_SELECT SEGMENT

        ASSUME   CS:PROCEDURE_SELECT,
&                DS:BRANCH_ADDRESSES

        MOV      BX,BRANCH_ADDRESSES
        MOV      DS,BX                ;moves above segment
                                      ;base-address into
                                      ;segment register DS.
        CMP      AL,0                 ;this test assures that
        JE       CONTINUE_MAIN_LINE   ;some bit of AL has been
                 ;set by earlier instructions to specify
                 ;a routine (prior insts. not shown).

        LEA      BX,BRANCH_TABLE_1    ;BX set to location holding
                                      ;address of first routine.
L:      SHR      AL,1                 ;puts least-significant bit
                                      ;of AL into the carry flag
                                      ;(CF).
        JNC      NOT_YET              ;if CF = 0, the ON bit
                                      ;in AL has not yet
                                      ;been found.
        JMP WORD PTR  [BX]            ;if CF = 1, then control
                                      ;is transferred (see
                                      ;explanation below).

NOT_YET:ADD   BX, TYPE BRANCH_TABLE_1 ;if no transfer, then
                                      ;the bit that is ON has
                                      ;not yet been found, so
                                      ;BX is set to ooint to
                                      ;the next entry in the
                                      ;address-table, by adding 2.
        JMP      L                    ;jump to L to shift
                                      ;and retest
CONTINUE_MAIN_LINE:                   ;we reach here only
                    .                 ;if no bit was set to
                    .                 ;indicate a desired
                    .                 ;routine
        ROUTINE_1:
                    .
                    .
                    .
        ROUTINE_2:
                    .
                    .
                    .
        ROUTINE_3:
                     .
                     .
                     .
PROCEDURE_SELECT ENDS
```

The line after "L:", JNC NOT YET, reads "jump if no carry", which means jump if CF = 0. This will skip over the next line's transfer if the "1" bit, signalling the desired procedure, has not yet appeared. If it has been found, CF will be I and this conditional jump JNC will be skipped. The appropriate procedure is then reached by the indirect jump instruction JMP WORD PTR [BX].

A jump is always to an address in the code segment, i.e., relative to CS. The offset defining that address in the code segment is not given explicitly here. Instead, an indirect JMP is used, with [BX] given as a pointer to the memory location where that offset is stored.

Register BX as used here within square brackets automatically refers to the contents of a location in the data segment. The contents of that location are the desired offset for the jump. In other words, the Instruction Pointer is replaced by the contents of a location in the data segment, whose offset is in BX. The next instruction, ADD BX, TYPE BRANCH _TABLE .1, adds 2 to BX, the index into the branch table. This causes BX to point to the next word of the table. The contents of that word are the offset of the "next" routine, again in the code segment.

Example 2:

```
BRANCH_ADDRESSES  SEGMENT
     BRANCH_TABLE_1      DW  ROUTINE_1
                         DW  ROUTINE_2
                         DW  ROUTINE_3
                         DW  ROUTINE_4
                         DW  ROUTINE_5
                         DW  ROUTINE_6
                         DW  ROUTINE_7
                         DW  ROUTINE_8
BRANCH_ADDRESSES  ENDS
PROCEDURE_SELECT  SEGMENT
             ASSUME CS:PROCEDURE_SELECT,
&                   DS:BRANCH_ADDRESSES
             MOV    BX,BRANCH_ADDRESSES         ;base-address of
             MOV    DS,BX                       ;segment containing
                                                ;lists
             LEA    BX,BRANCH_TABLE_1           ;base-address of list
                                                ;of branch addresses
             MOV    SI,7*TYPE BRANCH_TABLE_1    ;points initially to
                                                ;last such entry
                                                ;in list
             MOV    CX,8                        ;loop-counter allowing
                                                ;8 shifts maximum
     L:      SHL    AL,1                        ;shifts high-order
                                                ;AL bit into CF
             JNC    NOT_YET                     ;if CF = 0, routine
                                                ;represented by that
                                                ;bit not desired
             JMP    WORD  PTR [BX][SI]          ;if CF = 1, transfer
                                                ;to procedure
                                                ;represented by most
                                                ;recent bit tested
```

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

In Example 2 several elements have changed, though the net result is the same. Instead of being incremented, BX stays constant, pointing to the beginning of the list of branch addresses. SI is used as an index (subscript) within that list.

The number of shifts is controlled by the count register CX, which the LOOP instruction automatically decrements after each iteration. The accumulator AL is searched from its most-significant-bit using the shift-left instruction (SHL) instead of SHR. This accounts for the initialization of SI to 14, pointing initially to the last branch-address in the list, 14 bytes past the base-address in BX. SI is subsequently decremented in each iteration just as Example 1's BX was incremented.

The instruction JMP WORD PTR [BX][SI] uses the sum of BX and SI just as Example 1 used BX alone. That is, the sum gives the offset of a word in the data segment, and the contents of that word replaces the IP. The next instruction executed is thus the one whose code-segment offset was stored in the branch table.

If more than 1 bit were set in AL, these two examples would select different routines due to selecting the rightmost or leftmost such bit.

Transferring Data to Procedures

The data on which a procedure performs its operations may be made available in registers or memory locations. In many applications, however, reserving registers for this purpose can be inconvenient to the system flow of control and uneconomical in execution time, requiring frequent register saves and restores.

Reserving memory, on the other hand, can be uneconomical of space, especially if such data is needed only temporarily. It is often preferable to use and reuse a special area called a stack, storing and deleting interim data and parameters as needed.

Regardless of the method used to pass data to procedures, a stack will be necessary and useful. The CALL instruction uses the stack to save the return address. The RET instruction expects the return address to be on the stack. The stack is also usually used to save the caller's register values at the beginning of a procedure. Then, just before the procedure returns to the caller, these values can be restored.

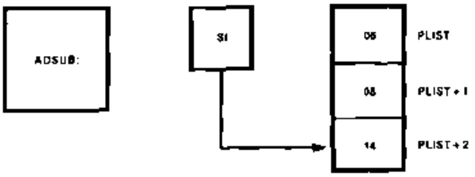
Example 3 shows the use of memory to pass parameters. Registers are used for this in Example 4. Example 5 uses a stack.

One way to use memory to pass data is to place the required elements (called a parameter list) in some data area. You then pass the first address of this area to the procedure.

For example, the following procedure, ADSUB, expects the address of a three-byte parameter list in the SI register. It adds the first and second bytes of the list, and stores the result in the third byte of the list.

The first time ADSUB is called, at label CALL1, it loads the accumulator from PLIST, adds the value from the next byte and stores the result in PLIST+2. Return is then made to the instruction at RET1.

AFTER first call to ADSUB:



The second time ADSUB is called, at label CALL2, the prior instruction has caused the SI register to point to the parameter list LIST2. The accumulator is loaded with 10, 35 is added, and the sum is stored at LIST2 + 2. Return is then made to the instruction at RET2.

Example 3:

