MCS-86™ MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL

Manual Order Number 9800640-02

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 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

The information in this document is subject to change without notice.

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.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

i	iSBC	Multimodule
ICE	Library Manager	PROMPT
iCS	MCS	Promware
Insite	Megachassis	RMX
Intel	Micromap	UPI
Intelevision	Multibus	μScope
Intellec		

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.

HOW TO USE THIS MANUAL

This manual describes the MCS-86 Macro Assembly Language, and is intended to be used by readers who have some familiarity with at least one assembly language, not necessarily an Intel assembly language.

Figure 0-1 highlights MCS-86 Macro Assembly Language features (directives, instructions, operands, labels, macros, etc.) and the chapters describing them. Refer to the Table of Contents and the Index for more references.

Figure 1-1 in Chapter 1 shows the overall MCS-86 software development process, and lists relevant manuals and their order numbers. If you are developing programs for the 8086 and/or 8088, you will probably need most of these manuals.

This manual is primarily a reference manual, and as such is not intended to be read cover-to-cover. However, in order to begin to understand this assembly language, you should first familiarize yourself with the following topics:

- Language Overview (Chapter 1)
- Segmentation and Addressability (Chapter 2)
- Defining and Initializing Data (Chapter 3)
- Accessing Data (Chapter 4)

Several features of this language are unusual for an assembly language:

1. It is strongly typed, not unlike some high-level languages; data items (labels identifying code, and variables identifying data) must be used as they are declared. Table 3-1 in Chapter 3 provides a summary of these notions.
2. As an interface to the 8086 architecture, this language provides segmentation and addressability mechanisms (in particular, the SEGMENT/ENDS statement-pair, and the ASSUME directive) which enable you to design your own 64K-byte windows into the megabyte memory. Chapter 2 describes segmentation. You should be sure to read the description of the ASSUME directive and that of "Anonymous References" in Chapter 2.
3. If you want to define, allocate, and initialize data items such as structures, records, arrays, and combinations, refer to Chapter 3.

   If you want to access these data items, refer to Chapter 4.

   Chapter 5 gives an encyclopedia of instruction mnemonics. Appendix J provides a convenient summary, including examples and flags affected.
4. MPL, the Macro Processor Language, is described both in Chapter 7 and in Appendix L. You can write programs without it, or you can speed up program development with it.

Finally, a sample program in Appendix K illustrates many of the language's unique features.

Input/Output is not discussed in this manual, except for the instructions IN and OUT (Chapter 5). Readers interested in I/O configurations and programming techniques are referred to the *MCS-86 User's Guide*, Order No. 9800722 and the *8086 Family User's Manual*, Order No. 9800722 (which also describes the programmable dual-channel 8089 and its assembly language).

Refer to the Intel Corporation publication, *8089 Assembler User's Guide*, Order No. 9800938 for a complete description of the 8089 assembly language and ASM89 assembler operation.

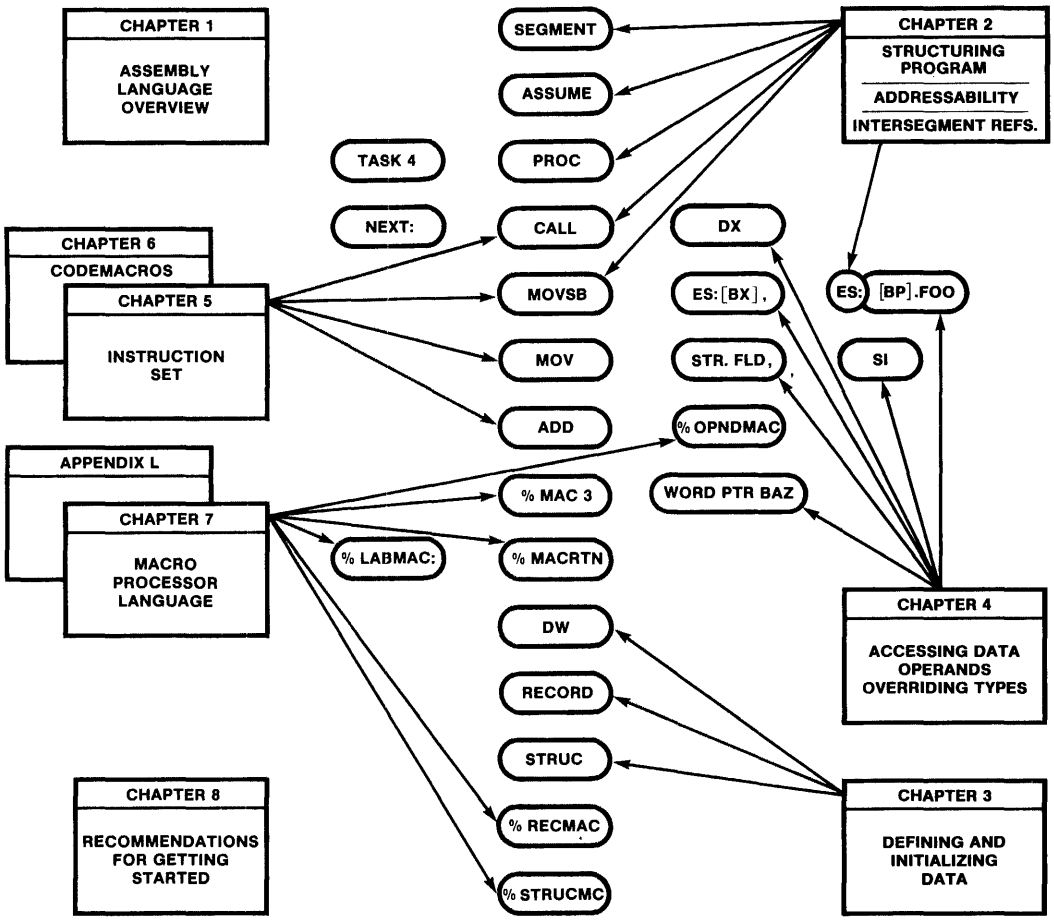


Figure 0-1. Structure of This Manual

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1
OVERVIEW OF THE MCS-86 MACRO ASSEMBLY LANGUAGE

This chapter presents an overview of the MCS-86 Macro Assembly Language, which is expressly designed for writing programs for the 16-bit 8086 processor. Figure 1-1 shows how the MCS-86 Macro Assembler fits into the 8086 software development environment.

The MCS-86 Macro Assembler, which creates object modules from these programs, forms part of a family of MCS-86 tools:

- CONV86, which converts error-free 8080/8085 source files to syntactically valid 8086 source files, and issues caution and error messages for conversions which may require editing.
- PLM86, which compiles programs written in PL/M-86, a high-level language, and creates object modules.
- LINK86, which combines object modules into load modules.
- LOC86, which binds load modules to absolute memory addresses.
- ICE-86, which provides in-circuit emulation for the 8086.

Manuals describing the languages and operation of MCS-86-related software are listed in the prefatory section to this manual, "How to Use This Manual."

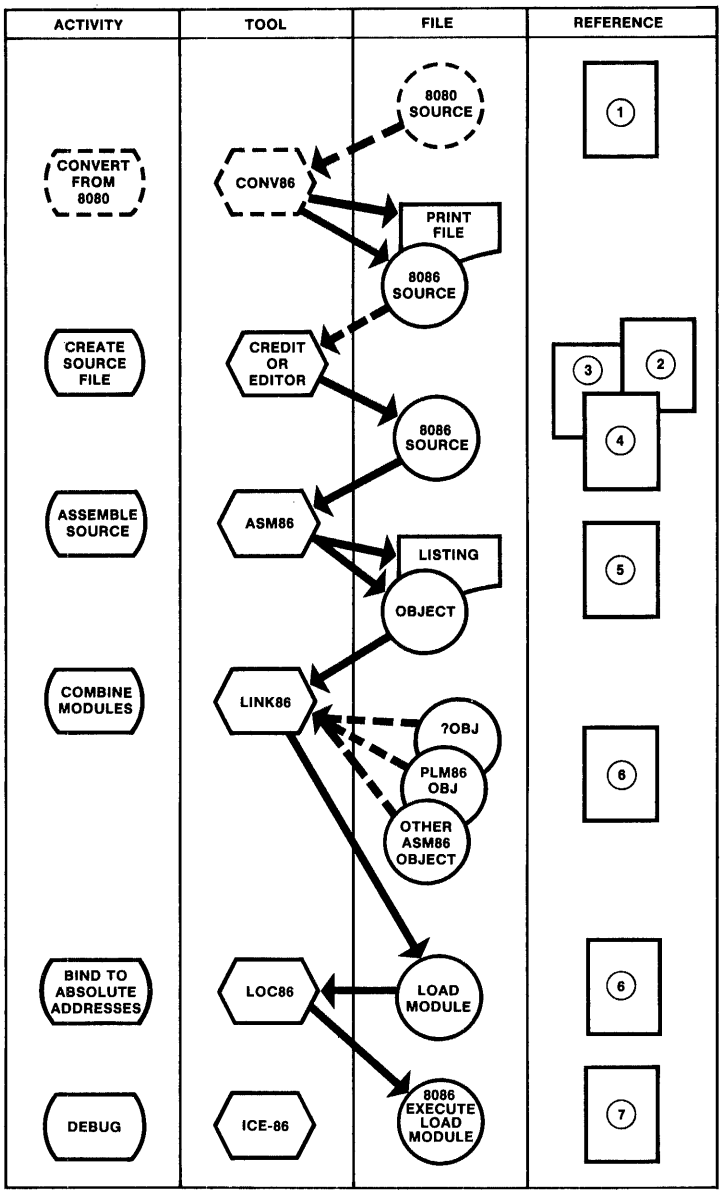
The MCS-86 Macro Assembly Language provides several powerful, yet relatively simple-to-use means of structuring and writing programs which can then be assembled, linked, located, and executed on the 16-bit 8086 microprocessor. If you are not familiar with the design of the 8086, you can read about it in the *8086 Family User's Manual,* Order No. 9800722.

Why Write Programs in Assembly Language?

Although high-level languages (PL/M, BASIC, FORTRAN, PASCAL) can decrease the development time needed for a program or system, they do not permit the programmer to directly access many of the chip's features, such as registers, processor flags, and special instructions. Moreover, high-level language compilers and interpreters tend to produce inefficient code for your algorithms. As a result, your assembly-language program, while sometimes taking longer to code and debug, usually runs much faster and occupies much less memory than the "equivalent" program written in a high-level language. Since program development time is itself expensive, however, the trade-off between development time and program performance should be analyzed for each application. The optimal solution is usually found in writing some routines in high-level language, and the more time-critical and space-critical routines in assembly language.

Is the Macro Processor Part of ASM86?

A single invocation of ASM86 gives control to the MCS-86 Macro Assembler, which contains as its front-end the Macro Processor. The Macro Processor scans your 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 file is assembled by the MCS-86 Macro Assembler.



(1) MCS-86 ASSEMBLY LANGUAGE CONVERTER OPERATING INSTRUCTIONS FOR ISIS-II USERS (9800642)

(2) CREDIT OPERATING INSTRUCTIONS FOR ISIS-II USERS (9800902)

(3) ISIS-II USER'S GUIDE (9800306)

(4) MCS-86 MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL (9800640)

(5) MCS-86 MACRO ASSEMBLER OPERATING INSTRUCTIONS FOR ISIS-II USERS (9800641)

(6) MCS-86 SOFTWARE DEVELOPMENT UTILITIES OPERATING INSTRUCTIONS FOR ISIS-II USERS (9800639)

(7) ICE-86 IN-CIRCUIT EMULATION OPERATING INSTRUCTIONS FOR ISIS-II USERS (9800741)

Figure 1-1. Software Development and the MCS-86 Macro Assembly Language

Using the Macro Processor Language (MPL) described in Chapter 7, you can create a library of powerful algorithms specific to your application from sequences of assembly-language instructions. Indeed, the combined use of MPL and MCS-86 Assembly Language offers both the run-time efficiency of assembly language and the decreased development time of high-level languages.

References in this manual to the MCS-86 Macro Assembly Language refer to to the assembly language proper; references to the MCS-86 Macro Assembler refer to the software that processes both MPL and the assembly language proper.

What Does the Assembly Language Offer?

Each of the language's unique advantages is further highlighted later in this chapter. They include:

- Powerful, yet human-engineered instruction set design.
- Sophisticated code and data structuring mechanisms usually found only in high-level languages.
- An assembler that checks:
  1. Consistency of data uses with data declarations. (This "strong typing" is used in most high-level languages.)
  2. Instruction forms to ensure shortest of valid forms is generated.
- A macro language rivalling any string manipulation language.

How Is the Instruction Set Designed?

Most assembly languages define a 1-1 correspondence between instruction mnemonics (e.g. MOVB, ADDI) and operation codes (opcodes, usually given in binary). The MCS-86 Macro Assembly Language establishes a one-to-many correspondence; a single instruction mnemonic can be assembled into one of several opcodes, depending on what types of operands (data items) are supplied with it. The assembler then chooses the appropriate opcode, based on the operand types.

The MCS-86 Macro Assembly language is thus a "strongly typed" language:

- It is typed, because data can be declared to be of different types (for instance, byte, word, doubleword).
- It is strongly typed, because mixed operand types are not permitted in the same operation (for instance, moving a declared byte to a word register, or vice versa).

Strong typing prevents you from inadvertently:

- Moving a word to a byte destination, and thus overwriting an adjacent byte.
- Moving a byte to a word destination, and thus leaving meaningless data in an adjacent byte.

But strong typing does not prevent you from deliberately performing these same operations. For instance, if DATA8 is declared type byte and if DATA16 is declared type word and if:

- You want to move the contents of 16-bit register AX to DATA8, thus overwriting the contents of DATA8 + 1, then you code:

```
MOV WORD PTR DATA8, AX ; Move AX to word at DATA8.
```

- You want to move the contents of 8-bit register AL to DATA16, thus leaving the high-order byte of DATA16 intact, then you code:

```
MOV BYTE PTR DATA16, AL ; Move AL to byte at DATA16.
```

These examples use the PTR (pointer) operator to override (temporarily redefine) the data types for the duration of the instruction's execution. Further references to DATA8 or DATA16 revert to the original data types, unless a type-overriding operator is again specified.

There are several ways to access typed data. If you need to access an array as both bytes and words, you can do it using PTR as above, or you can specify your data declaration as follows:

```
DATA8_ARRAY LABEL BYTE          ; Label used to access bytes in the following
DATA16_ARRAY DW 100 DUP (0)     ; 100 words of 0's.
```

Now to store the 8-bit register CH in the 10th byte (the first byte is byte 0) of the array, code:

```
MOV DATA8_ARRAY[9], CH
```

But to store the 16-bit register CX in the 10th and 11th bytes of the array, code:

```
MOV DATA16_ARRAY[9], CX
```

Type-overriding is fully described in Chapter 4.

These examples use MOV, but apply to any two-operand mnemonic (e.g. ADD, SUB, AND, XOR, etc.).

Generic Instruction Mnemonics and Codemacros

Thousands of distinct operations are represented by about 100 assembly-language mnemonics. This means you do not have to remember different mnemonics for "move word", "move byte", and "move immediate", for example; each is simply MOV. You code the mnemonic that applies to an entire class of operations, and from the operands (data items) you supply with it, the MCS-86 Macro Assembler chooses the best machine code.

This "generic" instruction facility is provided by "codemacros", which are described in Chapter 6. Codemacros should not be confused with macros (MPL, described in Chapter 7); codemacros define the formats of assembled instructions, whereas macros define transformations of source text into assembly language.

The assembler maps each assembly-language instruction into a machine-language instruction by referencing that mnemonic's set of codemacros, each of which specifies register, memory, displacement, opcode and type information. The assembler compares the operands you give with the mnemonic against the codemacros defined for that mnemonic, chooses the codemacro that matches your operand types, and expands that codemacro to a machine instruction. This is the fundamental assembly process.

Although you are free to redefine codemacros (and thus make up your own instruction set), it is not necessary for you to understand codemacros in order to write programs. If you want to know how an instruction is actually assembled, however, codemacros are the ultimate authority. Appendix A gives the entire set of codemacros for the MCS-86 Assembly Language. Chapter 5 shows assembled instruction formats.

Sample Instruction

Quite a bit of information can be expressed in a single, easy-to-code instruction. As an example of the power and simplicity of the language, consider the instruction:

```
ADD [BP][SI].STRUCFIELD3, DX
```

Without knowing anything about the language, you could deduce:

- That the contents of 16-bit register DX forms part of a sum.
- That the instruction performs a full-word addition operation.
- That the registers BP and SI are somehow used to calculate the address of the other part of the sum.
- That the identifier STRUCFIELD3, together with the special character period ".", are also used in the address calculation.

Figure 1-2 completes the picture. Here is the key:

1. Since it is the second operand, DX is the "source", and is being added to the contents of the word addressed by the expression [BP][SI].STRUCFIELD3, the "destination".
2. Register BP, also called the stack marker, participates in the instruction by forming the base address of the first operand. When BP is the base, the operand by default resides in the current stack segment. Since SS is the segment register for the stack segment, the 16-bit contents of SS yield the paragraph number (also called the "frame number") of the stack segment.
3. The 16-bit contents of register SI is used here to index the address of the "destination" operand. That is, BP and SI are added to form part of the effective address of the first operand.
4. The dot operator "." in this context refers to a structure. (Structure definition is fully described in Chapter 3.) STRUCFIELD3, the identifier that follows, identifies a structure field; its value gives the relative distance, in bytes, from the beginning of the structure to STRUCFIELD3. (The assembler generates relative offset values for each field of the structure relative to its beginning. The structure can thus be used as "storage template", or pattern of relative offset values.)
5. The address of the first operand, then, is:

```
16-bit Paragraph Number: SS
16-bit Offset: BP + SI + (relative offset of STRUCFIELD3)
-------------------------------------------------------------
20-bit Machine Address: 16*SS + BP + SI + (relative offset of STRUCFIELD3)
```

Thus, the effect of the instruction:

```
ADD [BP][SI].STRUCFIELD3, DX
```

is to add the contents of the 16-bit register DX to the word in the stack double-indexed by the 16-bit registers BP and SI, and offset by a structure-field displacement. This instruction, including opcode, base register, index register, structure displacement and relative offset, type information, direction, and source register, assembles into only three bytes.

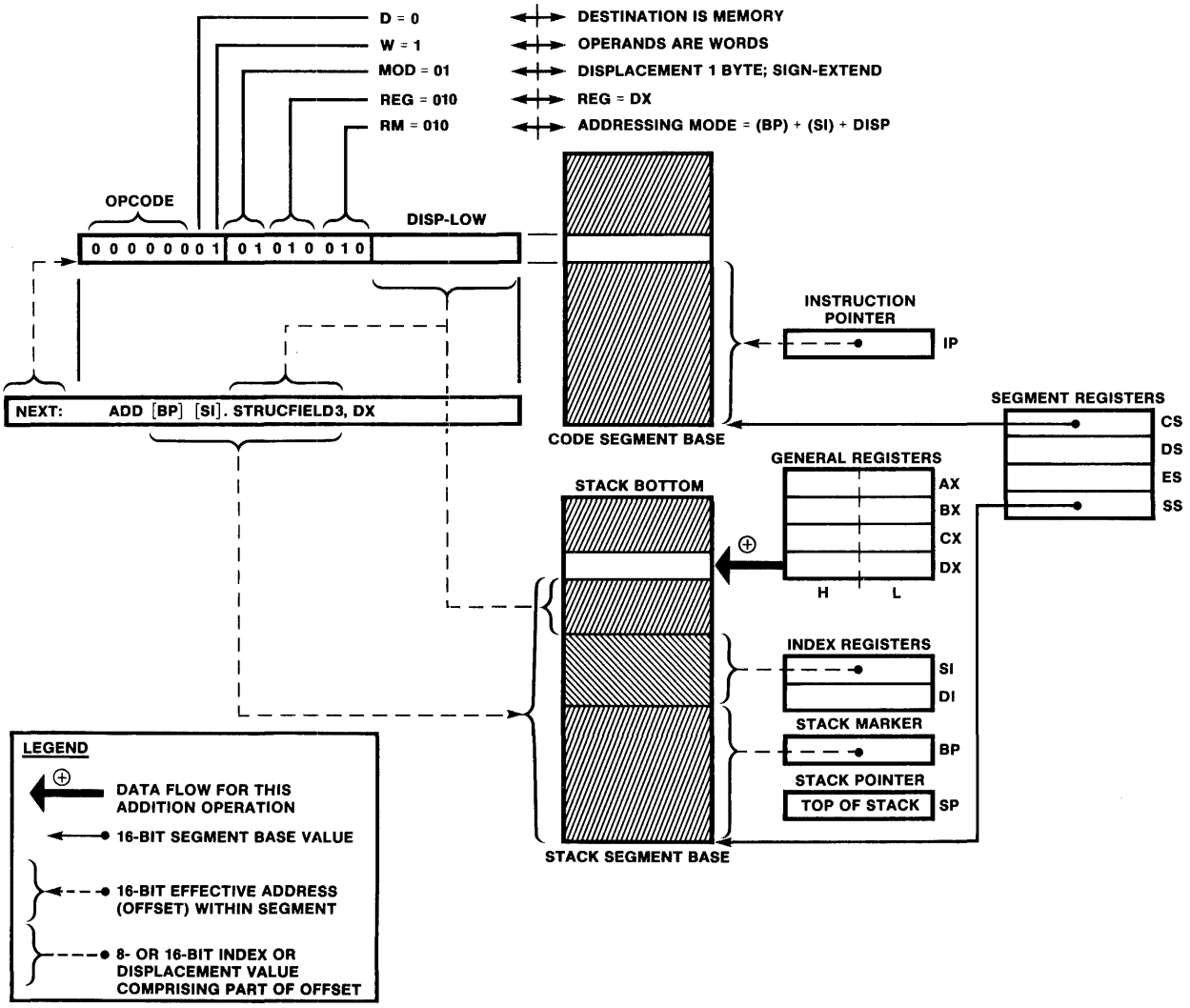


Figure 1-2. Analysis of Sample Instruction

How May Code and Data Be Structured?

Structures

The next example shows how structures can be used to define storage templates in order to access variables (whether in the stack or not). You can use structures to group together logically related data items, as in the following:

The user-named structure PHYS allows a programmer to define individual fields of size (in bytes) 1, 2, 2, 1, and 2 symbolically and have their relative offsets generated by the assembler:

```
PHYS    STRUC   ; No storage reserved — use this as template for JONES below.
AGE     DB 0    ; Reference to .AGE generates relative offset of 0.
EYES    DW 0    ; Reference to .EYES generates relative offset of 1.
HAIR    DW 0    ; Reference to .HAIR generates relative offset of 3.
HTFT    DB 0    ; Reference to .HTFT generates relative offset of 5.
HTIN    DW 0    ; Reference to .HTIN generates relative offset of 6.
PHYS    ENDS
```

Now the programmer can allocate real storage and initialize using PHYS as an operator:

```
JONES PHYS < 22, 'BL', 'BR', '5', '10' > ; Allocate 8 bytes, initialize
```

Notice that the user-assigned name PHYS is here used as an operator, and that the structure is initialized using both integer data (22) and character data ('BL', '5').

The preceding use of PHYS as an operator is the assembly-time counterpart of the following run-time initialization:

```
JONES DB 8 DUP(?)          ; Allocate 8 bytes (uninitialized).
MOV JONES.AGE, 22          ; Initialize .AGE field.
MOV JONES.EYES, 'BL'       ; Iinitialize .EYES field.
MOV JONES.HAIR, 'BR'       ; Initialize .HAIR field.
MOV JONES.HTFT, '5'        ; Initialize .HTFT field.
MOV JONES.HTIN, '10'       ; Initialize .HTIN field.
```

Chapter 3 describes still another flexible feature you can use in defining structures — that of initializing default value during the definition of the structure, and then overriding (when necessary) these default values during allocation of storage, all at assembly time.

Structures provide an extremely valuable programming tool not usually found in assembly languages. Chapter 3 describes structure definition, allocation and initialization. Chapter 4 describes how to access structures.

Arrays

You can define and initialize arrays of bytes, words, doublewords, structures, and records (defined below) using the DB, DW, DD, structure-name, and record-name directives, respectively:

```
BYTE_ARRAY    DB 100 DUP(1)        ; Allocate 100 bytes, initialize each to 1.
WORD_ARRAY    DW 256 DUP(0)        ; Allocate 256 words, initialize each to 0.
DWORD_ARRAY   DD 200 DUP(?)        ; Allocate 200 doublewords, don't care initialize.
PHYS_ARRAY    PHYS 1000 DUP(<>)    ; Allocate 1000 initialized copies of PHYS.
```

An example of an initialized record array follows under "RECORDS."

In referencing array elements, be sure to take into account:

- Arrays are zero-origined; thus, the first byte of an array FOO is FOO[0], not FOO[1]. The Nth byte is FOO[N-1].
- The index you code is interpreted as the number of BYTES from the start of the array, whether the array elements are bytes, words, or doublewords.

Elements within the arrays can then be accessed several ways:

```
ADD AH, BYTE_ARRAY              ; Add 1st byte of array to AH.
ADD AH, BYTE_ARRAY[0]           ; Add 1st byte of array to AH.
ADD AH, BYTE_ARRAY[7]           ; Add 8th byte of array to AH.
ADD AH, BYTE_ARRAY + 7          ; Add 8th byte of array to AH.
ADD AH, BYTE_ARRAY[SI]          ; Add (SI+1)st byte of array to AH.
MOV BX, WORD_ARRAY + 14         ; Move 8th word in array to BX.
MOV BX, WORD_ARRAY[14]          ; Move 8th word in array to BX.
MOV WORD_ARRAY[BX][SI], 7       ; Move 7 to word at (BX+SI+1)st byte of array.
MOV WORD_ARRAY[BX + SI], 7      ; Move 7 to word at (BX+SI+1)st byte of array.
```

Records

Records are analogous to structures, except that records deal with bit-offset values of bit-fields, whereas structures deal with byte-offset values of byte-fields. A record is a pattern defined to format a byte or word. You name each field within the record and assign it a length in bits; then a reference to the field-name is recognized by the assembler to be the shift count necessary to right-justify the field. You can isolate the unshifted field using the record operator MASK.

For example, suppose you are dealing with data formatted in 16-bit words as follows:

- Three 1-bit flip-flops (FF1, FF2, FF3)
- One 1-bit "Don't Care" field (DNC)
- Three 4-bit packed decimal digits (DIG1, DIG2, DIG3)

You could define the record as follows:

```
GONZO RECORD FF1:1, FF2:1, FF3:1, DNC:1, DIG1:4, DIG2:4, DIG3:4
```

Although the RECORD definition itself allocates no storage, you can use the record-name as an assembly-time operator to allocate copies of a defined record:

```
BUFFALO GONZO 100 DUP (<>) ;allocate 100 initialized copies
```

Now if you want to isolate the field corresponding to DIG2 from the second word of the array BUFFALO, and leave it in Bits 0-3 of register AX, you can code:

```
MOV AX, BUFFALO[2]      ; Move bit-packed word to AX.
AND AX, MASK DIG2       ; Leaves DIG2 field as is, sets all other bits to zero.
MOV CL, DIG2            ; Record name provides shift count. Load it into CL.
SHR AX, CL              ; Right-justifies DIG2 field to Bits 0-3 of AX.
```

To isolate another field of BUFFALO corresponding to the pattern of GONZO, you would use exactly the same code with the field-name in place of DIG2. (In general, the Nth word of the word-array BUFFALO is specified by BUFFALO[2*(N-1)]).

Chapter 3 describes yet another flexible feature of records — that of defining default values for fields in the RECORD definition directive, and then optionally overriding any or all of these values during allocation.

Records thus allow you to define and manipulate bit-packed bytes and words in a way that is simple to program and modify, and easy to read. Indeed, few high-level languages can match this feature.

Combinations

There is nothing to prevent you from defining arrays of records, or structures of arrays. The following allocates and initializes 1000 copies of PHYS as defined above under "STRUCTURES":

```
LOTSA_PHYS PHYS 1000 DUP(<>) ; Initialized 8000-byte array of 1000 PHYS copies.
```

More examples are given in Chapter 3.

Segmentation Concept

To aid in your memory management, the assembly language permits you to define segments as a means of grouping related information within memory blocks, each of which can be at most 64K-1 (65535) bytes in size.

A segment is the smallest relocatable unit of memory. Each block is contiguous (that is, there are no gaps allowed in a segment), but segments may be scattered throughout memory.

You can define as many segments as you like at assembly-time, provided you define at least one segment per assembly module. (Even if you omit segment definition statements, the assembler assigns the 5-character name ??SEG to a default segment.) Every instruction and every data item in your program must lie within some segment. There is nothing to prevent you from mixing code and data in some segments, although this practice is not always advisable. Some practical examples of segmentation are:

- A segment for global data
- A segment for local data
- A segment for the stack
- A segment for your main program
- A segment for shared (reentrant) subroutines
- A segment for serially reusable subroutines
- A segment for interrupt vectors
- A segment for interrupt routines

A frame, or physical segment, in the 8086 memory consists ofup to 65535 (64K - 1) bytes starting at an absolute address divisible by 16. Such an address is called a paragraph boundary. The paragraph numbers for the 8086 memory are thus 0, 16, 32, ..., 16*65535=1048560.

Since a logical segment (one you define) does not necessarily begin on a paragraph boundary, logical segments do not necessarily correspond to physical segments.

Since each segment begins in some paragraph, the four 16-bit segment registers (CS, DS, ES, and SS) are used to hold paragraph numbers where segments begin. There are therefore four "current" segments at any one time, and the paragraph (frame) number of each is called the segment base value, and is contained in its segment register as follows:

- CS register — always defines the current code segment
- DS register — usually defines the current data segment
- SS register — always defines the current stack segment
- ES register — can define an auxiliary data segment

At run-time, every 8086 memory reference requires two components in order to be physically addressed by the hardware:

1. A 16-bit segment base value which must be contained in one of the four segment registers CS, DS, ES, or SS, and
2. A 16-bit effective address giving the offset of the memory reference from the segment base value.

Effective addresses are usually calculated at assembly-time; segment base addresses can be specified at assembly-time, locate-time, or run-time. It is the responsibility of the programmer to maintain reliable segment base addresses in the segment registers. This procedure is described in Chapter 2.

When a data item is fetched from memory, 8086 CPU hardware combines the 16-bit effective address and 16-bit segment base address of the data item as follows:

```
20-bit address = 16*(segment base address) + effective address
```

For instance, if GONZO is assembled at offset 1200H in your data segment, and that segment is paragraph-aligned (the default), and if you load segment register DS with the value 5D00H, then the absolute address of GONZO is:

```
16*5D00H + 1200H = 5E200H
```

In practice, the assembly-language programmer need not be concerned with absolute 20-bit addresses. As a general rule it is best to deal with symbolic segment base addresses and symbolic effective addresses. In particular, performing arithmetic on segment base addresses is not recommended.

Procedures

The assembly language implements the subroutine concept using procedure definition. Whereas most assembly languages offer a starting-label and a return instruction as subroutine-building tools, the MCS-86 Macro Assembly Language takes into account the notion of a procedure as a block of code (and possibly data), and provides PROC and ENDP statements to demarcate the subroutine block. Thus, there can be no confusion as to the extent of the procedure:

```
READFILE PROC
  o
  o
  o
  RET
  o
  o
  o
  RET
READFILE ENDP
```

Procedures can be nested (one completely within another) but cannot overlap.

```
READFILE PROC
  o
  o
  o
  RET
  GETLINE PROC
    o
    o
    o
    RET
    GETCHAR PROC
      o
      o
      o
      RET
    GETCHAR ENDP
    o
    o
    o
  GETLINE ENDP
  o
  o
  o
READFILE ENDP
```

Operand Possibilities

The 8086 instruction set (described in Chapter 5) provides several different ways to address operands. Most two-operand instructions allow either memory or a register to serve as the first, or "destination", and either a memory, register or a constant within the instruction to serve as the second, or "source" operand. Memory-to-memory operations are excluded.

Operands in memory can 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.

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. (For instance, there is no Push Immediate instruction.) Virtually all 8086 operations may specify either 8- or 16-bit operands.

Registers

Registers are classed as follows:

- Segment 16-bit (CS, DS, SS, ES)
- General 16-bit (AX, BX, CX, DX, SP, BP, SI, DI)
- General 8-bit (AH, AL, BH, BL, CH, CL, DH, DL)
- Base and Index 16-bit (BX, BP, SI, DI)
- Flag 1-bit (AF, CF, DF, IF, OF, PF, SF, TF, ZF)

As described above, segment registers contain segment paragraph numbers. These registers are programmer-initialized, and beyond that are of no concern to the programmer except as described in Chapter 2 under "Segment Prefix".

Each of the general 8-bit, general 16-bit, and pointer and index 16-bit registers can participate in arithmetic and logical operations. Thus, although AX is frequently referred to as "the accumulator", the 8086 has eight distinct 16-bit accumulators (AX, BX, CX, DX, SP, BP, SI, DI) and eight distinct 8-bit accumulators (AH, AL, BH, BH, CH, CL, DH, DL), although each 8-bit accumulator is either the high-order byte (H) or the low-order byte (L) of AX, BX, CX, or DX.

The flags are updated after each instruction to reflect conditions detected in the processor or any accumulator. Appendix C describes flag operation. The instruction encyclopedia of Chapter 5 lists the flags affected for each instruction. Appendix J provides a summary of the instructions, including how flags are affected.

The flag-register mnemonics stand for:

AF — Auxiliary-carry
CF — Carry
DF — Direction
IF — Interrrupt-enable
OF — Overflow
PF — Parity
SF — Sign
TF — Trap
ZF — Zero

Addressing Modes

Operands (data items) can be addressed several different ways using various combinations of the following:

- Base registers — BX and BP
- Index registers — SI and DI
- Displacement — 8- or 16-bit value added to a base and/or index register
- Direct offset — 16-bit address without a base or index register

Using two-operand instructions (e.g. MOV, ADD, SUB, AND, OR, etc.), the source (rightmost) operand can be an immediate value (a constant contained in the instruction itself, such as MOV AX, 5), a register, or a memory reference. When the source is an immediate value, then the destination (leftmost) operand can be either a register or a memory reference.

If the source operand is not an immediate value, then one of the two operands must be a register. The other can be either a register or a memory reference.

When no register is specified in addressing a data item, the reference is termed direct. Examples:

```
ADD SUM, DX              ; SUM is addressed by 16-bit direct offset.
MOV BL, JONES.HTFT       ; Offset of JONES plus HTFT is 16-bit direct offset.
```

When a register is specified in addressing a data item, the reference is termed indirect. Examples:

```
ADD SUM[BX], DX          ; Dest. is base reg. plus 16-bit displacement.
MOV AX, [BP][SI]         ; Source is sum of base reg. and index reg.
OR AL, [BX][SI] + 20     ; Source is sum of base reg., index reg., 8-bit disp.
MOV [BP][SI+2], CH       ; Dest. is sum of base reg., index reg., 8-bit disp.
MOV [BX-1][SI+2], DX     ; Dest. is sum of base reg., index reg., 8-bit disp.
XOR BITS[BP][DI], AH     ; Dest. is sum of base reg., index reg*, 16-bit disp.
MOV AL, BITS[BX][DI+1]   ; Source is sum of base reg., index reg., 16-bit disp.
```

The rules for specifying valid operand expressions are detailed in Chapter 4, "Accessing Data".

Macros

The Macro Processing Language (MPL) provides a way for you to define shorthand function names for arbitrary text strings: constants, expressions, operands, directives, one or more instructions, comments, and so on. Moreover, you can define your functions using parameters, so that if you define a function to perform text-string replacement of an instruction, say:

```
PLOP: MOV FOO[BX][SI].FIELDC, WHATSIS XOR MASK DUKE ; Second operand immediate.
```

You can define any or all of the fields of the instruction to be parameters of the function. And, since you can nest function calls, you can build a side-file of your most frequently used operators, operands, expressions, mnemonics, labels, comments, and instruction sequences to use as you see fit.

You can use MPL to create data definition directives as well, and thus build a library of macro calls for defining bytes, words, doublewords, strings, records, and structures. A macro-time console I/O facility gives you the opportunity to assemble interactively.

An example in Chapter 7 shows how you can define an MPL function to interactively define, allocate storage for, and initialize a record array.

MPL's built-in functions include many of the string-manipulation capabilities heretofore available only in high-level languages, and a few that you might not be familiar with, such as MATCH, which you can use to parse strings.

MPL can be as simple or as complex as you want it to be; it's simply a question of what functions you want to define. And, in combination with the program- and data-structuring facilities of the MCS-86 Assembly Language, program development time can rival that of the high- level languages, without the concomitant expensive overhead of burgeoning memory requirements and inefficient code.

CHAPTER 2
STRUCTURING PROGRAMS

This chapter describes how to structure your MCS-86 Assembly Language programs according to the following topics (and their associated directives):

- Segmentation control and addressability (the SEGMENT/ENDS, ASSUME, and GROUP directives), including loading of segment registers, coding segment prefixes, and string instruction considerations (e.g. MOVS, MOVSB, MOVSW)
- Label definition (the LABEL directive)
- Procedure definition (the PROC, ENDP directives)
- Program linkage (the NAME, END, PUBLIC, and EXTRN directives)
- Location counter control (the ORG directive and the '$' symbol)

The EQU directive is defined and described in Chapter 4.