```
PARAMS  SEGMENT

PLIST      DB      6
           DB      8
           DB      ?
LIST2      DB      10
           DB      35
           DB      ?
             .
             .
             .
PARAMS  ENDS

STACK   SEGMENT
        DW  4  DUP  (?)
STACK_TOP  LABEL WORD
STACK   ENDS

ADDING  SEGMENT
ASSUME  CS:ADDING,   DS:PARAMS, SS:STACK

START:     MOV       AX,PARAMS
           MOV       DS,AX                ;initialize DS
           MOV       AX,STACK
           MOV       SS,AX                ;initialize SS
```

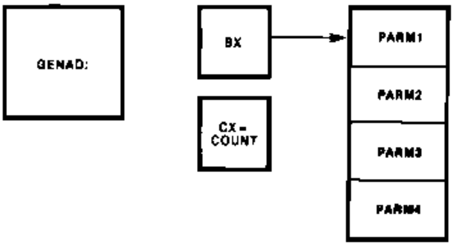
```
             MOV       SP,OFFSET STACK_TOP  ;initialize SP
             MOV       SI,OFFSET PLIST
CALL1:       CALL      ADSUB
RET1:         .
              .
              .
             LEA       SI,LIST2
CALL2:       CALL      ADSUB
 RET2:
              .
              .
              .
              .
              .
ADSUB        PROC
             MOV       AL,[SI]              ;get 1st parameter
             ADD       AL,[SI+1]            ;add 2nd parameter
             MOV       [SI+2],AL            ;store result in
             RET                            ;3rd parameter
ADSUB        ENDP
              .
              .
              .
ADDING   ENDS
         END START
```

The instructions just prior to each CALL load the SI register with the offset of the first parameter to be added. The MOV statement prior to CALL1 makes use of the OFFSET operator (discussed in Chapter 4). If this operator were omitted, SI would receive the contents of PLIST instead of its offset. The LEA instruction prior to CALL2 automatically puts the offset of its source (2nd operand) into the register destination (1st operand). The MOV statement is more efficient, but may only be used if just the offset is being loaded into the register. If the address involves an indexing register (e.g., PLIST [SI + 1]), then the LEA should be used, since this will add the contents of the SI, 1, and the offset of PLIST, putting the sum in the destination register.

A More General Solution

The approach used in Example 3 has its limitations, however. As coded, ADSUB will process a list of two and only two numbers to be added, and they must be contiguous in memory. Suppose you wanted a subroutine (GENAD) which would add an array containing an arbitrary number of bytes, located anywhere in memory, and leave the sum in the accumulator.

CALL to GENAD:



Example 4 below shows how this process can be written in ASM86. GENAD returns the sum in the accumulator. It receives the address of the array in the BX register, and the number of array elements in CX.

Example 4:

```
INITIAL_PARAMETERS            SEGMENT
RESULT     DB   0
PARM       DB   6, 82, 13, 16


INITIAL_PARAMETERS            ENDS

GENERAL_PROCEDURES            SEGMENT
           ASSUME CS:general_procedures  DS:initial_parameters


;The procedure is placed first, to avoid forward
;referencing the FAR procedure GENAD. Note that the
;program ;start address is after the procedure, at label
;"START2.

GENAD      PROC FAR
           PUSH SI            ;save current value of SI on the
                              ;stack (discussed below), so that
                              ;this routine can use this
                              ;register freely, restoring its
                              ;original contents just prior
                              ;to returning control to
                              ;calling routine.
INIT:      MOV  AL, 0         ;initialize AL to receive sum.
           MOV  SI, 0         ;initialize SI to point to first
                              ;array element

MORE?:     ADD  AL, [BX][SI]  ;add next array element to sum.
                              ;BX points to the start of the
                              ;array, and SI selects an element
                              ;of the array.

           INC  SI            ;have SI index the next
                              ;array element.
           LOOP MORE?         ;continue looping until CX is
                              ;zero (all array elements have
                              ;been added into AL)

           POP  SI            ;restore original contents of SI.
           RET                ;transfer to instruction
                              ;immediately following CALL.

GENAD      ENDP


;Program execution starts here (due to the label "start"
;named on the END directive below). Point DS to the
;INITIAL_PARAMETERS segment, and call GENAD with the array
;PARM.
```

```
START:      MOV  AX, INITIAL_PARAMETERS
            MOV  DS, AX

            MOV  CX, SIZE PARM      ;number of elements is
                                    ;passed in CX
            MOV  BX, OFFSET PARM    ;address of array PARM is
                                    ;passed in BX.
            CALL GENAD
            MOV  RESULT, AL         ;Sum is returned in AL

            HLT                 ;******* end of program *******
GENERAL_PROCEDURES              ENDS
            END  START
```

In GENAD, the first action is to save (PUSH) onto the stack the current value of SI before using it. Just before the RETurn, this value is restored (via POP). Thus this procedure does not destroy the status of registers (except AL and CX) possibly relied upon by the calling routine. Stacks are discussed in Chapter 4. Further examples appear below.

The routine does not explicitly save the value of CS because the CALL and RETurn save CS on the stack and restore it automatically. The accumulator AL is here expected to be usable without saving its pre-CALL contents. Using AL, the sum is modulo 256.

The FAR type declaration on the PROC statement forces the use of "long" CALLs to and RETurns from this procedure. This means the procedure is not expected to be in the same segment as all of the CALLs to it. In a "long" CALL the contents of CS are PUSHed onto the stack first, then the IP is PUSHed onto the stack. (This allows an eventual return to the next sequential instruction.) Control is then transferred to the procedure by first moving into CS the segment base address for the procedure, and then replacing the contents of IP with the offset of the procedure in that segment. A "long" RETurn reverses this process by POPping the former IP contents back off the stack into IP, and then POPping the former CS contents off the stack back into CS.

Within the inner body of GENAD, the statement

```
MOV  AL,0
```

initializes the sum to zero. The statement

```
MOV  SI,0
```

initializes SI to zero, to index the first element of the passed array.

The first statement in the loop

```
ADD  AL, [BX] [SI]
```

adds the array element indexed by SI into the sum in the accumulator (recall that the BX register points to the parameter array). In the next statement (INC SI), the array index in SI is incremented to point to the next array element. The last statement in the loop

```
LOOP  MORE?
```

executes the loop repeatedly until the count in CX (passed in as a parameter) is exhausted.

Using a Stack

Passing parameters on the stack offers different advantages than passing them in registers. Passing parameters in registers is faster, but more complicated. The conventions as to which parameter should end up in which register can be confusing, especially if there are many procedures.

For parameters passed on the stack, the convention need only specify the order they should be pushed onto the stack. High level language compilers (e.g., PL/M-86) generate code which passes parameters on the stack. Therefore, any procedure which expects its parameters on the stack is callable from PL/M (see the ASM86 macro assembler manual for more details). The 8086 also offers special instructions to facilitate using the stack for passing parameters. The RET instruction has an optional byte count (e.g., RET 4), which says how many bytes should be popped off the stack in addition to the return address. This makes returning from procedures very easy. Moreover, since the BP indexing-register uses the SS segment by default, it is very economical to use BP to reference data near the top of the stack.

Use of stacks may require some further introduction. A stack segment is expected to be used relative to the contents of the stack-segment register SS, just as a code segment uses CS and data segments use DS or ES. The stack segment below is defined for use in this discussion and the examples.

```
PARAMS_PASS  SEGMENT STACK
             DW 12 DUP (0)
LAST_WORD    LABEL WORD
PARAMS_PASS ENDS
```

Four instructions use a stack in predefined ways: PUSH, CALL, POP, and RETurn. They automatically use the stack pointer SP as an offset to the segment-base-address in SS. One of your first actions in a module which will use a stack must be to initialize SS and SP, e.g.,

```
MOV  AX,PARAMS_PASS
MOV  SS,AX
MOV  SP, OFFSET LAST_WORD
```

This use of LAST .WORD is critically important due to the built-in actions of the four instructions named above.

The first two, PUSH and CALL, store additional words on the stack by *decrementing* SP by 2. Thus the stack "grows downward" from the last word in the stack segment toward the segment-base-address lower in memory. Each successive address used for new data on the stack is a lower number. The location pointed to by SP is called the Top Of Stack (TOS). When a word is stored on the stack, e.g., by the instruction

```
PUSH  SOURCE_DATA
```

SP is decremented by 2 and the source data is moved onto the stack at the new offset now in SP. As described above in Example 4, CALL implicitly uses PUSH before transferring control to a procedure.

The instruction

```
POP  DESTINATION
```

takes the word at the "top-of-stack", i.e., pointed at by SP, and moves that word into the specified destination. POP also then automatically *adds* 2 to SP. This causes SP to point to the next higher-addressed word in the stack segment, farther from the segment's base-address. The figures accompanying the examples below show the expansion and contraction of a stack.

Example 5 below illustrates the use of a stack to pass the number of byte parameters plus the address of the first one. For this example all the parameters are expected in successive bytes after that one.

Supplying the Number of Parameters and the First Address, On the Stack

Example 5:

```
params_pass    SEGMENT STACK
               DW   12   DUP  (?)     ;reserve 12 words of
                                      ;stack space
last_word      LABEL  WORD            ;last_word is the
                                      ;offset of top of
                                      ;stack
params_pass    ENDS

data_items     SEGMENT

first  DB  11, 22, 33, 44, 55, 66
second DB  4, 5, 6
third  DB  94, 88
result DX  ?
data_items     ENDS

stk_usage_xmpl   SEGMENT
                 ASSUME CS: stk_usage_xmpl, DS: data_items, SS:params_

genaddr  PROC    FAR

         PUSH   BP                    ;save old copy of BP
         PUSH   BP, SP                ;move tos to BP (see
                                      ;figure 4)
         PUSH   BX                    ;save BX, so ok to use BX in
                                      ;genaddr
         PUSH   CX                    ;save CX, so ok to use CX in
                                      ;genaddr (figure 5)
         MOV    CX, [BP + 8]          ;get count of number of bytes
                                      ;in array
         MOV    BX, [BP + 10]         ;get address of array of
                                      ;bytes

         MOV    AX, 0                 ;AX := 0. AX holds running
                                      ;sum in adder loop.
adder:   ADD    AL, [BX]              ;add in the first byte
         ADC    AH, 0                 ;and add any carry into AH.
         INC    BX                    ;point to next byte to be
                                      ;added in.
         LOOP   adder                 ;CX := CX - 1;IF CX <> 0 THEN
                                      ;GOTO ADDER;

         POP    CX                    ;The registers must be
                                      ;restored in the
         POP    BX                    ;reverse order they were
                                      ;pushed.
         POP    BP
         RET    4                     ;return, popping off the 2
                                      ;WORD parameters
```