Chapter 1 introduces the concepts of segments and procedures. It is recommended that you read those sections first. Appendix L, when folded out, shows the sample program listing SAMPLE.LST containing examples of the SEGMENT/ENDS, ASSUME, PROC/ENDP, NAME, END, and EXTRN directives. This chapter contains examples as well.

Relationship of Segmentation to Assembly Modules

Your assembly module can result in:

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

or a combination of these, depending on your use of SEGMENT/ENDS directives. After assembly, you can combine segment fragments having the same name, and entire segments having appropriate combinability characteristics, using the LINK86 program, as described in *MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users*, Order No. 9800639.

Segmentation Control and Addressability

Formats of the SEGMENT/ENDS Directives

At run-time, each instruction and each variable of your program lies within some segment. If you do not name a segment, the assembler creates one, and names it ??SEG. To name your own segments, as well as to control their alignments, combinability, and contiguity (adjacency), you need the SEGMENT and ENDS directives, whose formats are as follows:

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

where the optional fields, if present, must be in the order shown, and:

align-type

specifies on what sort of boundary the segment must be located. The choices are:

1. PARA (the default) — specifies that the segment begins on a paragraph boundary, i.e., an address divisible by 16 (that is, least significant hexadecimal digit equal to 0H).
2. BYTE — specifies that the segment can begin anywhere.
3. WORD — specifies that the segment begins on a word boundary, i.e. an even address (least significant bit equal to 0B).
4. PAGE — specifies that the segment begins on a page boundary (an address whose two least significant hexadecimal digits are equal to 00H). This control is included for 8080 compatibility.
5. INPAGE — specifies that the entire segment occupies less than 256 bytes and that, when located, it must not overlap a page boundary. (Page boundaries are 00H, 100H, 200H, ... , 0FFF00H.) This control is included for 8080 compatibility.

combine-type

specifies how this segment may be combined with other segments for linking and locating. For details on combinability using LINK86, refer to *MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users*, Order No. 9800639.

The choices for combine-type are:

1. Not Combinable (Default) — If no combine-type is specified, the assembler assumes that this segment is not intended to be linked (using LINK86) with other segments. (If such an attempt is made, LINK86 will issue an error message.)
2. **PUBLIC** — specifies that this segment will be concatenated (made adjacent) to others of the same name when linked. You control the concatenating order during linkage using the LINK86 program.
3. **COMMON** — specifies that this segment and all other segments of the same name that are linked together (using LINK86) will begin at the same address, and thus overlap (as in a FORTRAN COMMON). The length of a linked COMMON is the maximum of the linked segments.
4. **AT expression** — specifies that this segment is to be located at the 16-bit paragraph number evaluated from the given expression. For example, if you specify AT 4444H, the segment begins at paragraph 4444, or absolute memory address 44440H. The expression can be any valid expression resulting in a constant (see "Expressions", Chapter 4), but no forward references are allowed.
5. **STACK** — specifies that this segment is to be part of the run-time stack segment, accessed last-in first-out (LIFO) using the assembler instructions PUSH, POP, CALL, INT, IRET, and RET. Stack segments are overlaid against high memory (all begin at the same address) and grow "downward". The storage allocated to the stack segment is the sum of storage allocations for each individual segment, since each might fill its own portion.
6. **MEMORY** — specifies that this segment is to be located "above" (at a higher address than) all other segments being linked together. If several segments having the combine-type MEMORY are linked together, only the first one encountered is treated as a MEMORY segment; all others are treated as COMMON segments.

'classname'

specifies a classname for the segment, and must be enclosed in single quotes. Specifying a classname gives you another means of collecting similarly specified segments at locate-time. (The first means is by segment name.)

Segments generated by the PL/M-86 compiler are given the following predefined classnames:

- CODE
- CONST
- DATA
- STACK
- MEMORY

If you are not linking or locating with PL/M-86 modules, you are free to make up your own classnames. By specifying a classname, you can manipulate the class of segments (possibly including PL/M-86-generated segments) at locate-time. (Refer to the manual, *MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users*, Order No. 9800639, for descriptions of how LINK86 and LOC86 treat classnames.)

"Nested" or "Embedded" Segments

Segments are never physically nested or embedded, although it is permissible for you to code a portion of a segment, start another and end it, and then resume coding the first. When this is done, the assembler concatenates (appends) the second portion of the segment to the first. The segments are said to be *lexically* nested (but not physically nested).

For example, the following sequence of code is permitted, and results in two separate segments (DATA1 will not be embedded in CODE1):

```
CODE1 SEGMENT
      ASSUME CS:CODE1, DS:DATA1
        o
        o
        o
  DATA1 SEGMENT
          o
          o
          o
  DATA1 ENDS
        o
        o
        o
CODE1 ENDS
```

Overlapping segments are not permitted; that is, each lexically nested segment (i.e., one lies "inside" another in the program listing) must be ended with an ENDS directive before the enclosing SEGMENT directive is closed by an ENDS directive.

For example, the following is NOT permitted:

```
CODE3 SEGMENT          ; Start segment.
        o
        o
        o
CODE4 SEGMENT          ; Start segment.
        o
        o
        o
CODE3 ENDS             ; End first inside second — *** ERROR ***
        o
        o
        o
CODE4 ENDS
```

The ASSUME Directive

At run-time, every 8086 memory reference requires two components in order to be physically addressed by the hardware:

1. A 16-bit segment base value which must be contained in one of the four segment registers CS, DS, ES, or SS, and
2. A 16-bit effective address giving the offset of the memory reference from the segment base value.

The ASSUME directive builds a symbolic link between:

- Your assembly-time definition (placement) of instructions and data in logical segments (between SEGMENT/ENDS pairs), and
- The run-time event of physically addressing instructions and data in memory through segment registers.

In other words, ASSUME is a "promise" to the assembler that instructions and data are run-time addressable through certain segment registers. The actual loading and manipulation of values in segment registers is the responsibility of the programmer; ASSUME enables the assembler to check that every data item and every instruction is addressable through the segment registers.

The format of the ASSUME directive is:

ASSUME seg-reg:seg-name [, ...]

or:

ASSUME NOTHING

where:

- **Seg-reg** is one of CS, DS, ES, or SS
- **Seg-name** is:
  1. A segment name, as in:
     ASSUME CS:CODE4, DS:DATA4
  2. A previously-defined GROUP name, as in:
     ASSUME DS:DGROUP2, CS:DGROUP2
  3. The expression SEG variable-name or SEG label-name, as in:
     ASSUME CS:SEG BEGIN, DS:SEG FOO
  4. The keyword NOTHING, as in:
     ASSUME ES:NOTHING

Note that ASSUME NOTHING is equivalent to:

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

A given "seg-reg : seg-name" pair stays in effect until a subsequent ASSUME assigns a different segment (or NOTHING) to the given "seg-reg". The keyword NOTHING cancels any previous ASSUMEs for the indicated registers. If a variable's segment-name is not specified in an ASSUME directive currently in effect, each reference to that variable must specify a segment prefix, or it will be flagged as an error. (See "Segment Prefix" in this chapter.)

Most of the time, you will need an ASSUME directive of the form:

```
ASSUME CS:code-segment, DS:data-segment
```

as, for example, in the following:

```
ARRAYS SEGMENT
      FOO         DW 100 DUP (0)  ; Array of 100 words, initially 0's.
      BAZ         DW 500 DUP (0)  ; Array of 500 words, initially 0's.
      AXOLOTL     DW 800 DUP (0)  ; Array of 800 words, initially 0's.
ARRAYS ENDS
SUM SEGMENT
     ASSUME CS:SUM, DS:ARRAYS ; SUM is addressable through CS.
     START: MOV AX, FOO          ; FOO addressable — defined in ARRAYS.
            ADD AX, BAZ          ; BAZ addressable — defined in ARRAYS.
            MOV AXOLOTL, AX      ; AXOLOTL addressable — defined in ARRAYS.
SUM ENDS
```

The ASSUME directive in the above example tells the assembler that:

1. The instructions in the segment "SUM" are addressable through CS. Note that since CS is not loaded by this program fragment, we are assuming that CS is set to point to the segment SUM before control is passed to it (through the label START). In general, CS is initialized by means of a long jump, long call, interrupt, or hardware RESET. Each of these loads a new segment base address into CS.

2. The symbolic references FOO, BAZ, and AXOLOTL are addressable through DS (they are defined in the segment ARRAYS).

As a second example, if FOO2 were defined in SUM, BAZ2 in ARRAYS, and AXOLOTL2 in a third segment POGUE, addressed by ES, you would code:

```
SUM SEGMENT
    ASSUME CS:SUM, DS:ARRAYS, ES:POGUE
        FOO2    DW   5
        START:  MOV  AX, FOO2       ; FOO2 addressable—defined in SUM.
                ADD  AX, BAZ2       ; BAZ2 addressable—defined in ARRAYS.
                MOV  AXOLOTL2, AX ; AXOLOTL2 addressable—defined in POGUE.
SUM ENDS
```

Your ASSUME thus "covers the bases", and you have fulfilled the "promise" that ASSUME makes to the assembler:

> Every instruction and every named data item is addressable through the segment registers specified in the ASSUME directive, unless overridden by a segment prefix. Actual loading of the segment registers is the responsibility of the programmer.

Loading Segment Registers

As stated earlier, the CS register can be loaded by:

- A long jump (JMP)
- A long call (CALL)
- An interrupt (INT n, or external interrupt)
- A hardware RESET

The addressability of long jumps/calls is discussed in this chapter under "Label Addressability".

The instruction INT N causes the instruction pointer (IP) to be loaded with the 16-bit value stored at absolute memory location 4*N, and causes the CS register to be loaded with the 16-bit value stored at absolute memory location 4*N + 2.

A hardware RESET sets CS to 0FFFFH and IP to 0.

You load the stack segment register, SS, as follows:

```
STACK1 SEGMENT
        DW 1000 DUP (0)         ; 1000-word stack of zeroes initially.
STACK_BOTTOM LABEL WORD ; Stack grows toward low memory
STACK1 ENDS
STACK_INIT  SEGMENT
            ASSUME   CS:STACK_INIT
            MOV      AX, STACK1
            MOV      SS, AX ; Never move immediate value to seg-reg.
            MOV      SP, OFFSET STACK_BOTTOM ; bottom = top initially.
STACK_INIT  ENDS
```

The next example shows how to load DS. ES can be loaded similarly.

DATA and DATA2 are segment names, which are treated as numbers and assembled as immediate values. Thus, segment names do not require segment registers in order to be addressable. Note that DS is loaded using the first two instructions in segment CODE, and ES by the next two. The segment DATA appears in an ASSUME directive (and is declared to be addressed by DS), but segment DATA2 is not covered by an ASSUME.

But FOO (in DATA) and BAZ (in DATA2) both need to be covered by ASSUME. So, although the segment registers are handled properly by the code, the assembler still reports an error for the line MOV BAZ, 99 because BAZ is not covered by either ASSUME.

```
DATA    SEGMENT
FOO     DB      0
DATA    ENDS

DATA2   SEGMENT
BAZ     DB      1
DATA2   ENDS

CODE    SEGMENT
ASSUME  CS:CODE
MOV     AX, DATA    ; Move base of segment DATA
MOV     DS, AX      ; Into segment register DS.
MOV     AX, DATA2   ; Move base of segment DATA2
MOV     ES, AX      ; Into segment register ES.
```

```
ASSUME   DS:DATA     ; Inform assembler of base value in DS.
MOV      FOO, 99     ; Addressable, since FOO is in DATA,
                     ; and base of FOO is in DS.
MOV      BAZ, 99     ; GIVES AN ERROR — BAZ is not
                     ; addressable, since we haven't told
                     ; the assembler that ES contains
                     ; the base value of DATA2.
```

When your program references a variable without a segment prefix, the assembler determines the segment containing that variable, and then examines your ASSUMEs to determine which segment register addresses that segment. If no ASSUME specifies the required segment, the assembler issues an error message. If ASSUME specifies the required segment, or if a segment prefix is specified with the variable reference (as described in the next paragraph), the assembler generates the correct code to address the given variable.

Segment Prefix

If a reference to a named variable is not covered by an ASSUME directive, you can inform the assembler of the variable's segment register by explicitly coding a segment prefix of the form:

```
seg-reg:
```

where "seg-reg" is one of CS, DS, ES, or SS in front of the variable reference, as in:

```
DS:BAZ
```

Although this construct has the advantage of not requiring an ASSUME directive for the variable reference, it has two disadvantages:

1. Its scope is one instruction; that is, in lieu of an ASSUME, the segment prefix must be coded for every reference to a variable.
2. It is more error-prone than ASSUME, since it refers to a segment register and not to a segment name, and also because it is easy to leave out.

Thus, the following are equivalent, and assemble correctly:

```
SUM SEGMENT                    SUM SEGMENT
ASSUME CS:SUM,DS:ARRAYS        ASSUME CS:SUM
   MOV AX, FOO                     MOV AX, DS:FOO
   ADD AX, BAZ                     ADD AX, DS:BAZ
   MOV AXOLOTL, AX                 MOV DS:AXOLOTL, AX
SUM ENDS                       SUM ENDS
```

where ARRAYS, for both assemblies, is defined to be:

```
ARRAYS SEGMENT
      FOO   DW          100 DUP (0) ; 100 words 0's
      BAZ   DW          500 DUP (0) ; 500 words 0's
      AXOLOTL DW        800 DUP (0) ; 800 words 0's
ARRAYS ENDS
```

The segment prefix is like a temporary ASSUME for the single instruction in which it is used. The segment register assignment specified in an ASSUME directive, however, stays in effect until a subsequent ASSUME directive reassigns the segment register (or deassigns it using ASSUME NOTHING).

Anonymous References

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 (also called "split") 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
          o
          o
          o
FOO     DW    500 DUP (0) ; 500 words of 0's
          o
          o
          o
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 SEG1, put
                        ; first word of FOO into AX.
```

Figure 2-1 shows single- and double-indexed anonymous references for the four current segments.

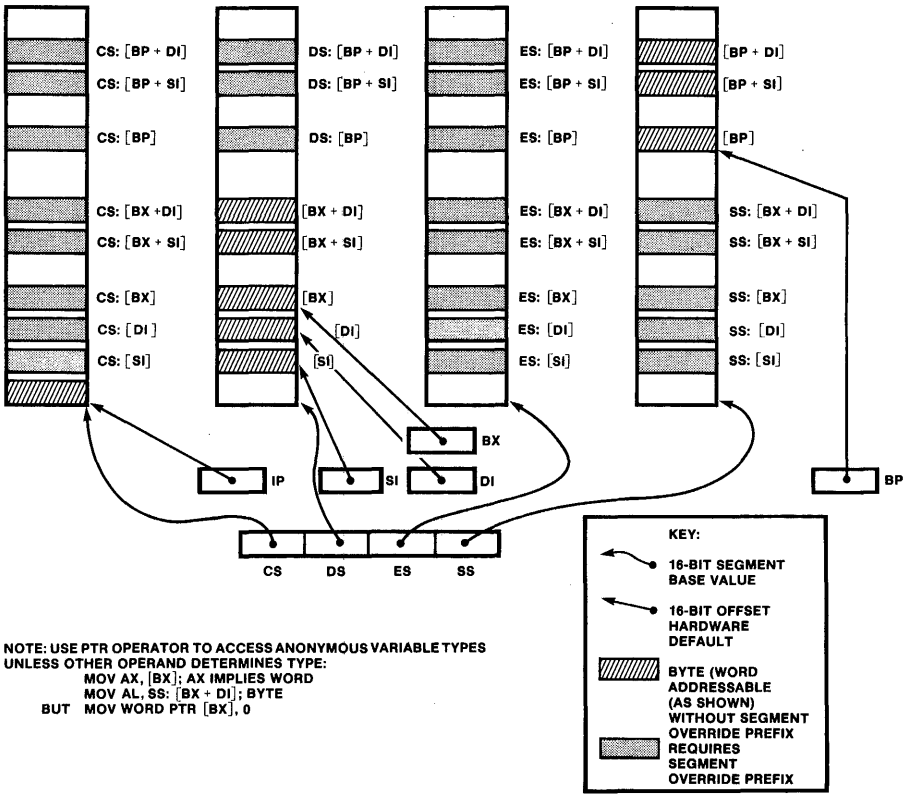


Figure 2-1. Anonymous Variable References and Segment Prefix Overrides

Examples Using Anonymous (Split) Variables

```
ADD AX, [BP]            is the same as      ADD AX, SS:[BP]
MOV [BX+2], AX          is the same as      MOV DS:[BX+2], AX
XOR [BX+SI], CX         is the same as      XOR DS:[BX+SI], CX
AND [BP+SI], CX         is the same as      AND SS:[BP+SI], CX
MOV BX, [DI].FLD        is the same as      MOV BX, DS:[DI].FLD
AND [SI], CX            is the same as      AND DS:[SI], CX
SUB [SP], DX            is the same as      SUB SS:[SP], DX

ADD AX, DS:[BP]           overrides         ADD AX, SS:[BP]
MOV CS:[BX+2], AX         overrides         MOV DS:[BX+2], AX
XOR SS:[BX+SI], CX        overrides         XOR DS:[BX+SI], CX
AND DS:[BP+SI], CX        overrides         AND SS:[BP+SI], CX
MOV BX, CS:[DI].FLD       overrides         MOV BX, DS:[DI].FLD
AND ES:[SI], CX           overrides         AND DS:[SI], CX
```

String Instructions and Memory References

Table 2-1 shows the mnemonics of the string instructions, which can be coded without operands (MOVSB, MOVSW, etc.) or with operands (MOVS, etc.).

Table 2-1. 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

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

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 in Chapter 5.)
3. Each can be coded with or without symbolic memory operands.
   - If symbolic operands are coded, the assembler can check the addressability of them for you.
   - Anonymous references which 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 which do not use the hardware defaults require both segment and type overriding:

     ```
     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 registers are as follows:
   - SI defaults to an offset in the segment addressed by DS
   - DI is required to be an offset in the segment addressed by ES

   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.

Groups (the GROUP directive)

You can use the GROUP directive to specify that certain segments lie within the same 64K bytes of memory. The format of this directive is:

name GROUP segnam [, ...]

where:

name is a unique identifier

segnam can be:

- The name field of a SEGMENT directive.
- An expression of the form **SEG variable-name** (refer to the SEG operator defined in Chapter 4).
- An expression of the form **SEG label-name** (refer to the SEG operator defined in Chapter 4).

[, ...] denotes an optional list of **segnams** separated by commas.

The GROUP directive defines a group with the given **name** as a collection of the given **segnam's**. You can use the group-name is almost all the ways you can use a segment name, except that a group-name cannot appear as a **segnam** in another GROUP statement.

Three uses are noteworthy:

1. As an immediate value, loaded first into a general register, and then into a segment register (never load an immediate value directly into a segment register):

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

   This loads the group base into DS. The group base is computed by LOC86 as a base value which covers all the named segments.
2. In an ASSUME statement, to indicate that all the segments named in the group are covered by the segment register:

   ```
   ASSUME DS:DGROUP
   ```
3. As an operand prefix, to specify that the group base value or offset is to be used (rather than the default segment base value or offset), as in the following:

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

Refer to the OFFSET operator in Chapter 4 for further information on how it pertains to groups.

The assembler cannot check to see if all the segments named will fit in 64K, but it causes such a check to be made by LOC86. If they do not fit, LOC86 will tell you. This directive does not specify where segments are loaded; the classname parameter in the SEGMENT directive does that. As an example of the GROUP directive:

```
BUNCH GROUP DATA2, STACK3
```

The appropriate ASSUME directive could be:

```
ASSUME CS:CODE1, DS:BUNCH, SS:BUNCH
```

Forward references to GROUPs are not permitted.

If contiguity (adjacency) of segments is desired, the classname parameter of the SEGMENT directive can achieve this goal more easily than use of the GROUP directive.

The order of the segments in the GROUP directive is not necessarily the order of the segments in memory. It is possible to address all the segments in a group from a single segment register (CS). However, great care should be taken in doing this, because offsets within the group do not correspond to offsets within a segment, notably assembly-time offsets. You should debug using LOC86 offsets, not ASM86 offsets.

LABEL Directive

The LABEL directive creates a name for the current location of assembly, whether data or instruction. The format of this directive is:

```
name LABEL type
```

where **name** is assigned the following attributes:

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

and **type** can be:

- NEAR or FAR, if executable code follows (usually). The label can be used in jumps or calls, but not in MOVs or other data manipulation instructions. Subscripting labels (of type NEAR or FAR) is not allowed.
- BYTE, WORD, DWORD, structure-name, or record-name, if data follows (usually). You can subscript an identifier declared using the LABEL directive if the directive assigns it a variable type, e.g. BYTE, WORD, etc. (See the example below under "Using LABEL with Variables.") In this case the name is a variable and is valid in MOVs, ADDs, etc., but not directly in jumps or calls. (An indirect jump or call uses a variable of type WORD or DWORD, as described in Chapter 4.)

Usage, rather than the assembly language, dictates the two instances of "(usually)" above in LABEL type declaration. The language does not prevent you from coding a NEAR or FAR label to precede data definition, nor does it prevent you from coding BYTE, WORD, etc. labels preceding code. However, great care should be taken in using data as code or vice versa.

The label/variable attributes segment, offset, and type are fully described in Chapter 3, "Defining and Initializing Data."

A name defined to be a variable using the LABEL directive is assigned LENGTH=1 and SIZE=TYPE (BYTE, WORD, DWORD, or N for structures, where N is the number of bytes defined for a given structure).

The principal uses for LABEL are:

1. To access variables (particularly arrays) by BYTE or WORD as needed. (An example is given below.)
2. To define a FAR label.
3. To provide an existing NEAR label (one with a colon) with a label having the same segment and offset values, but with the FAR distance attribute, so that the code is accessible from other segments as well. (An example is given below.)

Using LABEL with Variables

The LABEL directive can be used to associate another name and type to a location so that the data can be referenced in another way without using the PTR operator (defined in Chapter 4). For example, if you need to treat the same area of memory as both a byte and word array, you could define:

```
ARRAYB    LABEL    BYTE
ARRAYW    DW       1000 DUP (0)
```

Then in referencing this array, you could code:

```
ADD AL, ARRAYB[99]          ; Add 100th byte in array to AL.
ADD AX, ARRAYW[98]          ; Add  50th word in array to AX.
```

Using LABEL with Code

A label definition for types NEAR and FAR is allowed only when the segment currently being assembled is addressed by the CS segment register. This means that you must provide an ASSUME directive of the form:

```
ASSUME CS:name
```

where "name" is either the name of the current segment or the name of a group containing the current segment. (If a group-name is used, the label's offset is taken from the base of the group.)

When you already have a NEAR label (one with a colon) in your code, and you want to make that piece of code accessible from other segments, you can insert a LABEL directive (with a different name) and declare it FAR (you will also need a PUBLIC directive if access is desired from code in another assembly module):

```
ADD_MORE_ENTRY_POINT  LABEL  FAR
            ADD_MORE:    ADD AX, FOO[BX]
                         o
                         o
                         o
```

Label Addressability

Labels, as the operands of jumps and calls, present a much simpler case (to you and to the assembler) than variables (data), since the only relevant segment register is CS. The addressability of a label depends on how it is declared and how it is used:

1. Declaration — Is the target label (the target being jumped to or called) declared as NEAR or FAR?
2. Use — Are the jump/call instruction and its target assembled under the same ASSUME CS: directive?

Table 2-2 summarizes what the assembler generates in each case:

Table 2-2. Addressability of Jump/Call Target Labels

	Target Label Declared NEAR	**Target Label Declared FAR**
Jump/Call Assembled Under Same ASSUME CS:	NEAR Jump/Call	NEAR Jump FAR Call
Jump/Call Assembled Under Different ASSUME CS:	***ERROR***	FAR Jump FAR Call

In using this table, recall that NEAR jumps and calls are assembled with a 2-byte displacement (with wraparound, so that all 64K-1 bytes of a segment are addressable), whereas FAR jumps and calls are assembled using a 4-byte displacement (16-bit offset and 16-bit paragraph number, so that the entire one million bytes of memory are addressable).

The assembler uses the ASSUME CS: information to ensure that instructions at the target of the jump or call are in fact addressable when control is transferred. Moreover, if the ASSUME CS: information is different at the target, and the target is declared FAR, the assembler automatically generates a FAR (4-byte displacement) jump or call to satisfy the target ASSUME CS: information.

Procedures (the PROC/ENDP Directives)

The assembly language provides procedures to implement the concept of subroutines. Procedures can be executed in-line (control "falls through" to them), jumped to, or invoked by a CALL. Calls are recommended as a better programming practice.

The format of the PROC/ENDP directives is:

```
name    PROC    [ NEAR | FAR ]
          o
          o
          o
        RET
          o
          o
          o
name    ENDP
```

where "name" is an identifier which must appear in both the PROC and ENDP directives. It is assigned type NEAR or FAR, whichever is specified. If neither is specified, the type defaults to NEAR. You should specify FAR if the procedure will be called from code which has another ASSUME CS value. The procedure type determines whether RET is assembled NEAR (2-byte offset) or FAR (2-byte offset and 2-byte segment base value).

Advantages of Using Procedures

Although any algorithm can be implemented without procedures, procedures are recommended because:

- They form the basis for modular programming.
- They are easier to read, update, and document.
- They can comprise your program libraries.
- They can reduce your program's total object code.

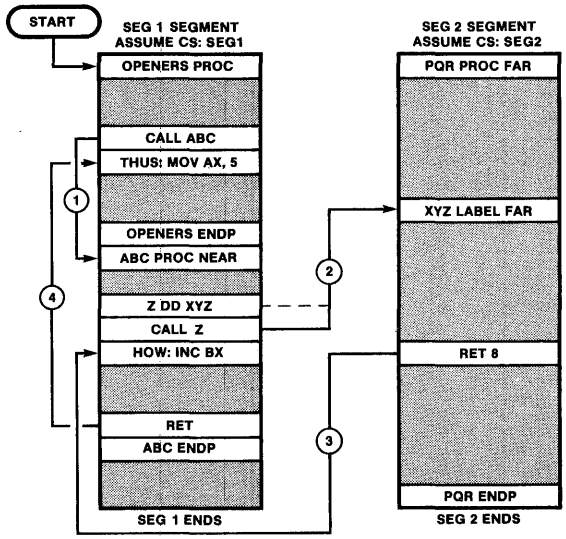
Calling a Procedure

When you call a NEAR procedure, the instruction pointer (IP, the address of the next sequential instruction) is automatically pushed on the stack, and control is transferred to the first instruction in the procedure.

When you call a FAR procedure, the CS register is automatically pushed on the stack first, then the IP, and control is transferred to the first instruction in the procedure.

Multiple entry points to a procedure are allowed; you should declare these as FAR labels if they are referenced from another segment. If referenced only from their own segment, the labels can default to NEAR. Entry points can be mixed NEAR and FAR, but doing so requires great care, since all returns assembled will be of the same type as the enclosing procedures.

Figure 2-2 shows procedure CALL/RET control flow.



KEY:

START	①	②	③	④
CAN BE FROM HARDWARE RESET EXTERNAL INTERRUPT INT N, CALL BX, NEAR/FAR JUMP/CALL, BUT IN ANY CASE CS ← SEG 1 IP ← OFFSET OPENERS	SP ← SP-2 [SP] ← IP IP ← OFFSET ABC	SP ← SP-2 [SP] ← CS CS ← SEG 2 SP ← SP-2 [SP] ← IP IP ← OFFSET XYZ	IP ← [SP] SP ← SP + 2 CS ← [SP] SP ← SP + 2 AND SP ← SP + 8 (FOR RET 8)	IP ← [SP] SP ← SP + 2

Figure 2-2. CALL/RET Control Flow

Recursive Procedures, Nested Procedures, and In-Line Procedures

Procedures can call other procedures; the same rules for declaration, calling, and returning apply.

A recursive procedure (a procedure which calls itself, or which calls another procedure which in turn calls the first, etc.):

- Must be coded reentrantly; that is, it must manipulate its local variables on the stack, using [BP] addressing modes
- Must discard its local variables from the stack before returning.

Recursion is not recommended when there is an obvious solution by iteration, i.e., internal looping.

The size of your stack segment determines the "nesting limit" — that is, how many levels of calls can be nested. Remember that FAR calls take up two words on the stack, and NEAR calls one word. Any values passed on the stack (and any local variables stored on the stack) should also be taken into account. (For example, if

your procedure is called by another procedure that has placed a value (an argument, sometimes called a parameter) on the stack, and your procedure issues a return without popping that value off the stack, that argument will still be on the stack after the return. Rather than having the calling procedure pop arguments after every call, you can issue a RET 2 and the top 2 bytes of the stack will be discarded upon returning. (See the description of RET in Chapter 5.)

If one procedure is declared within the PROC/ENDP directive-pair of another, that procedure will execute in-line (control "falls through" to it) unless control is directed around it using a jump or call. It is better programming practice to call a procedure (and return from it) than to execute it in-line or jump to/from it. Jumps and fall-throughs fail to take the stack into account, and this may result in returns to unexpected locations.

Returning from a Procedure

There is no automatic or implicit return from a procedure. Returns are normally specified using the RET instruction; returns from interrupt routines are specified using the IRET instruction.

More than one RET (or IRET) instruction can appear in a procedure; the RET (or IRET) instruction need not be the last in the procedure.

A return from a FAR procedure pops the word at the top of the stack into IP, then pops the next word into CS, thus accomplishing the return of control to the next sequential instruction following the point of call.

A return from a NEAR procedure pops the word at the top of the stack into IP, thus returning control.

If the procedure (or any procedure it calls) uses the stack for storage of temporary data items, these data items must be discarded (by popping or directly adjusting SP) before (or during) the return; otherwise, the IP/CS registers will receive them upon "return" -- which in this case may transfer control to an unexpected location.

Program Linkage Directives (NAME/END, PUBLIC, and EXTRN)

MCS-86 relocation and linkage (R & L) facilities enable you to combine several different assembly modules into a single load module for execution. (R & L facilities are LINK86 and LOC86, described in *MCS-86™ Software Development Utilities Operating Instructions for ISIS-II Users*, Order No. 9800641-03).

To identify intermodular symbolic references for program linkage, your assembly module can use these three program linkage directives:

- NAME — assigns a name to the object module generated by this assembly.
- PUBLIC — specifies symbols defined in this assembly module whose attributes are to be made available to other modules at link-time.
- EXTRN — specifies symbols defined in other assembly modules (and declared PUBLIC by them) whose attributes are needed by this assembly module at link-time.

NOTE

Unlike other assemblers, ASM86 interprets an EXTRN according to its context, i.e. placement. Refer to the section below, "Placement of EXTRNs"

The NAME Directive

The NAME directive assigns a name to the object module generated by this assembly. The format of the directive is:

NAME module-name

Example:

```
NAME MOD123
```

The NAME directive is not permitted a label. Thus,

```
MODNAME: NAME MOD123 ; Not acceptable
```

is INVALID.

If the NAME directive is miscoded or missing, the assembler supplies a default module name consisting of the source file name. Thus, if no valid NAME directive appears in a source file PSHBIT.S86, the resulting module name is PSHBIT.

The PUBLIC Directive

The PUBLIC directive specifies which symbols in this module are to be made available to other modules at link-time. Its format is:

PUBLIC symbol [, ...]

where symbol is defined in this assembly module to be a number, a variable, or a label (including PROC labels).

For example,

```
PUBLIC FOO, AXOLOTL, GET_RECORD
```

The PUBLIC directive itself is not permitted a label. Thus,

```
OVER_HERE: PUBLIC BAZ ; Not acceptable
```

is INVALID.

PUBLIC directives can appear on any line of an assembly module. Any symbol declared PUBLIC which is not defined is flagged as an error.

The EXTRN Directive

The EXTRN directive informs the assembler of the types, and, optionally, the segment attributes of certain symbols which are referenced within this module but are defined elsewhere, presumably in modules to be linked with this one. (Each must be declared as PUBLIC in the module which defines it.) The linkage process supplies the missing attributes — usually just the OFFSET, but potentially the SEG as well.

The format of this directive is:

EXTRN name:type [,...]

where "name" is the symbol defined elsewhere

and "type" must match the declaration of "name" in the module declaring it PUBLIC, and must be one of:

- BYTE, WORD, DWORD, structure name, or record name (for variables)
- NEAR or FAR (for labels or procedures)
- ABS (for pure numbers)

For example,

```
EXTRN FOO:WORD, BAZ:BYTE, COMPUTE_AVERAGE:FAR, LEN:ABS
```

where FOO is declared using DW or RECORD, BAZ is declared using DB or RECORD, COMPUTE_AVERAGE is declared using PROC FAR or LABEL FAR, and LEN appears as a value, for instance in an EQU directive (EQU is defined and described in Chapter 4).

As with the other linkage directives, EXTRN cannot itself have a name. Thus NOT_HERE EXTRN SIN is not allowed.

Placement of EXTRNs

Where you place an EXTRN for a given external symbol depends on whether you know which segment contains the definition of the given external symbol. If a given segment does not contain the definition of a symbol, do not place an EXTRN for that symbol in any part of the given segment.

If you know the segment in which a given external symbol (variable or label) is defined, declare it in an EXTRN directive inside another SEGMENT/ENDS pair using the same segment name. This sets the SEG attribute of all symbols in the EXTRN list to the current segment. The external variable or label can then be accessed in the same fashion as "normal" variables or labels.

If, however, you do not know the segment in which a given external symbol is defined, declare it in an EXTRN directive at the top of your program, outside all SEGMENT/ENDS pairs.

Then, to address an external symbol so declared outside all segments:

1. Load the 16-bit segment part into a segment register using the SEG operator (described in Chapter 4):

```
MOV AX, SEG FOO     ; Load segment base value into AX
MOV ES, AX          ; and thence to ES.
                    ; (NEVER load immediate values
                    ; directly into segment registers.)
```

2. AND:

   - EITHER: Use an ASSUME with "SEG external-name" where the segment name is normally required, e.g. ASSUME ES:SEG FOO. Then all subsequent references to FOO (e.g. MOV BX, FOO) should assemble correctly.
   - OR: Use a segment register override operator for each reference to that external (e.g. MOV BX, ES:FOO).

   In either case, make sure the segment register contains what you say it does.

A Systematic Way to Handle Externals

A good programming practice to follow for declaring external labels and variables is to create an INCLUDE file for each assembly module to contain the EXTRN declarations for the symbols declared PUBLIC therein. The INCLUDE file should contain SEGMENT PUBLIC/ENDS pairs for each segment and between them an EXTRN directive listing the variables (with their types) for that segment. In doing this, you make these variables and labels easily available to other assembly modules, since the externals can now be referenced as if they were ordinary locally-defined variables. For example:

```
DATA  SEGMENT  PUBLIC
      EXTRN    BUFFER_SIZE:WORD, BUFFER:BYTE, ERROR_STATUS:BYTE
DATA  ENDS

CODE  SEGMENT  PUBLIC
      EXTRN    ADD_TO_BUFFER:NEAR, OPEN_BUFFER:NEAR, CLOSE_BUFFER:NEAR
CODE  ENDS
```

The END Directive

The END directive identifies the end of the source program and terminates the assembler. The format is:

END [label-name]

where "expression" results in values for CS and IP. For example,

```
END START1
```

Exactly one END directive must appear in each assembly-language source file, and it must be the last source statement. If it is not, remaining statements are ignored.

If the optional label-name is present, it is used as the starting address for program execution. When several modules are to be linked together, only one can specify a starting address. That module is the main module.

The Location Counter ($) and the ORG Directive

The location counter is the assembly-time counterpart of the instruction pointer. That is, the location counter contains a value (symbolically represented by the dollar-sign ($)) that tells the assembler at what offset from the current segment to assemble the next instruction or data item. If a segment is reopened with a SEGMENT directive whose name-field was encountered before, the location counter is set to the value that was saved when the last ENDS directive was encountered for that segment.

You can set the location counter to a nonnegative number using the ORG directive. Its format is:

ORG expression

where "expression" is evaluated modulo 65536 and must not contain any forward references. You may include '$' (the current value of the location counter) in the expression; for instance,

```
ORG OFFSET $ + 1000
```

adds the decimal value 1000 to the current value of the location counter. The effect is to reserve 1000 uninitialized bytes from the last assembled byte.

It is not recommended that you specify values such as:

```
ORG OFFSET $ - 1000
```

since the effect would be to overwrite your last 1000 bytes of assembly (or to re-ORG high in the current segment, if $ - 1000 is negative).

The directive cannot be assigned a name or label; thus START: ORG 0 and SKIP ORG 2000 are both invalid.

CHAPTER 3
DEFINING AND INITIALIZING DATA

This chapter describes how you define and initialize data items in your MCS-86 Assembly Language program. Chapter 1 highlights several data definition and initialization features; it can be used as an adjunct to this chapter. Chapter 4 describes how to access data items as operands in instructions. Chapter 5 contains an encyclopedia of instruction mnemonics.

Identifiers

Identifiers are used to name entities within programs such as data items, segments, procedures, etc. An identifier has the following characteristics:

1. It starts with a letter or one of the three special characters:
   - Question mark (?), with hexadecimal value 3FH.
   - Commercial at (@), with hexadecimal value 40H.
   - Underscore (_), with hexadecimal value 5FH. (On some keyboards, this character is represented by a back-arrow.)
2. It may contain letters, digits, and the three cited special characters.
3. It is considered unique only up to 31 characters, but can be any length.

Data Items and Attributes

The data items you define form a basis for the operands you code to your program instructions and assembler directives, and therefore influence greatly the form your program ultimately takes. Code manipulates the data values, but the forms that you give data items, rather than values taken on, shape the code.

The various kinds of data items definable in a language are like different kinds of containers for data; choosing the "right" containers for your program data makes for more efficient and orderly transfer and manipulation of data, and hence better code.

The attributes of data items are analogous to the accessibility and intended use of containers.

This assembly language recognizes three basic kinds of data items—constants, variables, and labels. A constant is simply a name associated with a pure number which has no distinguishing characteristics other than its value. A constant has no attributes.

Variables identify data items that are manipulated; they form the operands of MOV, ADD, AND, MUL, etc.

Labels identify executable code; they form the operands of CALL, JMP, and the conditional jumps.

Variables and labels have distinguishing characteristics called attributes, which answer the questions:

- Where is the variable/label defined?

  Because of the addressing scheme of the 8086, this is a two-part question:

  1. In which SEGMENT is the variable/label defined?

2. What is the OFFSET of the variable/label within that segment? That is, how many bytes from the segment base does the definition of the variable/label reside? (In traditional assemblers, variables and labels are indistinguishable, and the offset of a variable/label is its only attribute.)

- How is the variable/label intended to be used?

  For a variable, this means the declared size, in bytes, and is called the TYPE attribute.

  For a label, this refers to whether segments other than the label's "home" segment can refer to it, and is called the DISTANCE attribute. (This concerns itself with size, since it really means, "Does a reference to the label require a 2-byte pointer or a 4-byte pointer?")

Data Definition Overview

You can define constants, variables, labels, structures, and records, Structures and records are special cases of variables.

- A CONSTANT is a pure number without attributes, for example:

```
FOO EQU 5 ;Let FOO represent the constant 5.
```

  This statement defines a value but no location or intended use for FOO, which can be assembled as a byte (8 bits), a word (2 bytes), or a doubleword (4 bytes), as needed. Thus the constant FOO has no attributes.

- A VARIABLE is defined to reside at a certain OFFSET within a specific SEGMENT, and is declared to reserve a fixed storage-cell TYPE—a byte, a word, a doubleword, or more (as for structures). For example, the statement:

```
BAZ DW 7 ;Declare BAZ a WORD having initial value 0007H.
```

  is assembled in the current segment at an offset equal to the value of the location counter ($), and reserves two bytes of memory. Notice that no colon is used to define a variable.

- A LABEL is also defined at a certain OFFSET within a specific SEGMENT (called here its "home" segment), and identifies executable code. If the label is referenced only from within its "home" segment, it can be declared to have a DISTANCE attribute of NEAR. A label can be implicitly declared by its presence, suffixed by a colon, in front of a line of code; such a label is always NEAR. If a CALL or JMP in another segment references the label, its DISTANCE attribute must be declared to be FAR. For example,

```
UPDATE_MASTER   LABEL   FAR     ; JMP/CALL here from other segments
  OPEN_M_FILE:  MOV     DX, BX  ; JMP/cond'l. jump/CALL here from this
                                ; segment.
                o
                o
                o
```

  Here the LABEL directive declares UPDATE_MASTER to be a FAR label, meaning that code in another segment references it, presumably in a CALL or JMP instruction. OPEN_M_FILE is also a label, since it is suffixed with a colon and precedes code, but it is a NEAR label and can only be referenced from this segment. UPDATE_MASTER and OPEN_M_FILE have the same segment and offset attributes, but different distance attributes, since OPEN_M_FILE is NEAR, and UPDATE_MASTER is declared FAR.

  Labels can also be declared using the PROC directive, described in Chapter 2. Chapter 4 describes various ways to CALL and JMP to labels.

The remaining kinds of data items—records and structures—are defined as variables, in terms of bytes, words, and doublewords.

Expressions can evaluate to a constant (3*4+7), a variable (TABLE + 5), or a label (GETBUF + 3); they are described at the end of Chapter 4.

Table 3-1 summarizes the attributes of data items.

Table 3-1. MCS-86™ Assembly Language Data Items

Data Item:	**VARIABLE**	**LABEL**
Identifies:	Data	Executable Code
SEGMENT	For both variables and labels, the SEGMENT attribute is the "base value" of the segment in which the variable or label is defined. This is a run-time value which must be in a segment register in order for the variable or label to be addressed. At assembly-time, the SEGMENT attribute is represented by a segment name; the assembler ensures run-time addressability of variables and labels by correlating ASSUME CS, DS, ES, and SS (and segment prefix) information with variable and label references. The 2-byte "base value" fields set up by the assembler can be filled in at assembly-time, link-time, locate-time, or run-time, but ultimately are destined for one of the segment registers. Segment "base values" for labels are destined for CS, while those for variables may eventually be loaded into CS, DS, ES, or SS. You can use the SEG operator (see Chapter 4) to isolate the locate-time 16-bit segment base value of a variable or label's segment.	
OFFSET	For both variables and labels, the OFFSET attribute is the 16-bit value representing the number of bytes from the base (start) of the segment in which the variable/label is defined, to the point of definition. Here the run-time value may be different from the assembly-time value, depending on the alignment and combine-type (see SEGMENT directive, Chapter 2) of the segment. You can use the OFFSET operator (see Chapter 4) to isolate the 16-bit offset value of a variable or label. When debugging, use LOC86 offsets, not ASM86 offsets.	
3rd Attribute:	TYPE	DISTANCE
Pertains to:	Declared size	Intrasegment reference(s)
Possibilities:	BYTE (or 1) WORD (or 2) DWORD (or 4) RECORD (1 or 2) STRUC (n)	NEAR if referenced only in segment in which it is defined FAR if referenced from any other segments
Defined using:	DB (for bytes) DW (for words) DD (for doublewords) record-name structure-name	Identifier followed by colon (:) preceding code is a NEAR label*. Declare FAR or (default) NEAR with: • LABEL directive • PROC directive

* Not quite always. The RECORD definition (in this chapter) uses a colon to separate field-name from field-width. Also, the EXTRN directive uses a colon to separate identifier and type/distance attribute. And ASSUME, of course, uses a colon to separate seg-reg from seg-name.

Constants

In general, constants can be binary, octal, decimal, hexadecimal, or ASCII, as shown in table 3-2.

Table 3-2. 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 77777Q
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'

Permissible Range of Values

The permissible range of values for constants is given with the individual data definition directives DB, DW, and DD. The maximum range of values a number can have is −0FFFFH through 0FFFFH. All assembly-time arithmetic operations are performed using signed two's complement arithmetic on 17-bit values.

Occurrence of Constants

Constants can appear as self-defining 8-bit or 16-bit values in an instruction, for instance:

```
MOV AH, 5       ; 8-bit immediate value.
MOV AX, 256     ;16-bit immediate value.
```

Or they can appear as values assigned to symbols, using the EQU directive:

```
FIVE EQU 5      ;5 used wherever FIVE referenced.
MOV AH, FIVE    ;Assembles the same as MOV AH, 5.
```

These constants are interpreted as decimal constants since no other base is specified. EQU is fully defined at the end of Chapter 4.

Basically, its format is:

symbol EQU expression

where expression can be any assembly-language item or expression. For example:

```
PARM1 EQU [BP].P1
```

Defining Variables (DB, DW, DD Directives)

The DB, DW, and DD directives can be used to define variables and/or initialize memory. Variable names should *NOT* be suffixed with a colon, as is common in many assembly languages. Thus:

```
VARIABLE   DW  678  ; No colon for DB, DW, or DD.
VARIABLE:  DW  678  ; ***ERROR***
```

These directives allocate and initialize memory in units of BYTES (8 bits), WORDS (2 bytes), and DWORDS (doublewords, or 4 bytes), respectively. If the optional variable-name field is present, a variable with that name is defined and given the following attributes:

- SEGMENT—the variable is associated with the current segment
- OFFSET—the variable is assigned the current offset from the start of the segment.
- TYPE
  - BYTE or 1, if DB is specified
  - WORD or 2, if DW is specified
  - DWORD or 4, if DD is specified

A wide variety of constructs is possible using the DB, DW, and DD directives. The general form is given first, embracing six forms in all:

Formats for DB, DW, and DD Expressions

1. Constant expression
2. Indeterminate initialization (the reserved symbol '?')
3. Address expression
4. ASCII character string of more than two characters (DB only)
5. Data-initialization list
6. Replicated values (a DUP clause)

Rather than studying the general form, you may find it more convenient to inspect the examples (under *FORMAT 1, FORMAT 2, ..., FORMAT 6*) following it.

General Form For DB, DW, and DD

The general form for the DB, DW, and DD directives is:

$$\text{[variable-name]} \quad \left\{\text{DB | DW | DD}\right\} \quad \left\{\begin{array}{l}\text{iexp [, ...]} \\ \text{exp' DUP (iexp [, ...])}\end{array}\right\}$$

where:

variable-name

is an identifier (colons are NOT permitted)

$$\left\{\text{DB | DW | DD}\right\}$$

means you must code **DB** or **DW** or **DD**, but only one of these

iexp [, ...]

means you must code at least one **iexp** (described below), and if you code more than one, you must separate them with commas

iexp

is one of the following:

1. A constant expression (see examples under ***FORMAT 1*** below)
2. The character **?** for indeterminate initialization (see examples under ***FORMAT 2*** below)
3. An address expression (see examples under ***FORMAT 3*** below)
4. An ASCII string longer than 2 characters (DB only, see examples under ***FORMAT 4*** below)
5. A data-initialization list (see examples under ***FORMAT 5*** below)
6. **exp' DUP (iexp [, ...])**

   specifies a replication count of "exp" copies of "iexp [, ...]", where exp' evaluates to a positive number and "iexp [, ...]' is as just described for the first case. (See examples under ***FORMAT 6*** below.)

Examples of DB, DW, DD Formats

Formats 1-6 below are keyed to forms 1-6 above.

FORMAT 1: INITIALIZING WITH A CONSTANT EXPRESSION

[variable-name] {DB | DW | DD} expression

- DB—DEFINE BYTE
  - ALLOCATION: A DB followed by a constant (other than a string) allocates one byte. (DB strings are described under ***FORMAT 4*** below. As many bytes are allocated for strings as characters specified.)
  - STRINGS: DB permits strings of any length. Examples are given under ***FORMAT 4*** below.
  - RANGE: DB accepts values in the range −256 through 255. Any value between −256 and −129 inclusive is stored as a nonnegative value between 0 and 127 inclusive. The mapping is: −129=127, ..., −255=1, −256=0.

- DW—DEFINE WORD
  - ALLOCATION: If DW is specified, one word (2 bytes) is allocated. Bytes are swapped within each word, with the least significant byte (LSB) occupying the lower-addressed byte, and the most significant byte (MSB) occupying the higher-addressed byte. Thus, 0AABBH is stored 0BBH in the least significant byte (LSB), followed by 0AAH in the most significant byte (MSB).
  - STRINGS: Two-character ASCII strings are inverted similarly. Thus, 'AB' is stored as 4241H. (DW 'BA' defines the same storage pattern as DB 'AB'.) Strings longer than two characters are not permitted with DW.
  - RANGE: DW accepts values in the range −65536 through 65535. Any value between −65536 and −32769 inclusive is stored as a nonnegative value between 0 and 32767 inclusive. The mapping is: −32769=32767, ..., −65535=1, −65536=0.

- DD—DEFINE DOUBLEWORD
  - ALLOCATION: If DD is specified, two words (4 bytes) are allocated.
  - STRINGS: The low-order word is assigned the same value as if a DW were specified. The high-order word is set to 0. ASCII strings of more than two characters are not permitted with DD.
  - RANGE: Same as DW.

Examples of Initialized Constant Expressions

In the following examples, note the absence of colons:

```
AB                      DB  'AB'          ; Stored as 4142H.
BA                      DW  'AB'          ; Stored as 4241H.
OFFAB                   DW  AB            ; Assembly-time offset of variable AB.
FEH                     DW  0FEH          ; Word having value 254.
OFF_FEH                 DW  OFFSET FEH    ; Offset fixed-up at locate-time.
SEG_FEH                 DW  SEG FEH       ; Segment fixed-up at locate-time.
                                          ; See Chapter 4 for OFFSET, SEG operators.
NUMBER                  DW  1234H         ; 34H at NUMBER, 12H at NUMBER+1.
PARM1                   DW  0FFFFH        ; 16 bits of 1's.
                        DB  100           ; Unnamed byte having value 01100100B.
MIDDLE_C                DW  523           ; Word having value 10BH.
INCHES_PER_MILE         DW  5280*12       ; Assembler performs arithmetic.
DEGREE_AT_EQUATOR       DW  25000/360     ; ASM86 computes (no rounding) 69 or 0045H.
MINS_PER_MAN_MONTH      DD  22*8*60       ; Assembler performs arithmetic.
```

FORMAT 2: DEFINING VARIABLES WITH INDETERMINATE INITIALIZATION

A single unquoted question mark (?) is a reserved symbol; you use it to tell the assembler that you do not care how memory is initialized. The values occupying cells initialized using the question mark are unpredictable; do not count on their being consistent from one assembly or run to the next, or even within the same assembly or run. When you code the question mark, you are in effect saying, "I do not require initialization here; moreover, I do not expect any consistency with respect to the initial contents of this cell at run-time."

In the following examples, note the absence of colons:

```
    SUM     DW    ?           ; Define and allocate a word, contents indeterminate.
            DW    ?           ; Allocate a nameless word, contents indeterminate.
WHATEVER    DB    ?           ; Define and allocate a byte, contents indeterminate.
LOTSA_DBS   DB    1000 DUP(?) ; 1000 bytes. See DUP clause under FORMAT 6 below.
```

FORMAT 3: INITIALIZING AN ADDRESS EXPRESSION (DW & DD ONLY)

[variable-name] { DW | DD } addr-expr

where "addr-expr" is subject to the following rules:

- Absolute numbers may always be added or subtracted from variables or absolute numbers. When a number is added to a variable, the result is a variable of the same type, having an offset equal to the sum of the number added plus the offset of the original variable within its segment.
- Variables cannot be added to variables or labels. Labels cannot be added to variables or labels.
- Variables and labels can be subtracted from variables and labels. The result is a pure number without attributes.

- For DW, the OFFSET part of a label or variable is the initial value assigned. This value is filled in at locate-time. The effect is as if an OFFSET operator (described in Chapter 4 under "Attribute Operators") were applied to the address expression. Be sure to refer to the description in Chapter 4 if you are using GROUPs.

- For DD, the effect is as if the low-order word were assigned the value of OFFSET of the address expression, and the high-order word were assigned the value of the SEG of the address expression. These values are filled in a locate-time. These operators are defined in Chapter 4. Be sure to refer to them if you are using GROUPs.

Examples of Initializing Using Address Expressions

In the following examples, note the absence of colons:

```
TABLE_OFFSET      DW  TABLE      ; 16-bit offset of TABLE.
STATUS_BYTE       DW  TABLE + 3  ; Offset of 4th byte in TABLE.
STATUS_PREFIX     DW  TABLE - 1  ; Offset of byte preceding TABLE.
DWORD_PTR_TABLE   DD  TABLE      ; 16-bit offset followed by16-bit segment base value.
                                 ; (Offsets and seg base values are filled in by
                                 ; LOC86).
```

FORMAT 4: DEFINING STRINGS LONGER THAN TWO CHARACTERS (DB ONLY)

DB permits strings of up to 255 characters. Successive characters occupy successively increasing locations. Thus, 'ABC' is stored as 414243H. Strings must be enclosed by single quotes ('). If you want to include a single quote in a string, code it as two consecutive single quotes.