```
genaddr  ENDP

stk_usage_xmpl ENDS

caller          SEGMENT
                ASSUME CS: caller, DS: data_items, SS: params_pass

start:   MOV     AX, data_items       ;paragraph number of
                                      ;data segment to AX
         MOV     DS, AX               ;and then to DS.
         MOV     AX, params_pass      ;paragraph number of
                                      ;stack segment to AX
         MOV     SS, AX               ;and then to SS
         MOV     SP, OFFSET last_word ;offset of the
                                      ;stack_top to the SP

         MOV     AX, OFFSET first     ;offset of first to
                                      ;AX
         PUSH    AX                   ;then onto the stack
         MOV     AX, SIZE first       ;number of bytes in
                                      ;first array to AX
         PUSH    AX                   ;then onto the stack
         CALL    genaddr              ;Call the far
                                      ;procedure
         MOV     result,AX
                 .
                 .
                 .
         MOV     AX, OFFSET second
         PUSH    AX
         MOV     AX, SIZE second
         PUSH    AX                   ;same as above except
                                      ;doing second
         CALL    genaddr
         MOV     result,AX
                 .
                 .
                 .
         MOV     AX, OFFSET third
         PUSH    AX
         MOV     AX, SIZE third       ;same as above except
                                      ;doing third
         PUSH    AX
         CALL    genaddr
         MOV     result,AX
                 .
                 .
         HLT
caller          ENDS
                END  start
```

To indicate why each register was saved, the above code has each PUSH placed just prior to the first local use of that register. Earlier examples clustered those PUSHes at the top of the routine, just as the POPs appear (in reverse order) at the end. This makes it easy to see the proper order of saving and restoring. In either case you must consider carefully where the parameters are relative to the pointer you are using, e.g., BP. Making your own diagrams can help.

Note that the RET instruction of "genaddr" is a RET 4; the two parameters are popped off the stack as the RETurn is executed. Without the 4, this 12 word stack named "PARAMS_ PASS" could only be used three times. The fourth call would cause two words outside that segment to be clobbered.

This is why: prior to each call the parameter words are pushed onto the stack. Then each call uses two words of the stack to store the return address. Each execution of the procedure pushes three more words onto the stack to preserve register values. These last five words are popped off by the procedure's end and return, but those first two parameters would remain.

Multibyte Addition and Subtraction

The carry flag and the ADC (add with carry) instructions may be used to add unsigned data quantities of arbitrary length. Consider the following addition of two three-byte unsigned hexadecimal numbers:

```
 32AF8A
+84BA90
-------
 B76A1A
```

To perform this addition, you can use ADD or ADC to add the low-order byte of each number. ADD sets the carry flag for use in subsequent instructions, but does not include the carry flag in the addition.

Step 3		Step 2		Step 1
32		AF		8A
84		BA		90
B7		6A		1A
	carry=1		carry=1	

The routine below performs this multibyte addition, making these assumptions:

The numbers to be added are stored from low-order byte to high-order byte beginning at memory locations FIRST and SECOND, respectively.

The result will be stored from low-order byte to high-order byte beginning at memory location FIRST, replacing the original contents of these locations.

MEMORY BEFORE	MEMORY AFTER	
FIRST + SECOND + CF	FIRST	SECOND
8A + 90 + 0 – 1A	1A	90
AF + BA + 1 = 6A	6A	BA
32 + 84 + 1 = B7	B7	84

The routine uses an ADC instruction to add the low-order bytes of the operands. This could cause the result to be high by one if the carry flag were left set by some previous instruction. This routine avoids the problem by clearing the carry flag with the CLC instruction just before LOOPER.

Since none of the instructions in the program loop affect the carry flag except ADC, the addition with carry will proceed correctly.

```
MULTI_TWO   SEGMENT

ASSUME        CS:MULTI_TWO,
&             DS:ADD_DATA_2

START:        MOV  AX,ADD_DATA_2
              MOV  DS,AX

;The routine determines which number is longer and stores
;the result there. The size in bytes of the smaller number
;controls LOOP1, i.e., where both numbers do have a byte
;of data to be added.
;The difference in size controls LOOP2, which is needed if
;there is a final carry.

            MOV  AX,   NUM2            ;Initially assume NUM2
                                       ;larger, and
            LEA  BX,   SECOND          ;give BX address of
                                       ;longer number,
            LEA  BP,   FIRST           ;BP address of shorter
                                       ;number.

            CMP  AX,   NUM1            ;Check assumption.
            JGE  NUM2_BIGGER           ;continue with values
                                       ;as they are unless N2
                                       ;> N1.

            XCHG AX,   NUM1            ;Switch NUM2 and NUM1,
                                       ;exchanging
            XCHG AX,   NUM2            ;through AL NUM2 now <
                                       ;NUM1.

            XCHG BX,   BP              ;Must also now switch
                                       ;addresses referred to,
                                       ;so that number of
                                       ;bytes still
                                       ;corresponds with
                                       ;correct number,
                                       ;and sum goes
                                       ;to longer place.

NUM2_BIGGER:MOV  CX,   NUM2
            SUB  CX,   NUM1            ;NUM2 now gets
                                       ;difference

            MOV  NUM2, CX
            MOV  CX,   NUM1            ;of sizes. Use smaller
                                       ;number of bytes for
                                       ;central add.
            CLC                        ;Clear carry of
                                       ;possible prior
                                       ;setting
            MOV  SI,   0               ;Initialize index to
                                       ;bytes of addends. Then
                                       ;SI=SI+1.
LOOP1:      MOV  AL,   DS: [BP] [SI]   ;Get byte of shorter
                                       ;number.

            ADC  [BX] [SI, AL          ;Add it to relevant
                                       ;byte of
            INC  SI                       ;longer number. Then
                                          ;SI=SI+1
            LOOP LOOP1                    ;

            MOV  CX,   NUM2               ;Number of bytes yet
                                          ;unused in longer
                                          ;number.

LOOP2:      JNB  DONE                     ;If no carry, CF=0,
                                          ;then done.
            ADC  BYTE PTR [BX] [SI],0     ;Add carry to remaining
                                          ;bytes
            INC  SI                       ;of longer number. Then
                                          ;SI=SI+1.

            LOOP LOOP2
DONE:        .
             .                            ;
             .                            ;

MULTI_TWO   ENDS
            END  START
```

With some additional instructions, this same routine will do arithmetic for packed-decimal numbers. Packed-decimal means the 8 bits of each byte are interpreted as 2 decimal digits, e.g., 01100111B would mean 67 decimal instead of 67 hexadecimal (103 decimal).

Below is the core of an 8086 routine to do decimal subtraction for packed-decimal numbers.

Example 7:

```
        MOV  SI, 0
        MOV  CX, NUMBYTES
        CLC
MORE?:  MOV  AL, FIRST [SI]
        SBB  AL, SECOND [SI]
        DAS
        MOV  SECOND [SI], AL
        INC  SI
        LOOP MORE?
```

Interrupt Procedures

Example 8:

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

```
;rather in this hypothetical control application. The
;procedures merely illustrate the diverse possibilities
;for handling situations of varying importance and
;urgency.

ASSUME  CS:INTERRUPT_PROCEDURES, DS:DATA_VAR

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

               DD   ALARM_1    ;non-maskable interrupt
                               type 2

;One 64K area of memory contains pointers to the routines
;that handle interrupts. This area begins at absolute
;address zero. The address for the routine appropriate
;to each interrupt type is expected as the contents of the
;double word whose address is 4 times that type. Thus the
;address for the handler of non-maskable-interrupt type 2
;is stored as the contents of absolute location 8. These
;addresses are also called interrupt vectors since they
;point to the respective procedures.
;The first 32 interrupt types (0-31) are defined or
;reserved by INTEL for present and future uses. (See the
;8086 User's Manual for more detail.) User-interrupt type
;32 must therefore use location 128 (=80H) for its
;interrupt vector.

               ORG  08H

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

INTERRUPT_PROC_TABLE   ENDS

DATA_VAR   SEGMENT   PUBLIC

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

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

DATA_VAR  ENDS
```

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

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

INTERRUPT_PROCEDURES   SEGMENT

ALARM_1   PROC   FAR

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

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

ALARM_1   ENDP

ALARM_2   PROC   FAR

        PUSH  DX
        PUSH  AX
        MOV   DX,   WARNING_LIGHTS
        MOV   AL,   1                   ;turn on warning light #1
        OUT   DX,AL                     ;to warn operator of device

        MOV   ALARM_FLAG, 0FFH          ;set alarm flag to inhibit
        POP   AX                        ;later processes which may
                                        ;now be dangerous
        POP   DX
        IRET                            ;return from interrupt:
                                        ;this restores the flags
                                        ;and returns control
                                        ;the interrupted
                                        ;instruction stream


ALARM_2   ENDP

DEVICE_1   PROC

        PUSH  DX
        PUSH  AX
        MOV   DX, DEVICE_1_PORT
        IN    AL,DX                     ;get input byte from
        MOV   INPUT_1_VAL, AL           ;device_ store value

        MOV   INPUT_FLAG,2              ;this may alert another
                                        ;routine or device that
                                        ;this interrupt and input
                                        ;occurred
        POP   AX
        POP   DX
        IRET

DEVICE_1   ENDP

DEVICE_2   PROC

        PUSH  DX                    ;when this interrupt-type
        PUSH  AX                    ;occurs, the action necessary
                                    ;is to notify device_2_port
                                    ;of the event

        MOV   AL,    OUTPUT_2_VAL  ;get value, to output
        MOV   DX,    DEVICE_2_PORT ;to device_2_port
        OUT   DX,AL
        POP   AX
        POP   DX
        IRET

DEVICE_2   ENDP

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

        POP   DX
        IRET

DEVICE_3   ENDP

DEVICE_4   PROC

        PUSH  DX
        PUSH  CX                    ;a device_4 interrupt
        PUSH  AX                    ;orovides a value which
        MOV   DX, DEVICE_4_PORT     ;needs immediate
                                    ;conversion by another
        IN    AL,DX                 ;procedurebefore this
        MOV   CL, AL                ;interrupt-handler can allow
                                    ;it to be used at input_4_val

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

        POP   AX
        POP   CX
        POP   DX
        IRET

DEVICE_4   ENDP

INTERRUPT_PROCEDURES   ENDS

                    END
```

Timing Loop

Example 9:

```
;This example is a procedure for supplying timing loops
;for a program. The amount of time delayed is set by a
;byte parameter passed in the AL register, with the amount
;of time = PARAM * 100 microseconds. This is assuming that
;the 8086 is running at 8 MHZ.

ASSUME CS:TIMER_SEG

TIMER_SEG    SEGMENT

TIME         PROC

DELAY_LOOP:  MOV  CL, 78H   ;shift count for supplying
             SHR  CL,CL     ;proper delay via SHR countdown
             DEC  AL        ;decrement timer count
             JNZ  DELAY_LOOP

             RET
TIME         ENDP
TIMER_SEG    ENDS
             END
```

I/O Routines

The examples below (10-13) illustrate the type of procedures used by the SDK86 Serial I/O Monitor to communicate with the keyboard and display units during execution.

The first, SIO_CHAR_RDY, tests whether an input character is awaiting processing.

The second SIO_OUT CHAR, outputs a character unless SIO_CHAR_RDY reports an input character is there, which is handled first.

The third, SIO_OUT_ STRING, puts out an entire string of characters, e.g., a page heading, using SIO OUT CHAR for each output byte.

Example 10:

```
SIO_CHAR_RDY    PROC    NEAR

   PUSH BP          ;save old value
   MOV  BP, SP

   MOV  DX, 0FFF2H  ;address of status port to DX
   IN   AL,DX       ;input from status port
   TEST AL, 2H      ;is read-data-ready line=1,
                    ;i.e., character pending?
   JNZ  READY       ;if so, return TRUE

   MOV  AL, 0       ;if not, return FALSE: AL=0
   POP  BP          ;restore old value
   RET              ;done, no char waiting
```

```
READY:

   MOV  AL, OFFH    ;return TRUE: AL=all ones
   POP  BP          ;restore old value
   RET              ;done, char is waiting

SIO_CHAR_RDY   ENDP
```

Example 11:

The above procedure also appears in this example, which introduces names for some of the specific numbers used above, and for some that will be used in later examples. These names can make it easier to read the procedure and understand what is going on, or at least what is intended.