```
LETTER        DB  'ABCDEFGHIJKLMNOPQRSTUVWXYZ'    ; 26 bytes allocated.
DIGIT         DB  '0123456789'
SPECIAL       DB  '?_@'
HEX_DIGIT     DB  '0123456789ABCDEF'
SINGLE_QUOTE  DB  ''''                            ; 1 byte allocated.
DATE          DB  '08/15/79'
QUOTE         DB  ''''
FORTUNE       DB  'FAINT HEART NEVER WON FAIR MAIDEN'
```

FORMAT 5: DEFINING AND INITIALIZING A DATA LIST

[variable-name] DB | DW | DD expr [, ...]

This directive initializes bytes, words, or doublewords in consecutive memory locations. Up to 16 items may be specified.

In the following examples, note the absence of colons:

```
PRIMES       DW 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53   ; Max. no. items 16.
FIBONACCI    DW 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610  ; 16 words.
WINNERS      DB 1, 5, 6, 11, 17, 19, 21, 27, 29, 31                          ; 10 bytes
CUBES        DW 0, 1, 8, 27, 64, 125, 6*6*6, 7*7*7, 8*8*8, 9*9*9             ; 10 words.
EQUIDISTANT  DW TABLE + 5, TABLE + 10, TABLE + 15                            ; Offset values, 3 words.
MESSAGE      DB 'HELLO, FRIEND.', 0DH, 0AH                                   ; 14-byte text plus <CR><LF>
```

FORMAT 6: REPLICATING INITIALIZATION VALUES (DUP)

The reserved symbol DUP specifies an "**iexp**" (or list of "**iexp**"s) to be repeated a number of times. The format is:

exp' DUP (iexp)

where **exp'** is the replication count (which must evaluate to a positive number), the parentheses must be coded, and **iexp** can be:

- An expression (numeric, or, if DW or DD, address as well)
- A question mark
- A list of items
- More DUP replications

```
ZIPS    DB  100 DUP (0)                        ; 100 bytes initialized to 0.
        DB  2 DUP (0, 3 DUP (1))               ; 8 unnamed bytes: 0,1,1,1,0,1,1,1.
NUMS    DW  100 DUP (5 DUP (4), 7)             ; 100 copies of words with values 4,4,4,4,4,7.
HI      DB  1000 DUP ('WELCOME', 0DH, 0AH)     ; Friendly sign-on with <CR><LF>'s.
```

Defining and Initializing Labels

Chapter 2 defines and describes labels more completely under the heading "The LABEL Directive". This is a summary included for completeness of this chapter.

Labels identify locations of executable code in your assembly, and as such can be the operands of jumps and calls. They have three attributes: SEGMENT, OFFSET, and DISTANCE. The DISTANCE attribute can be NEAR (if the label is referenced only from its "home" segment) or FAR (if any other segment references it).

A NEAR label is defined by its appearance, suffixed by a colon, preceding executable code, as in:

```
DO_IT_AGAIN: XOR AX, BUFFER[BX] ; Set AX=0 if AX=BUFFER[BX]
```

- Or by the presence of a LABEL directive specifying (or defaulting to) type NEAR:

```
ROUTINE_XYZ  LABEL  NEAR
             MOV    AX, DX
               o
               o
               o
```

- Or by the presence of a PROC directive specifying type NEAR:

```
COMPUTE_STAND_DEV  PROC  NEAR
```

(If no type is specified, NEAR is the default.)

Entry points can be declared as FAR labels using the LABEL or PROC directive:

```
      SIGMA    PROC  FAR
                 o

                 o
SIGMA_ENTRY LABEL  FAR
      DO_IT:   MOV   AX, [BX]   ; Label with colon is always NEAR.
                 o
                 o
                 o
      SIGMA    ENDP
```

Records

This section describes how to define a record, and how to allocate storage and initialize it for one or more records. Chapter 4 describes how to access records and record fields as operands. The sample program in Appendix K shows record definition, allocation, initialization, and access examples. Chapter 6 and Appendix A describe how how codemacros use records.

A record is a bit pattern you define in order to format bytes and words for bit- packing. The record definition itself does not allocate storage, however. You can allocate and initialize any number of 8- or 16-bit records using the definition's record-name as an assembly-time operator, as described below.

Figure 3-1 uses an example to outline record definition, initialization, allocation, and access.

The format of the record definition statement is:

record-name RECORD field-name : expression [=exp'] [, ...]

where:

- **record-name** is a unique identifier, and must be present
- Each **field-name** is a unique identifier, and must be present
- The colon (:) is coded as shown, between **field-name** and **expression**
- **expression** evaluates to a constant in the range 1 to 16, inclusive, and specifies the number of bits defined by **field-name**. (If symbols are present, they must not be forward references.)
- The sum of **expression**'s (record field-widths) must evaluate to a constant in the range 1 to 16 inclusive. If the sum is 8 or less, a record of width 8 bits is defined. If the sum is between 9 and 16 inclusive, a record of width 16 bits is defined. In either case, the user-defined record fields are right-justified (in the least significant bit-positions) of the byte or word. Word records format bits 0:7 in LSB, bits 8:15 in MSB.
- The optional clause **=exp'** gives a default value for the field. The default value has the following characteristics:
  1. It can be retained or overridden when storage is allocated using the record-name.
  2. If no default value is specified, zero is used.
  3. If specified, the default value:
     - Either evaluates to a positive integer expressible in the number of bits defined by the field. (For example, a field three bits wide can hold 7 (111B), but not 8 (1000B).)
     - Or, if the record-field is exactly 8 bits wide, it may be initialized to a character. The character so used must be enclosed in single quotes, as in: FIELDC:8='A')

For example, the record definition CHIPS defines a pattern as follows:

```
CHIPS RECORD RAM:7, EPROM:4, ROM:5
```

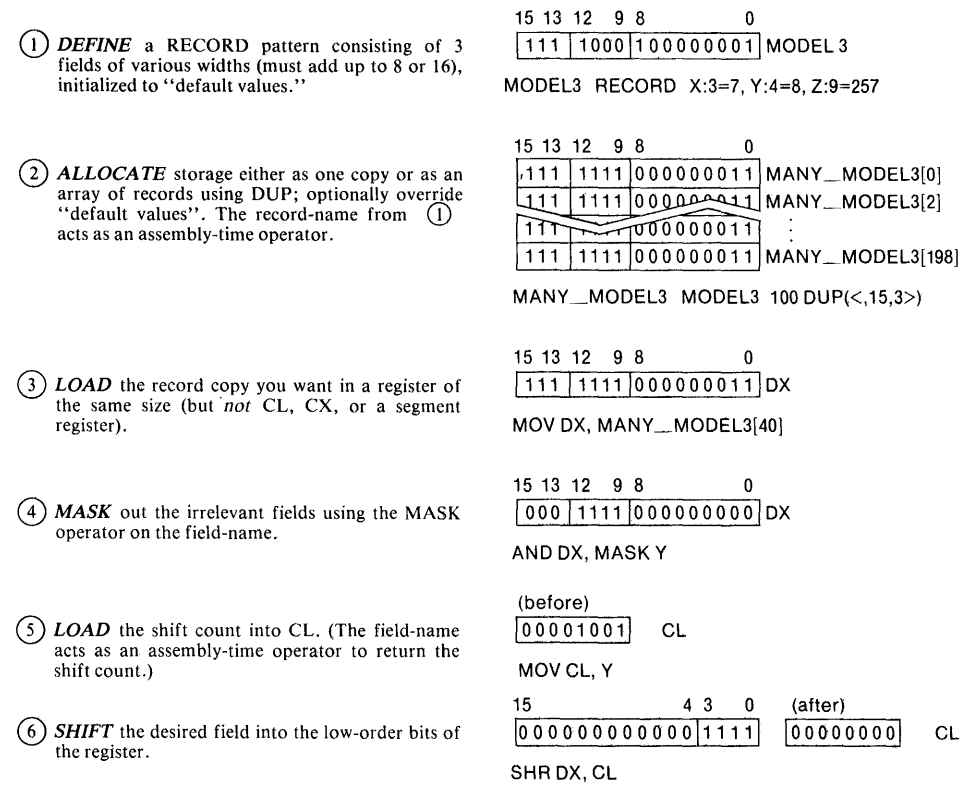
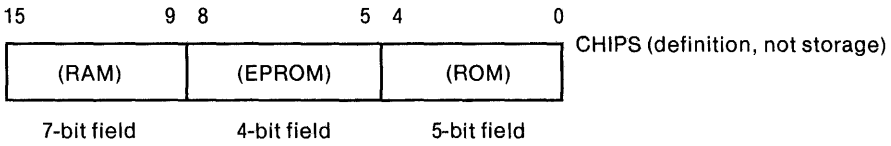


Figure 3-1. How to Define, Allocate, Initialize, and Access Records

Thus, when storage is allocated, the fields have the following characteristics:

Field Name	Field Width	Bit Positions	Maximum Value
RAM	7	15 — 9	2**7-1 = 127
EPROM	4	8 — 5	2**4-1 = 15
ROM	5	4 — 0	2**5-1 = 31

This pattern partitions 16 bits as follows:



Using the default-value clause of the definition, CHIPS2 could be defined as:

```
CHIPS2 RECORD RAM2:7=4, EPROM2:4=2, ROM2:5=25
```

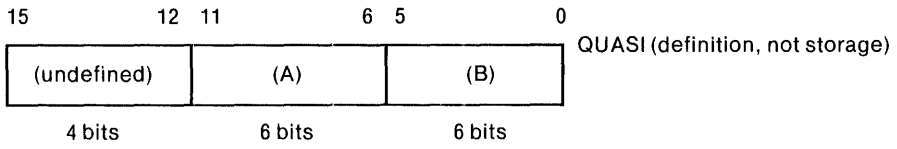
Both CHIPS and CHIPS2 define a record pattern consisting of (MSB to LSB) a 7-bit field, a 4-bit field, and a 5-bit field. The difference between them is that CHIPS2 is defined with default values for its fields, which become the initial values (if not overridden) when actual storage is allocated and initialized using the record-name CHIPS2 as an operator.

"Partial" Records

As mentioned in the description of the RECORD directive, if a record's user-defined fields do not occupy an entire byte or word, the assembler right-justifies the user-defined fields in the least-significant bit-positions of the byte or word defined by the record. Thus, the record definition:

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

formats a word as follows:



Record Allocation and Initialization

You can allocate and initialize records (after you have declared them) by using the record name as an operator. The general format is:

[name] record-name { < [exp] [, ...] > | exp' DUP (< [exp] [, ...] >) }

where:

[name]

denotes an optional name for the first byte or word (depending on the RECORD definition) of the allocated storage;

record-name

specifies the name (user-assigned) of the record from the RECORD directive defining the format and optional default field values

< [exp] [, ...] >

specifies a (possibly empty) list of field-initialization or optional override values such that trailing fields default, as in:

<> means retain originally defined values from RECORD directive.

<5> means override first record field initial value with 5.

<5,5> means override first and second fields' initial values with 5.

<,5> means override second field initial value with 5.

<,,5> means override third field initial value with 5.

<5,,5> means override first and third fields' initial values with 5.

<,5,5> means override second and third initial values with 5.

exp

specifies a constant, a constant expression, or the indeterminate initialization character ?. If a constant expression is coded, it must evaluate to a number expressible in binary within the width of the given record field, or else the number is truncated on the left.

exp' DUP (< [exp] [, ...] >)

where **exp'** is a constant or constant-expression specifying the number of copies of the record to be allocated. The replication count exp' must evaluate to a positive integer. The pattern of record field override values between angle-brackets (<>) is exactly as described above. The angle-brackets must be enclosed in parentheses, as shown.

Record Allocation/Initialization Examples

In order to contrast record initialization with/without default initial values, consider the two RECORD definitions used earlier, which define field-formatting, but do not allocate storage:

```
CHIPS   RECORD  RAM:7, EPROM:4, ROM:5      ; Fields are 7-bit, 4-bit, 5-bit.
```

and the similarly-defined, but with default values:

```
CHIPS2  RECORD  RAM2:7=4, EPROM2:4=2, ROM2:5=25      ; Default values 4, 2, 25.
```

Using CHIPS as an operator, storage can be allocated and formatted like CHIPS:

```
SYSTEM CHIPS 100 DUP(<127,11,31>)      ; Allocate 100 copies of CHIPS, override initial
                                       ; values
```

allocates 100 words and initializes each to the values RAM = 127, EPROM = 11, ROM = 31 in the format of CHIPS given above.

If, however, you have specified record-field default values in the RECORD directive, you can allocate and initialize using some or all of the default values. For instance, the directive:

```
SYSTEM2  CHIPS2  100 DUP(<>)      ; Allocate 100 copies of CHIPS2, keep default values.
```

allocates 100 words formatted as in the definition of CHIPS2 above, and initializes each word according to the default values specified in the definition of CHIPS2. Thus, for each copy, RAM2=4, EPROM2=2, and ROM2=25.

As indicated in the general form for record allocation and initialization, default values for record fields can be selectively overridden. Thus, a third array of records SYSTEM3 could be allocated and initialized to the default values RAM2 = 4 and EPROM2 = 2, while ROM2 could be overridden to 18 by specifying:

```
SYSTEM3  CHIPS2  100 DUP(<,,18>)      ;Override ROM2 default value, keep others.
```

Notice that leading commas must be present just as if values were specified. Any default values can be overridden this way. Thus, if 50 copies of words formatted like CHIPS2 and having initial values RAM = 4, EPROM = 3, ROM = 25 were required, the directive:

```
SYSTEM4  CHIPS2  50 DUP(<,3>)      ; Override EPROM2, keep others
```

Records in Expressions

A record can be used as part or all of an expression, as in the following:

```
R     RECORD    CHAR1:8, CHAR2:8
          o
          o
          o
      MOV       AX, R< 0ABH , 'C' >                ; Load AX with 0AB43H.
      MOV       BX, R< 5 , 7 > + R< 3 , 4 >        ; Load BX with 080BH.
      MOV       CX, R< 86H, 23H > XOR  R< 135, 35 > ; Load CX with 100H.
          o
          o
          o
```

Structures

This section describes how to define, allocate storage for, and initialize structures. Chapter 4 describes how to access data using structure-fields. The sample program in Appendix K shows structure definition, allocation, initialization, and access examples. Figure 3-2 outlines, using an example, how to define, allocate, initialize, and access structure-fields.

Structures enable you to define storage templates of offset values. The STRUC and ENDS statements define the extent of the storage template; between them, your DB, DW, and DD directives determine the spacing of offsets within the structure template definition.

The format of the STRUC/ENDS statement-pair and enclosed DB, DW, and DD directives is as follows:

```
structure-name   STRUC
                   o
                   o
                   o
[field-name]     { DB | DW | DD } { exp [, ... ]
                                  { exp' DUP ( exp [, ... ] )
                   o
                   o
                   o
structure-name   ENDS
```

where DB, DW, and DD expressions are as described earlier in this chapter, except that no forward references are permitted. It is essential that matching STRUC/ENDS pairs have the same structure-name. Field-names are optional, but must be unique identifiers.

An example of a structure follows:

```
S    STRUC
 F1 DB    0
 F2 DW    ?
 F3 DB    1, 2, 3
 F4 DD    TABLE
 F5 DW    100 DUP (5)
 F6 DB    '10/05/79'
 F7 DW    5, ?, 0EACH
S    ENDS
```

The structure-name S, when used to allocate storage, defines a variable of type 224 (the number of bytes it defines).

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
```

"VIRTUAL" STRUCTURE

15			0	
0	F	F	E	.FIRST
OFFSET (BUFFER)				.SECOND
0	5	0	7	.THIRD
*INDET		4	1	.FOURTH
0	1	0	1	.SIXTH

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.

B1 BLUEPRINT < >

0	F	F	E	B1.FIRST
OFFSET (BUFFER)				B1.SECOND
0	5	0	7	B1.THIRD
*INDET		4	1	B1.FOURTH
0	1	0	1	B1.SIXTH

B2 BLUEPRINT <,0,,,255>

0	F	F	E	B2.FIRST
0	0	0	0	B2.SECOND
0	5	0	7	B2.THIRD
F	F	4	1	B2.FOURTH
0	1	0	1	B2.SIXTH

B3 BLUEPRINT 5 DUP (<,,,,50>)

0	F	F	E	B3.FIRST[0]
OFFSET (BUFFER)				B3.SECOND[0]
0	5	0	7	B3.THIRD[0]
3	2	4	1	B3.FOURTH[0]
0	1	0	1	B3.SIXTH[0]
0	F	F	E	B3.FIRST[10]
OFFSET (BUFFER)				B3.SECOND[10]
3	2	4	1	⋮
0	1	0	1	B3.SIXTH[30]
0	F	F	E	B3.FIRST[40]
OFFSET (BUFFER)				B3.SECOND[40]
0	5	0	7	B3.THIRD[40]
3	2	4	1	B3.FOURTH[40]
0	1	0	1	B3.SIXTH[40]

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. How to Define, Allocate, Initialize, and Access Structures

Initial (Default) Values for Structure Fields

Any initial values specified in the DB, DW, or DD directives specify default values for the structure fields. They have no immediate effect, since no storage is allocated by the STRUC/ENDS pair.

However, if storage is subsequently allocated using the structure-name as an operator, these values can be used to initialize the allocated storage, and are called structure-field default values.

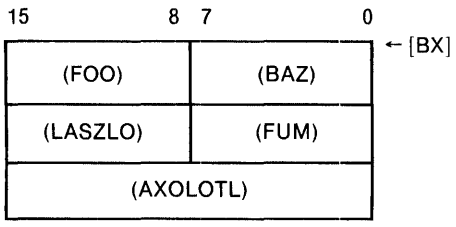
Overridable (Simple) Structure Fields

Certain structure-field default values can be overridden when storage is allocated; these overridable fields are called simple fields. A simple field can be defined by a DB, DW, or DD, but it cannot be a multiple specification (e.g. a list or a DUP clause). An exception to this restriction is that a DB character string is overridable. Several examples should clarify what is overridable, and what is not:

```
SUPER  STRUC
       DB  ?              ; A simple field, overridable.
       DB  1, 2, 3        ; A multiple field, not overridable.
       DW  10 DUP (?)     ; A multiple field, not overridable.
       DB  'MESSAGE'      ; A long simple field, overridable by a string.
       DD  TABLE          ; A simple field, overridable.
       DW  TAB - 8        ; A simple field, overridable.
       DW  TAB, TAB + 4   ; A multiple field, not overridable.
       DB  ?, 2, 3        ; A multiple field, not overridable.
SUPER  ENDS
```

Example of Structure Definition

A structure is a convenient way to group a collection of byte and word offsets so that they may be referenced from any base. Suppose, for example, that your subroutine is passed the address of a control block in register BX, and you wish to reference its various fields:



By defining a structure of the same form in your subroutine and assigning the same field-names to its fields, you set up an offset-tracking mechanism at assembly-time:

```
BX_BLOCK   STRUC
  BAZ      DB   100     ; Default values can be overridden
  FOO      DB   0       ; later in this assembly.
  FUM      DB   0FFH
  LASZLO   DB   0
  AXOLOTL  DW   3
BX_BLOCK   ENDS
```

Storage need not be allocated for structure offsets to be referenced, as in the following reference to .FUM:

```
MOV AL, [BX].FUM
```

The anonymous variable [BX].FUM assembles to type BYTE, and in this case references the byte at offset (BX)+2 within the DS-addressed segment.

On the other hand, a reference to .AXOLOTL, as in:

```
ADD DX, [BP][SI].AXOLOTL
```

The anonymous variable [BP] [SI].AXOLOTL assembles to type WORD, and in this case references the word at offset (BP)+(SI)+4 within the SS addressed segment.

Structure Allocation and Initialization

You can allocate and initialize structures (after you have declared them) by using the structure-name as an operator. The general format is similar to that for record initialization:

[name] structure-name $\left\{ \begin{array}{l} \text{< [exp] [, ...] >} \\ \text{exp' DUP (< [exp] [, ...] >)} \end{array} \right\}$

where:

[name]

denotes an optional name for the N bytes defined by the structure (depending on the STRUC/ENDS definition) of the allocated storage

structure-name

specifies the user-assigned name from the STRUC/ENDS directives enclosing the DB | DW | DD directive(s).

< [exp] [, ...] >

specifies a (possibly empty) list of field-initialization or optional override values such that trailing fields default, as in:

- <> means retain originally defined values from STRUC/ENDS directives.
- <5> means override first structure-field initial value with 5.
- <5,5> means override first and second structure-fields' initial values with 5.
- <,5> means override second structure-field initial value with 5.
- <,,5> means override third structure-field initial value with 5.
- <5,,5> means override first and third structure-fields' initial values with 5.
- <,5,5> means override second and third structure-fields' initial values with 5.

exp

specifies a constant, a constant expression, a string, or the indeterminate initialization character ?.