The example also uses BX and reorders the code to save a few bytes.

```
       TRUE  EQU  OFFH
      FALSE  EQU  OH
STATUS_PORT  EQU  OFFF2H
  DATA_PORT  EQU  OFFFOH
 ASCII_MASK  EQU  7FH
  CONTROL_S  EQU  13H
  CONTROL_Q  EQU  11H
   CARR_RET  EQU  ODH

SIO_CHAR_RDY2   PROC   NEAR

             PUSH  BX                ;save old BX value
             MOV   BL, TRUE          ;prepare for one result
             MOV   DX, STATUS_PORT   ;check the facts
             IN    AL,DX             ;char waiting???
             TEST  AL, 2H            ;if 2nd bit ON, char is
             JNZ   RESULT            ;waiting hence skip over
             MOV   BL, FALSE         ;FALSE set-up here if 2nd
                                     ;bit was off, hence no
                                     ;char waiting
RESULT:      MOV   AL, BL            ;AL receives whichever
             POP   BX                restore old BX value
             RET                     ;

SIO_CHAR_RDY2   ENDP
```

Example 12:

```
SIO_OUT_CHAR   PROC   NEAR

;This routine outputs an input parameter to the USART
;output port when UART is ready for output transmit
;buffer empty. The input to this routine is on the stack.

        PUSH  BP
        MOV   BP, SP

        CALL  SIO_CHAR_RDY     ;keyboard input pending?
        RCR   AL, 1            ;put low-byte into CF to test
        JNB   OUTPUT           ;if no input char waiting from
                               ;keyboard, go to output loop
        MOV   DX, DATA_PORT   ;char waiting: get it
        IN    AL,DX           ;char to AL from that port
                              ;strip off high bit, leaving
        AND AL, ASCII_MASK    ;ASCII code
        MOV   CHAR, AL        ;save char
        CMP   AL, CONTROL_S   ;is char control-S?
        JNZ   OUTPUT          ;if this halt-display signal
                              ;is not rec'd, continue
                              ;output at OUTPUT

CHECK:                        ;if control-S rec'd, must
                              ;await its release
        CMP   CHAR, CONTROL_Q ;Control-Q received?
        JZ    OUTPUT          ;if this continuation-signal
                              ;rec'd, to do next output
        CALL  SIO_CHAR_RDY    ;keep checking for new keyboard
        RCR   AL, 1           ;input, looping from CHECK
        JNB   CHECK           ;to here until input waiting

        MOV   DX, DATA_PORT   ;get waiting character
        IN    AL,DX
        AND   AL, ASCII_MASK
        MOV   CHAR, AL        ;
        CMP   AL, CARR_RET    ;if char=carriage-return,
        JNZ   CHECK           ;skip this instruction, which
                              ;loops to await control-Q, and
        JMP   NEXTCOMMAND     ;go to NEXTCOMMAND

OUTPUT:
CONTINUE:
        MOV   DX, STATUS_PORT ;loop until status port
        IN    AL,DX           ;and transmit line indicate
        TEST  AL, 1           ;ready to put out character
        JZ    OUTPUT          ;

        MOV   DX, DATA_PORT   ;output port address to DX
        MOV   AL, [BP] + 4    ;character from stack to AL
        OUT   DX,AL           ;output character in AL through

        POP   BP              ;restore original BP value
        RET   2               ;repositions SP behind prior
                              ;parameter

SIO_OUT_CHAR   ENDP
```

Example 13:

```
SIO_OUT_STRING   PROC   NEAR

;Outputs a string stored in the ''extra'' segment (uses ES
;as base), the string being pointed to by a 2-word pointer
;on the stack

    PUSH  BP
    MOV   BP, SP
    MOV   SI,0

    LES   BX,  DWORD  PTR  [BP] + 4
```

```
;load ES with base address and BX with offset of string
;(addresses pushed onto stack by calling routine)

CHECK:

    CMP    BYTE PTR ES: [BX] [SI], 0
                                   ;terminator character
                                   ;is ASCII
    JZ     DONE                    ;null = all zeroes if
                                   ;done, exit

    MOV    AL,BYTE PTR ES: [BX] [SI] ;put next char on
    PUSH   AX
    CALL   SIO_OUT_CHAR            ;stack for output by
                                   ;this called procedure

    INC    SI                      ;point index to next
                                   ;char

    JMP    CHECK
DONE:
    POP    BP
    RET    4                       ;after return, resets
                                   ;SP behind former
                                   ;parameters

SIO_OUT_STRING    ENDP
```

186 Instruction Set Summary

H

FUNCTION	FORMAT				186 Clock Cycles	Comments
DATA TRANSFER						
MOV = Move:						
Register to Register/Memory	1 0 0 0 1 0 0 w	mod reg r/m			2/12	
Register/memory to register	1 0 0 0 1 0 1 w	mod reg r/m			2/9	
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1	12–13	8/16-bit
Immediate to register	1 0 1 1 w reg	data	data if w = 1		3–4	8/16-bit
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high		9	
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high		8	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m			2/9	
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m			2/11	
PUSH = Push:						
Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m			16	
Register	0 1 0 1 0 reg				10	
Segment register	0 0 0 reg 1 1 0				9	
Immediate	0 1 1 0 1 0 s 0	data	data if s = 0		10	
PUSHA = Push All	0 1 1 0 0 0 0 0				36	
POP = Pop:						
Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m			20	
Register	0 1 0 1 1 reg				10	
Segment register	0 0 0 reg 1 1 1	(reg ≠ 01)			8	
POPA = Pop All	0 1 1 0 0 0 0 1				51	
XCHG = Exchange:						
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m			4/17	
Register with accumulator	1 0 0 1 0 reg				3	
IN = Input from:						
Fixed port	1 1 1 0 0 1 0 w	port			10	
Variable port	1 1 1 0 1 1 0 w				8	
OUT = Output to:						
Fixed port	1 1 1 0 0 1 1 w	port			9	
Variable port	1 1 1 0 1 1 1 w				7	
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1				11	
LEA = Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m			6	
LDS = Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m	(mod ≠ 11)		18	
LES = Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m	(mod ≠ 11)		18	
LAHF = Load AH with flags	1 0 0 1 1 1 1 1				2	
SAHF = Store AH into flags	1 0 0 1 1 1 1 0				3	
PUSHF = Push flags	1 0 0 1 1 1 0 0				9	
POPF = Pop flags	1 0 0 1 1 1 0 1				8	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

186 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT				186 Clock Cycles	Comments
ARITHMETIC						
ADD = Add:						
Reg/memory with register to either	0 0 0 0 0 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s w = 01	4/16	
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w = 1		3/4	8/16-bit
ADC = Add with carry:						
Reg/memory with register to either	0 0 0 1 0 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s w = 01	4/16	
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w = 1		3/4	8/16-bit
INC = Increment:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m			3/15	
Register	0 1 0 0 0 reg				3	
SUB = Subtract:						
Reg/memory and register to either	0 0 1 0 1 0 d w	mod reg r/m			3/10	
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w = 01	4/16	
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w = 1		3/4	8/16-bit
SBB = Subtract with borrow:						
Reg/memory and register to either	0 0 0 1 1 0 d w	mod reg r/m			3/10	
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w = 01	4/16	
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w = 1		3/4	8/16-bit
DEC = Decrement:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m			3/15	
Register	0 1 0 0 1 reg				3	
CMP = Compare:						
Register/memory with register	0 0 1 1 1 0 1 w	mod reg r/m			3/10	
Register with register/memory	0 0 1 1 1 0 0 w	mod reg r/m			3/10	
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01	3/10	
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w = 1		3/4	8/16-bit
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m			3	
AAA = ASCII adjust for add	0 0 1 1 0 1 1 1				8	
DAA = Decimal adjust for add	0 0 1 0 0 1 1 1				4	
AAS = ASCII adjust for subtract	0 0 1 1 1 1 1 1				7	
DAS = Decimal adjust for subtract	0 0 1 0 1 1 1 1				4	
MUL = Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m				
Register-Byte					26–28	
Register-Word					35–37	
Memory-Byte					32–34	
Memory-Word					41–43	
IMUL = Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m				
Register-Byte					25–28	
Register-Word					34–37	
Memory-Byte					31–34	
Memory-Word					40–43	
IMUL = Integer immediate multiply (signed)	0 1 1 0 1 0 s 1	mod reg r/m	data	data if s = 0	22–25/29–32	
DIV = Divide (unsigned):	1 1 1 1 0 1 1 w	mod 1 1 0 r/m				
Register-Byte					29	
Register-Word					38	
Memory-Byte					35	
Memory-Word					44	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

186 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT				186 Clock Cycles	Comments
ARITHMETIC (Continued)						
IDIV = Integer divide (signed):	1 1 1 1 0 1 1 w	mod 1 1 1 r/m				
Register-Byte					44–52	
Register-Word					53–61	
Memory-Byte					50–58	
Memory-Word					59–67	
AAM = ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0			19	
AAD = ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0			15	
CBW = Convert byte to word	1 0 0 1 1 0 0 0				2	
CWD = Convert word to double word	1 0 0 1 1 0 0 1				4	
LOGIC						
Shift/Rotate Instructions:						
Register/Memory by 1	1 1 0 1 0 0 0 w	mod TTT r/m			2/15	
Register/Memory by CL	1 1 0 1 0 0 1 w	mod TTT r/m			5+n/17+n	
Register/Memory by Count	1 1 0 0 0 0 0 w	mod TTT r/m	count		5+n/17+n	

TTT	Instruction
0 0 0	ROL
0 0 1	ROR
0 1 0	RCL
0 1 1	RCR
1 0 0	SHL/SAL
1 0 1	SHR
1 1 1	SAR

FUNCTION	FORMAT				186 Clock Cycles	Comments
AND = **And:**						
Reg/memory and register to either	0 0 1 0 0 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1	4/16	
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w = 1		3/4	8/16-bit
TEST = **And function to flags, no result:**						
Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m			3/10	
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1	4/10	
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w = 1		3/4	8/16-bit
OR = **Or:**						
Reg/memory and register to either	0 0 0 0 1 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1	4/16	
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w = 1		3/4	8/16-bit
XOR = **Exclusive or:**						
Reg/memory and register to either	0 0 1 1 0 0 d w	mod reg r/m			3/10	
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1	4/16	
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w = 1		3/4	8/16-bit
NOT = Invert register/memory	1 1 1 1 0 1 1 w	mod 0 1 0 r/m			3	
STRING MANIPULATION:						
MOVS = Move byte/word	1 0 1 0 0 1 0 w				14	
CMPS = Compare byte/word	1 0 1 0 0 1 1 w				22	
SCAS = Scan byte/word	1 0 1 0 1 1 1 w				15	
LODS Load byte/wd to AL/AX	1 0 1 0 1 1 0 w				12	
STOS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w				10	
INS = **Input byte/wd from DX port**	0 1 1 0 1 1 0 w				14	
OUTS = **Output byte/wd to DX port**	0 1 1 0 1 1 1 w				14	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