If a string is used:

1. Strings of 1 or 2 characters can be used anywhere an expression can be used as an override (2-character strings cannot override a byte).
2. Strings of more than 2 characters can override only DB fields containing strings, and then only when the overriding string is no longer than the string to be overridden. If a shorter string is given, it is padded out using the MSBs of the default string. If a longer string is given as an override, it is truncated to the field length.

exp' DUP (< [exp] [, ...] >)

where **exp'** is a constant or constant-expression specifying the number of copies of the structure to be allocated. The replication count **exp'** must evaluate to a positive integer. The pattern of structure field override values between angle-brackets (<>) is exactly as described above. The angle-brackets must be enclosed in parentheses, as shown.

Recalling the definition of BX__BLOCK:

```
BX__BLOCK    STRUC
  BAZ        DB   100     ; Default values can be overridden
  FOO        DB   0       ; later in this assembly.
  FUM        DB   0FFH
  LASZLO     DB   0
  AXOLOTL    DW   3
BX__BLOCK    ENDS
```

Storage is allocated using BX__BLOCK, the user-assigned name, and the default values specified above can be defaulted or overridden:

```
BX__BLOCK1     BX__BLOCK  5 DUP (<>)          ; Make 5 copies, keep default values.
BX__BLOCK2     BX__BLOCK  <,7>                ; Make 1 copy, override FOO=0 with
                                              ; FOO=7.
BX__BLOCK3     BX__BLOCK  <1,2,3,4>           ; Make 1 copy, override all values as
                                              ; shown.
MANY__BX__BL   BX__BLOCK  1000 DUP (<,,,,0>)  ; Override AXOLOTL=0 in 1000 copies.
```

CHAPTER 4
ACCESSING DATA (OPERANDS AND EXPRESSIONS)

This chapter describes how to specify operands in the context of various instructions and directives. Operands of the string instructions (MOVS, MOVSB, MOVSW, etc.) are special cases, and are described in Chapter 2.

Also described (at the end of this chapter) are expressions, operator precedence (hierarchy), and the EQU directive.

Operands: Immediate, Register, Memory

The 8086 instruction set (described in Chapter 5) provides several different ways to address operands. Most two-operand instructions allow either memory or a register to serve as the first operand (the "destination"), and either memory, a register, or a constant within the instruction to serve as the second operand (the "source"). Memory-to-memory operations are excluded.

Operands in memory can 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.

The result of a two-operand operation may be directed to either memory or a register. Single-operand operations are applicable uniformly to any register or memory operand (but not immediate constants; for instance, there is no Push Immediate instruction). Virtually all 8086 operations may specify either 8- or 16-bit operands.

Operands are described here as follows:

- Immediate Operands
- Register Operands
- Memory Operands
  - Labels (Defined in Chapter 2)
  - JMP/CALL Operands (Labels and Variables)
  - Variables (Defined in Chapter 3)
    - Simple
    - Indexed (Subscripted Arrays)
    - Structures
- Attribute Operators
  - Attribute-Overriding Operators
    - The Pointer (PTR) operator
    - Segment Override
    - The SHORT Operator
    - The THIS Operator
    - The HIGH and LOW Operators
  - Value-Returning Operators
    - The OFFSET, SEG, and TYPE Operators
    - The LENGTH and SIZE Operators
  - Record-Specific Operators
    - Shift Count (Field Name)
    - The MASK Operator
    - The WIDTH Operator

Immediate Operands

Every two-operand instruction except multiply, divide, and the string operations can specify an immediate value as an expression as the rightmost (source) operand. The general forms are:

[label:] mnemonic memory-reference, expression [;comment]

and

[label:] mnemonic register, expression [;comment]

where:

- **label** (optional) is an identifier
- **mnemonic** is any two-operand mnemonic (e.g. MOV, ADD, XOR)
- **memory-reference** is as defined under "Memory Operands" in this chapter, and is summarized by:
  - Direct 16-bit offset address
  - Indirect through BX or BP, optionally with an 8-bit or a 16-bit displacement
  - Indirect through SI or DI, optionally with an 8-bit or a 16-bit displacement
  - Indirect through BX plus SI or DI, or through BP plus SI or DI, optionally with an 8-bit or a 16-bit displacement.
- **register** is any general-purpose register (not a segment register)
- **expression** is as defined under "Expressions" at the end of this chapter. Rules of formation of constants are given in Table 3-2, Chapter 3.

The assembler determines if the destination is of type BYTE or WORD, evaluates the expression using 17-bit arithmetic, and, if the destination operand can accommodate the result, encodes the value of the expression using 2's complement arithmetic as an 8-bit or 16-bit field (depending on the type, BYTE or WORD, of the destination operand) in the instruction being assembled. Sixteen-bit values are assembled low-order-byte-first.

Examples of Addressing Modes Using Immediate Values

```
MOV  AL, 3                      ; 8-bit immediate value to register.
ADD  BX, 5                      ; 16-bit immediate value to register.
XOR  WORD PTR PROFILE, 0F0H     ; 16-bit imm. val. to 16-bit direct offset.
AND  BYTE PTR PROFILE, 101B     ; 8-bit imm. val. to 16-bit direct offset.
SUB  WORD PTR [BX], 777Q        ; 16-bit imm. val. through BX, no displacement.
ADD  BYTE PTR [BP], 99          ; 8-bit imm. val. through BP, no displacement.
AND  FOO[BX], 0F0FH             ; 16-bit imm. val. (assuming FOO to be type WORD here)
                                ; through BX, 16-bit displacement.
OR   BYTE PTR FOO[BX], 0F0H     ; 8-bit imm. val. through BX, 16-bit disp.
XOR  WORD PTR [BX][SI], 0F3F1H  ; 16-bit imm. val. through BX+SI, no disp.
XOR  BYTE PTR [BX + DI], 0F1H   ; 8-bit imm. val. through BX+DI, no disp.
AND  BYTE PTR [BP+SI+1], 0F3H   ; 8-bit imm. val. through BP + SI,
                                ; 8-bit displacement for +1 in [].
AND  FOO[BX + DI + 200], 1      ; 16-bit imm. val. through BX+DI+(FOO+200).
```

Register Operands

Registers are as follows:

- Segment 16-bit (CS, DS, SS, ES)
- General 16-bit (AX, BX, CX, DX, SP, BP, SI, DI)
- General 8-bit (AH, AL, BH, BL, CH, CL, DH, DL)
- Pointer and Index 16-bit (BX, BP, SI, DI)
- Flag 1-bit (AF, CF, DF, IF, OF, PF, SF, TF, ZF)

Segment registers contain segment base values. These registers are programmer- initialized, and beyond that are of little concern to the programmer except as described in Chapter 2 under "Anonymous References".

Each of the general 8-bit, general 16-bit, and pointer and index 16-bit registers can participate in arithmetic and logical operations. Thus, although AX is frequently referred to as "the accumulator", the 8086 has eight distinct 16-bit accumulators (AX, BX, CX, DX, SP, BP, SI, DI) and eight distinct 8-bit accumulators (AH, AL, BH, BL, CH, CL, DH, DL), although each 8-bit accumulator is either the high-order byte (H) or the low-order byte (L) of AX, BX, CX, or DX.

The flags are updated after each instruction to reflect conditions detected in the processor or any accumulator. The instruction encyclopedia of Chapter 5 lists the flags affected for each instruction.

Appendix C describes flag operation. Appendix J lists how each instruction affects the flags.

The flag-register mnemonics stand for:

AF — Auxiliary-carry	PF — Parity
CF — Carry	SF — Sign
DF — Direction	TF — Trap
IF — Interrrupt-enable	ZF — Zero
OF — Overflow	

Registers as Explicit Operands

Two-operand instructions explicitly specifying registers take the forms:

- Register to register

  [label:] mnemonic reg , reg [;comment]

  For instance:

  ```
  ADD AX, SI ; AX = AX + SI
  ```

- Immediate to register

  [label:] mnemonic reg , imm [;comment]

  For instance:

  ```
  AND BH, 40H ; Mask out all but Bit 6.
  ```

- Memory to register

  [label:] mnemonic reg , mem [;comment]

  For instance:

  ```
  LOAD_STRUC_FIELD: MOV AX, [DI].AXOLOTL ; Assuming .AXOLOTL type WORD
  ```

- Register to memory

  [label:] mnemonic mem , reg [;comment]

  For instance:

  ```
  ADD FOO[BX][SI], DI ; FOO[BX + SI] = FOO[BX + SI] + (DI)
  ```

  One-operand instructions explicitly specifying registers take the form:

  [label:] mnemonic reg [;comment]

  For instance:

  ```
  DEC SI ; SI = SI-1
  ```

Segment Registers

The segment registers CS, DS, ES, and SS are described in Chapter 2, with special attention to their manipulation under the heading "Loading Segment Registers".

Segment registers may also be specified in operands as segment prefixes either to override an ASSUME for a named variable, or to override a hardware-default segment for an anonymous reference (e.g. [BX]). This source fragment shows both uses:

```
SEG1   SEGMENT
         ASSUME   CS:SEG1,
         MOV      AX, DS:BAZ      ; Without DS:, BAZ gets ASM86 error message.
           o                      ; (Assuming DS contains DATA1.)
           o                      ; (See Chapter 2 for more information.)
           o
         ADD      SI, CS:[BX]     ; Without CS:, [BX] defaults to DS-segment.
           o                      ; (See "Anonymous References" in Chapter 2.)
           o
           o
SEG1   ENDS
DATA1  SEGMENT
           o
           o
           o
  BAZ    DW (?)
           o
           o
           o
DATA1  ENDS
```

These topics are covered in greater detail in Chapter 2.

Pointer and Index Registers

The pointer (BX, BP) and index (SI, DI) registers are 16-bit registers that can participate in logical and arithmetic operations. They are distinguished, however, in their use in addressing modes, as described in Chapter 2 and summarized here:

- Anonymous references, for instance:

```
MOV   AX, [BX]        ; Move word at DS:(BX) to AX.
MOV   AX, [BP]        ; Move word at SS:(BP) to AX.
MOV   AX, [BX][SI]    ; Move word at DS:(BX + SI) to AX.
MOV   AX, [BX][DI]    ; Move word at DS:(BX + DI) to AX.
MOV   AX, [BP][SI]    ; Move word at SS:(BP + SI) to AX.
MOV   AX, [BP][DI]    ; Move word at SS:(BP + DI) to AX.
```

  Since no named variable is specified, segments are hardware-defaulted unless overridden by segment prefixes. See "Anonymous References" in Chapter 2 for more information.

- Indexed variable references, for instance:

```
MOV   AX, FOO[BX]     ; Move word in SEG FOO:((OFFSET FOO) + (BX)) to AX.
MOV   AX, FOO[BP]     ; Move word in SEG FOO:((OFFSET FOO) + (BP)) to AX.
etc.
```

  Here the named variable, in conjunction with the ASSUME directive, determines the segment. See Chapter 2 for more information. The SEG and OFFSET directives are described later in this chapter, under "Value-Returning Operators".

General Registers; H and L Group

When one operand of a two-operand instruction specifies a 16-bit general register, or pointer/index register, the other operand of the instruction:

- If memory, must be a WORD reference, as in:

```
MOV DI, FOO ; FOO declared type WORD.
```

- If register, must be a WORD register, as in:

```
MOV DX, DI ; Both 16-bit registers.
```

- If immediate, must evaluate to an 8-bit or a 16-bit quantity:

```
MOV SI, 5 ; SI = 0000000000000101B
```

or

```
MOV SI, 0FFFFH ; SI = 1111111111111111B
```

Similarly, when one operand of a two-operand instruction specifies an 8-bit general register (AH, AL, BH, BL, CH, CL, DH, DL), the other operand of the instruction:

- If memory, must be a BYTE reference, as in:

```
ADD DL, BYTE PTR FOO ; Pick up byte at FOO, not WORD (as declared).
```

- If register, must be a BYTE register, as in:

```
XOR BL, AH ; BL = (BL OR AH) AND NOT (BL AND AH) ; Registers must be same size.
```

- If immediate, must evaluate to an 8-bit value, as in:

```
AND DH, 0F1H ; Mask out Bits 1:2:3.
```

The general registers, pointer and index registers, and H and L group (8-bit subregisters of AX, BX, CX, and DX) are described in detail in the *MCS-86™ User's Manual* and the *8086 Family User's Manual.*

Registers as Implicit Operands

Some instructions use registers implicitly:

Instruction	Implicitly Uses
AAA, AAD, AAM, AAS	AL, AH
CBW, CWD	AL, AX or AX:DX
DAA, DAS	AL
IN, OUT	AL or AX
MUL, IMUL, DIV, IDIV	AL, AX or AX:DX
LAHF, SAHF	AH
LES	ES
LDS	DS
Shifts, Rotates	CL
String	CX, SI, DI
XLAT	AL, BX

Flag Registers

The 1-bit flag registers are never specified as operands; they are manipulated either by flag instructions (e.g. STC, CLC, CMC) or by instructions which implicitly affect them (e.g. INC, DEC, ADD, MUL, DIV).

Appendix C describes flag operation; Appendix J provides a summary of how flags are affected by each instruction.

Memory Operands

A memory operand is either a label or a variable; Chapter 2 defines both.

JMP and CALL Operands (Variables, Labels, Registers, and Address Expressions)

The operands of JMP and CALL can be a label, a variable, a register, or an address expression. In what follows, the "JMP/CALL target" means the code location to receive CPU control as a result of a JMP or CALL.

Jumps and calls can be direct or indirect:

- The operand of a direct JMP/CALL is a label identified with the JMP/CALL target. The format is:

  [name:] JMP | CALL label [;comment]

  For instance:

  ```
  JMP GET_CHAR  ; GET_CHAR identifies code.
  ```

- The operand of an indirect JMP/CALL is not the JMP/CALL target itself, but is instead a WORD or DWORD pointer to the JMP/CALL target. (A WORD pointer consists of a 16-bit offset; a DWORD pointer consists of a 16-bit offset followed by a 16-bit segment base value. If you use DW and DD respectively to define these pointers, LOC86 fills in the fields at locate-time. The assembly-time values in your ASM86 listing do not reflect these LOC86-assigned values.) The format is:

  [name:] JMP | CALL addr-expr [;comment]

  where **addr-expr** can be:

  - A register containing the offset (in the current CS-addressed segment) of the JMP/CALL target, as in:

    ```
    JMP BX  ; Pass control to CS:(BX).
    ```

  - A WORD variable containing the offset (in the current CS-addressed segment) of the JMP/CALL, as in:

    ```
            CALL    NEAR_LABEL_ARRAY[DI]    ; References entry in
                                            ; array of routine addresses.
              o
              o
              o
    NEAR_LABEL_ARRAY  DW  ROUTINE1    ; Offset of ROUTINE1 in this segment.
                      DW  ROUTINE2    ; Offset of ROUTINE2 in this segment.
                        o
                        o
                        o
    ```

    Use caution when indexing arrays. No matter what the type (WORD or DWORD), the expression in square-brackets (DI above) is interpreted as the number of BYTEs into the array. Remember too that the first entry begins at byte 0, not 1.

  - A DWORD variable pointing to the JMP/CALL target in any segment, as in:

    ```
            CALL    FAR_LABEL_ARRAY[BX]     ; References DWORD entry (see
                                            ; declaration below).
              o
              o
              o
    ```

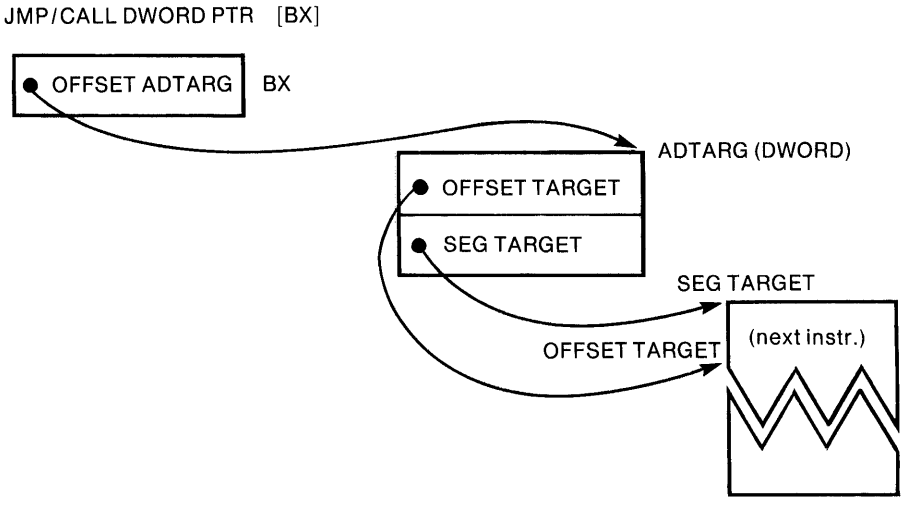
```
FAR_LABEL_ARRAY  DD  FAR_PROC1     ; DWORD is 16-bit OFFSET,
                                    ; 16-bit SEG.
                 DD  FAR_PROC2     ; Values are LOC86-supplied.
                  o
                  o
                  o
```

The same caution applies here concerning indexed arrays: subscripts are treated as BYTE displacements.

- The expression [BX], which is treated as a pointer to a WORD (offset) or DWORD (offset and segment) variable, which in turn points to the JMP/CALL target (thus embodying two levels of indirection), as in:

```
                 MOV   BX, OFFSET TARG1_ADDR  ; Load offset of pointer
                                              ; to NEAR routine in BX.
                  o
                  o
                  o
                 JMP   WORD PTR [BX]          ; Go to WRITE_BUFFER
                                              ; in this segment.
                  o
                  o
                  o
                 MOV   BX, OFFSET TARG2_ADDR  ; Load offset of pointer
                                              ; to FAR procedure.
                  o
                  o
                  o
                 CALL  DWORD PTR [BX]         ; Execute FAR procedure
                                              ; CLOSE_MASTER.
                  o
                  o
                  o
TARG1_ADDR       DW    WRITE_BUFFER           ; LOC86-supplied offset
                                              ; value.
TARG2_ADDR       DD    CLOSE_MASTER           ; LOC86-supplied offset
                                              ; value followed by
                                              ; LOC86-supplied
                                              ; segment value.
```

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



The type of JMP or CALL generated by the assembler depends on how much the assembler knows about the target. Table 4-1 shows the possibilities.

Table 4-1. Assembler-Generated Jumps and Calls

	Target Label Declared NEAR	Target Label Declared FAR
Jump/Call Assembled Under Same ASSUME CS:	NEAR Jump/Call	NEAR Jump FAR Call
Jump/Call Assembled Under Different ASSUME CS:	***ERROR***	FAR Jump FAR Call

Using the SHORT Operator. When a jump is to a location within the same segment and the relative displacement of the jump lies within the range −128 to 127 bytes, and the target label has not yet been defined, you can save a byte by coding the SHORT operator before the target label in the instruction, as in:

```
        JC   SHORT WRAPUP   ; Distance from end of jump
          o
          o
          o
WRAPUP: PUSH   BX           ; to here < 128 bytes.
```

In a case like this, the label operand is assembled as a 1-byte signed (2's complement) displacement. If you code the SHORT operator and the displacement is outside the range [−128, 127], an error message is issued.