186 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT				186 Clock Cycles	Comments
STRING MANIPULATION (Continued):						
Repeated by count in CX						
MOVS Move string	1 1 1 1 0 0 1 0	1 0 1 0 0 1 0 w			8+8n	
CMPS Compare string	1 1 1 1 0 0 1 z	1 0 1 0 0 1 1 w			5+22n	
SCAS Scan string	1 1 1 1 0 0 1 z	1 0 1 0 1 1 1 w			5+15n	
LODS – Load string	1 1 1 1 0 0 1 0	1 0 1 0 1 1 0 w			6+11n	
STOS – Store string	1 1 1 1 0 0 1 0	1 0 1 0 1 0 1 w			6+9n	
INS = Input string	1 1 1 1 0 0 1 0	0 1 1 0 1 1 0 w			8+8n	
OUTS = Output string	1 1 1 1 0 0 1 0	0 1 1 0 1 1 1 w			8+8n	
CONTROL TRANSFER						
CALL – Call:						
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high		14	
Register memory indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m			13/19	
Direct intersegment	1 0 0 1 1 0 1 0	segment offset			23	
		segment selector				
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		(mod ≠ 11)	38	
JMP – Unconditional jump:						
Short/long	1 1 1 0 1 0 1 1	disp-low			13	
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high		13	
Register memory indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m			11/17	
Direct intersegment	1 1 1 0 1 0 1 0	segment offset			13	
		segment selector				
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m		(mod ≠ 11)	26	
RET = Return from CALL:						
Within segment	1 1 0 0 0 0 1 1				16	
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high		18	
Intersegment	1 1 0 0 1 0 1 1				22	
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high		25	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

186 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT				186 Clock Cycles	Comments
CONTROL TRANSFER (Continued):						
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	disp			4/13	13 if JMP taken 4 if JMP not taken
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp			4/13	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp			4/13	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp			4/13	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp			4/13	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	disp			4/13	
JO = Jump on overflow	0 1 1 1 0 0 0 0	disp			4/13	
JS = Jump on sign	0 1 1 1 1 0 0 0	disp			4/13	
JNE/JNZ = Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp			4/13	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp			4/13	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp			4/13	
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp			4/13	
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp			4/13	
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	disp			4/13	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	disp			4/13	
JNS = Jump on not sign	0 1 1 1 1 0 0 1	disp			4/13	
LOOP = Loop CX times	1 1 1 0 0 0 1 0	disp			5/15	
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1	disp			6/16	
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0	disp			6/16 16	JMP taken/
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1	disp			5	JMP not taken
ENTER = Enter Procedure	1 1 0 0 1 0 0 0	data-low	data-high	L		
L = 0					15	
L = 1					25	
L > 1					22+16(n−1)	
LEAVE = Leave Procedure	1 1 0 0 1 0 0 1				8	
INT = Interrupt:						
Type specified	1 1 0 0 1 1 0 1	type			47	

Type 3	1 1 0 0 1 1 0 0				45	if INT. taken/
						if INT. not taken

INTO = Interrupt on overflow	1 1 0 0 1 1 1 0				48/4	
IRET = Interrupt return	1 1 0 0 1 1 1 1				28	
BOUND = Detect value out of range	0 1 1 0 0 0 1 0	mod reg r/m			33–35	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

186 INSTRUCTION SET SUMMARY (Continued)

FUNCTION	FORMAT	186 Clock Cycles	Comments
PROCESSOR CONTROL			
CLC Clear carry	1 1 1 1 1 0 0 0	2	
CMC = Complement carry	1 1 1 1 0 1 0 1	2	
STC – Set carry	1 1 1 1 1 0 0 1	2	
CLD – Clear direction	1 1 1 1 1 1 0 0	2	
STD = Set direction	1 1 1 1 1 1 0 1	2	
CLI - Clear interrupt	1 1 1 1 1 0 1 0	2	
STI = Set interrupt	1 1 1 1 1 0 1 1	2	
HLT = Halt	1 1 1 1 0 1 0 0	2	
WAIT - Wait	1 0 0 1 1 0 1 1	6	if $\overline{\text{test}}$ = 0
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0	2	
ESC Processor Extension Escape	1 0 0 1 1 T T T | mod LLL r/m (TTT LLL are opcode to processor extension)	6	

Shaded areas indicate instructions not available in iAPX 86, 88 microsystems.

FOOTNOTES

The effective Address (EA) of the memory operand is computed according to the mod and r/m fields:

if mod = 11 then r/m is treated as a REG field

if mod = 00 then DISP = 0*, disp-low and disp-high are absent

if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent

if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP

if r/m = 001 then EA = (BX) + (DI) + DISP

if r/m = 010 then EA = (BP) + (SI) + DISP

if r/m = 011 then EA = (BP) + (DI) + DISP

if r/m = 100 then EA = (SI) + DISP

if r/m = 101 then EA = (DI) + DISP

if r/m = 110 then EA = (BP) + DISP*

if r/m = 111 then EA = (BX) + DISP

DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

reg is assigned according to the following:

reg	Segment Register
00	ES
01	CS
10	SS
11	DS

REG is assigned according to the following table:

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

The physical addresses of all operands addressed by the BP register are computed using the SS segment register. The physical addresses of the destination operands of the string primitive operations (those addressed by the DI register) are computed using the ES segment, which may not be overridden

Index

WE'D LIKE YOUR OPINION

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future documents

To order publications, contact the Intel Literature Department (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR AVERAGE EXCELLENT

1 2 3 4 5 6 7 8 9 10

□ Readability
□ Technical depth
□ Technical accuracy
□ Usefulness of material for your needs
□ Comprehensibility of material
□ OVERALL QUALITY OF THIS MANUAL

If you gave a 4 or less (in any category), please explain here:

____

____

____

What suggestions would you have for improving this manual:

____

____

★ ★ ★ ATTENTION ★ ★ ★

Receive 50% off on the next Intel publication you buy. Send us your comments, and we'll send you a 50%-off certificate.

If you would like us to call you for more specifics about this book, provide the following information. Please print clearly.

Name ______________________________________

Phone Number (________)______________________

Address ____________________________________

____

____

Thanks for taking the time to fill out this form.

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States, use the preprinted address provided on this form to return your comments. No postage is required. If you are not in the United States, return your comments to the Intel sales office in your country. For your convenience, a list of international sales offices is provided on the back cover of this document.

Intel Corporation
DTO Technical Publications, HF2-38
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9987