Implicit SHORT Jumps/Calls. If a NEAR jump/call is to a label already encountered by the assembler (including expressions using $, the location counter), and the relative displacement is within the range [−128, 127], the jump/call is assembled with a 1-byte displacement just as if you had coded the SHORT operator, as in:

```
TRY_IT: XOR   FOO[BX], AX    ; If distance < 129 from end of jump
          o
          o
          o
        JNZ   TRY_IT         ; jump is assembled
                             ; with 1-byte self-relative displacement.
```

Variables

Before reading this section, you should be familiar with how to define and initialize variables, as described in Chapter 3. This section describes how to specify the following kinds of variables as operands to instructions:

- Simple
- Indexed
- Structures

Records are described under "Record-Specific Operators" in this chapter.

Anonymous variables (indirect memory references which do not specify a user-named symbol) are discussed in Chapter 2 under "Anonymous References". Examples of anonymous variables are in the following:

```
MOV  CX, [BP][SI]
AND  AX, ES:[BX + DI]
XOR  BX, DS:[BP - 1]
```

Simple Variables. As an operand, a simple variable is an unmodified identifier which is used the same way it is declared. For example,

```
SIMP1  DB   5
SIMP2  DW   5
       o
       o
       o
       MOV  AL, SIMP1    ; Move byte to byte register.
       MOV  AX, SIMP2    ; Move word to word register.
```

To mix types within an instruction (moving a byte to a word register, for instance) requires a type-overriding operator (in this case PTR). Type-overriding is described under "Attribute Overriding Operators" in this chapter.

Indexed Variables. An indexed variable is a simple variable suffixed by square-brackets which enclose:

- A constant or a constant expression (e.g [5])
- A base register (BX or BP) or an index register (SI or DI)
- A base or index register plus or minus a constant expression (in any order)
- A base register plus an index register plus or minus a constant or constant expression (in any order)

Indexed variables are similar to the high-level language concept of subscripted variables or arrays, with three important distinctions:

1. Indexing is zero-based; thus, FOO[0] is the same as FOO (FOO[1] is the second byte in the array)
2. The offset of an indexed variable is the sum of:
   - The offset of the simple variable, plus
   - The number of BYTES (no matter what type the variable is) that the square-bracketed expression evaluates to.
3. The type of the resulting operand is the same as the type of the simple variable (BYTE or WORD)

Thus, for example, if you have defined:

```
FOO DW 500 DUP (?)
```

and later in your program, when these values have meaning, you reference memory as follows:

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

Then the value of BX does not select the corresponding element of the array FOO; it selects the word beginning BX bytes past the offset of FOO.

Expressions as Subscripts. You can use expressions as subscripts to index arrays, for instance:

```
ADD FOO[100*TYPE FOO], 9   ; Add 9 to 101st element of FOO.
```

Using TYPE (which returns 2 if FOO is declared WORD, etc.), you need not concern yourself with counting bytes, so long as you remember that indexing is zero-based (the first element begins at FOO[0]).

The TYPE operator is described later in this chapter.

Double-Indexed Variables. A double-indexed variable is of the form:

simple-variable [base][index]

or

simple-variable [index][base]

with the restrictions that:

- Base must be:
  - A constant or a constant expression
  - BX or BP
  - BX or BP plus or minus a constant or a constant expression in any order
- Index must be:
  - A constant or a constant expression
  - SI or DI
  - SI or DI plus or minus a constant or a constant expression in any order

The effect of double-indexing is the same as if index and base were summed between a single pair of square-brackets. Thus,

```
FOO[BX-5][SI+3]
```

is the same as:

```
FOO[BX+SI-2]
```

which in fact is acceptable. So is

```
FOO[-2+BX+SI]
```

Structures. Each structure field defined within a structure defines a type, and a partial offset value within the structure. This value is simply the number of bytes from the beginning of the structure to the beginning of the structure field.

The partial offset value and TYPE defined by a structure field are referenced by the structure field name prefixed by a dot (.) or period. Thus, if S is a structure defined by:

```
S     STRUC
  A     DB  0
  B     DW  0
  C     DD  0
  D     DW  0
  E     DB  0
S     ENDS
```

Then a reference to .A results in a partial offset value of 0, to .B results in 1, to .C in 3, to .D in 7, and to .E in 9.

These structure field references can be suffixed to any memory reference.

The memory reference M.F, where F is a structure field, has the following attributes:

- Segment — the same as M
- Offset — the offset of M plus the partial offset value of F
- Type — the same as F

The following examples use partial offset values from the structure S defined above:

Operand	Segment	Offset	Type
FOO.C	same as FOO	OFFSET(FOO) + 3	DWORD
FOO[BX].D	same as FOO	OFFSET(FOO) + BX + 7	WORD
[BX].B	DS-addressed	BX + 1	WORD
[BP - 4].E	SS-addressed	BP + 5	BYTE

Using Structures in Forward/Backward-Linked Lists. As a further example of structures, consider the simple definition:

```
LINK    STRUC
  TO    DW      0
  FROM  DW      0
  INFO  DB      96 DUP (?)
LINK    ENDS
```

This structure template defines three fields:

- TO, a word to point to the next link in the list
- FROM, a word to point to the preceding link in the list
- INFO, a 96-byte field for application-specific information

An array of 500 copies of LINK can be allocated as follows:

```
CHAIN LINK 500 DUP(<>)   ; 500 copies of LINK.
```

At run-time, CHAIN can easily be transformed into a forward-and-backward linked list. The following sequence of code shows how to initialize the FROM fields (with the first containing zero):

```
        MOV   BX, OFFSET CHAIN      ; Point BX at CHAIN[0].
        MOV [BX] .FROM, 0           ; First .FROM
        MOV   SI, TYPE CHAIN        ; Number of bytes in one structure (100 here).
        MOV   CX, LENGTH CHAIN - 1  ; Number of structure copies - 1.
PLUG:   MOV [BX + SI].FROM, BX      ; Put ptr. to CHAIN[n] in CHAIN[n + 1].FROM.
        ADD   BX, SI                ; Bump BX by no. bytes in structure.
        LOOP  PLUG                  ; CX = CX-1, then jump if CX <> 0.
```

The TO fields can be filled in similarly.

Linked lists provide an efficient means of storing/retrieving ordered items in a list that is frequently updated, such as a list of control blocks for prioritized tasks, or a queue of users contending for a system resource.

Attribute Operators

Just as indexing, structure, arithmetic and logical operators can appear in operands, so may a class of operators termed attribute operators appear. Attribute operators can be used to:

- Override (alter for one instruction) an operand's attributes
- Yield the values of operand attributes
- Isolate record fields

Attribute-Overriding Operators

It is sometimes necessary to override the segment, offset, and type or distance of variables or labels in order to use them more efficiently, or to access them in several ways. The assembly language provides attribute-overriding operators so that this degree of freedom can be achieved. These operators are:

- PTR (pointer) -- overrides the type (BYTE, WORD, DWORD) of a variable or the distance (NEAR, FAR) of a label
- Segment override -- overrides the segment of a variable
- SHORT -- overrides NEAR or FAR for very short jumps/calls
- THIS -- creates an operand of any type or distance at an offset equal to the current value of the location counter
- HIGH and LOW -- for 8080 assembly language compatibility

PTR—the Pointer Operator. PTR is an infix dyadic (two arguments) operator. Its form is:

type PTR addr-expr

where:

1. type -- BYTE, WORD, DWORD, NEAR, FAR, or structure-name
2. addr-expr -- variable, label, or number

In general, the effect of PTR is to assign the attribute specified on the left to the variable, label, or number specified on the right.

Specifically, PTR assigns the following attributes to "exp" when the expression:

type PTR exp

is encountered:

When "exp" is a:	...then attributes of result are:		
	SEGMENT	OFFSET	TYPE
variable or label	same as "exp" SEG exp	same as "exp" OFFSET exp	type
number	none, undefined	exp itself	type

PTR is quite useful, as the following examples show:

- Increment a byte or word in memory:

```
INC BYTE PTR [BX]
INC WORD PTR [SI]
```

- Move an immediate value to a byte or word in memory:

```
MOV WORD PTR [DI], 99
MOV BYTE PPR [DI], 99
```

- Jump through two levels of indirection:

```
JMP DWORD PTR [BX]    ; BX points to 2-byte offset followed by
                      ; 2-byte segment base.
```

- Pick up a word from a byte array or vice versa:

```
FOOW DW 100 DUP (?)
FOOB DB 200 DUP (?)
        o
        o
        o
        ADD AL, BYTE PTR FOOW[101]    ; Add low-order byte of 50th word to AL.
        ADD DX, WORD PTR FOOB[20]     ; Add word at 21st byte to DX.
```

- With extreme care, treat data as code, as in:

```
JMP NEAR PTR FOO
```

- Create an anonymous variable at a given offset from a segment, as in:

```
MOV AL, DS:BYTE PTR 5   ; Move Byte 5 in DS-addressed segment to AL.
```

and

```
MOV BX, SEG3: WORD PTR 3000   ; Move word at Byte 3000 in SEG3 to BX.
```

Segment Override. Chapter 2 discusses the segment override operator, denoted by the colon (:). This operator takes three forms:

- **seg-reg: addr-expr**
- **segment-name : addr-expr**
- **group-name : addr-expr**

You use the segment override operator to override the SEGMENT attribute of a label, variable, or address-expression. The table that follows shows how all three attributes are affected; the first two forms perform a direct override, while the third (group-name:addr-exp) recalculates the offset from the GROUP base.

Segment Override Form Used	SEGMENT of Result	OFFSET of Result	TYPE of Result
seg-reg:addr-exp	seg-reg	OFFSET(addr-expr) (unchanged)	TYPE(addr-expr) (unchanged)
seg-name:addr-expr	seg-name	OFFSET(addr-expr) (unchanged)	TYPE(addr-expr) (unchanged)
group-name:addr-expr	group-name	adjusted to give offset from GROUP base	TYPE(addr-expr) (unchanged)

The SHORT Operator. The SHORT operator accepts one argument, an offset addressable through the CS segment register. SHORT is used in conditional jumps, jumps, and calls when the target code is within a 1-byte signed (2's complement) self-relative displacement. That is, the target must be no more than 128 bytes behind the beginning of the jump/call instruction, and no more than 127 bytes ahead of it. SHORT saves you a byte; you don't even need to code it when the target definition precedes (lexically, i.e. to the assembler) the jump. (SHORT is also described earlier in this chapter at the end of the section, "JMP and CALL Operands.")

The THIS Operator. THIS accepts one argument, a type (BYTE, WORD, DWORD) or distance (NEAR, FAR) attribute. THIS defines a data item with a specified TYPE at the current location of assembly. The format of THIS is:

```
THIS  type | distance
```

The data item thus defined has the following attributes:

- Segment — the current segment being assembled
- Offset — the current offset in assembly
- Type or distance — as specified

For example, THIS can be inserted:

- To allow flexibility in referencing a byte/word array:

```
FOOB  EQU  THIS BYTE
FOOW  DW   120 DUP (?)
```

  In this example, the EQU directive is equivalent to:

```
FOOB  LABEL  BYTE
```

- To allow flexibility in referencing a label:

```
FAR_OUT   EQU  THIS  FAR
NEAR_IN:  MOV  AX, FOO
            o
            o
            o
```

- Finally, as a point of interest, the location counter symbol '$' is equivalent to THIS NEAR. THIS permits you to type the current value of the location counter.

The HIGH and LOW Operators. These operators are called the byte isolation operators. Each accepts a number or addr-expr as an argument. HIGH returns the high-order byte; LOW the low-order. These operators are included in the assembly language to support 8080-to-8086 conversion, and are not intended for straight 8086 programming.

HIGH and LOW can be applied to themselves; if Q is a relocatable quantity, the following identities hold:

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

Value-Returning Operators

These operators are passive; they return values, but they do not override attributes. They are:

- SEG—when applied to a variable or label, returns the segment value of the variable or label. This operator can be useful in building ASSUME directives, or for initializing segment registers, both of which are described in Chapter 2.
- OFFSET—when applied to a variable or label, returns the offset of the variable or label. This value is resolved at locate-time, when final alignment of the segment is frozen. Since the assembly-time offsets generated on your listing can change if your segment is combined with pieces of the same segment defined in other assembly modules, or is not aligned on a paragraph boundary, the OFFSET operator gives you valuable access to locate-time offsets that might otherwise be in error, were you to calculate them from listings.

OFFSET is useful in accessing variables indirectly, as in:

```
FUM     DW    500 DUP (?)
          o
          o
          o
        MOV   BX, OFFSET FUM
        MOV   SI, 0
AGAIN:    o
          o
          o
        ADD   AX, [BX][SI]      ; Same as [BX + SI]
          o
          o
        ADD   SI, 2
        JMP   AGAIN
```

If you are using the GROUP directive, do not expect the OFFSET operator to yield the offset of a variable within the group, as it will return the offset of the variable within its segment instead. If you need the offset of the variable within the group, use the GROUP override operator instead, as in:

```
DGROUP  GROUP     DATA, ??SEG
 DATA   SEGMENT
           o
           o
           o
  FOO      DB     0
           o
           o
           o
         DW       FOO                      ; Gives offset within segment.
         DW       DGROUP:FOO               ; Gives offset within group.
         DD       FOO                      ; ***INCORRECT***
         DD       DGROUP:FOO               ; ***CORRECT***
 DATA     ENDS
         ASSUME   CS:??SEG, DS:DGROUP
         MOV      BX, OFFSET FOO           ; Loads seg offset of FOO.
         MOV      BX, OFFSET DGROUP:FOO    ; Loads group offset of FOO.
           o
           o
           o
```

The GROUP statement must precede all other uses of the group-name; that is, no forward references to group-names are permitted.

In most assembly languages, the only attribute of a variable is its offset, so that a reference to a variable's name is a reference to its offset. Since this assembly language defines three attributes for a variable, the OFFSET operator is required to isolate the offset value.

However, OFFSET is not required in a DW directive (described in Chapter 2), as for example in:

```
TABLE_PREFIX_BYTE DW TABLE - 1   ;Offset of byte preceding TABLE.
```

An implicit OFFSET is applied to variables in address expressions appearing in DW and DD directives.

- TYPE accepts one argument, which can be either a variable or a label. For variables, TYPE returns 1 for type BYTE, 2 for type WORD, 4 for type DWORD, and N (the number of bytes) in a variable declared with a structure type. For labels, TYPE returns either NEAR or FAR.

TYPE is useful in array calculations, as in:

```
        MOV BX, OFFSET ARRAY
        MOV CX, LENGTH ARRAY    ; LENGTH = # elements.
        MOV SI, 0
        XOR AX,AX               ; AX=0
AGAIN:  ADD AX, [BX+SI]         ; Same as [BX] [SI].
          o
          o
          o
        ADD SI,  TYPE ARRAY
        LOOP AGAIN
```

- LENGTH accepts one argument, a variable, and returns the number of units (not necessarily bytes) allocated for that variable. For example,

```
FEE  DB 150 (?)        ; LENGTH FEE = 150
FUM DW 150 (?)         ; LENGTH FUM = 150
```

- SIZE returns the total number of bytes allocated for a variable, and is related to LENGTH and TYPE by the identity:

  SIZE = LENGTH*TYPE

  Examples of LENGTH and TYPE are given earlier in this chapter under "Structures in Forward/Backward-Linked Lists".

Record-Specific Operators

Records are defined in Chapter 3. The record-specific operators are:

- Shift-count, which is the field-name of th record. Refer to the example below.
- The MASK operator, which accepts a record-field as its only argument and returns a bit-mask defined to be 1's in bit positions included by the field and 0's elsewhere. Refer to the example below.
- The WIDTH operator, which returns the width of a record or a record field as the number of bits in the record or field.

A record is either of type BYTE or WORD, depending on whether its definition formats 8 or 16 bits.

To isolate a record field in a register, two record-specific operators are used:

- the MASK operator, which takes a record field-name as a (right) operand, and yields a (byte or word) bit-pattern consisting of 1's in the record-field bit positions and 0's elsewhere, and
- the record-field name itself, which provides the shift-count needed to right-justify the record-field.

For example, if DITTO is a word in memory with a bit-pattern the same as that defined by the record PATTERN, and PATTERN is defined as:

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

where A through E are record field names (from high-order to low-order) and each field's length is as specified.

To isolate the field in DITTO corresponding to C in PATTERN, write:

```
MOV     DX, DITTO       ; Use any general register but CX.
AND     DX, MASK C      ; Mask out fields A, B, D, E.
MOV     CL, C           ; Must use CL or CX for count (10 here).
SHR     DX, CL          ; Now field C of DITTO is Bits 0:1 of DX.
```

Expressions

Expressions are evaluated left-to-right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. Parentheses can be used to override the normal order of operator precedence, as shown in item 4 below.

Hierarchy (Precedence) of Operators

The classes of operators in order of decreasing precedence are:

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, LOW.

4. Multiplication/division: *, /, MOD, SHL, SHR.

   These are infix operators, for instance:

   ```
   MOV AX, 100 MOD 17 ; AX = 15 = 000FH, since 100 = 5*17 + 15.
   ```

   (MOD accepts only absolute-number operands.)

   ```
   MOV AX, 101B SHL (2*2)   ; AX = 01010000B (shift left 4 bits).
   ```

   Since operators are evaluated left-to-right,

   ```
   101B SHL (2*2) = 01010000B
   ```

   while

   ```
   101B SHL 2*2 = 00101000B
   ```

5. Addition/subtraction (both unary/binary): +, −.

6. Relational: EQ, NE, LT, LE, GT, GE.

   These ("is equal to", "is not equal to", "is less than", "is less than or equal to", "is greater than", and "is greater than or equal to", respectively) operators yield a 16-bit result of all 1's for TRUE (0FFFFH), and all 0's for FALSE (0000H), for instance:

   ```
   MOV AX, 3 EQ 11B   ; AX = 0FFFFH, since 3 = 11B.
   ```

   Given two assembly-time values X and Y, the following defines an array having as many bytes as the lesser value of X and Y:

   ```
   MIN  DB -(X LE Y)*X + -(Y LT X)*Y DUP (0)   ; Length = minimum of X and Y.
   ```

7. Logical NOT.

   NOT forms the 1's complement, e.g.

   ```
   NOT(10101111B)=(01010000B)
   ```

8. Logical AND.

   AND is infix and maps 1's in corresponding positions into 1, and 0's elsewhere in the result, for instance:

   ```
   10110011B AND 11001101B = 10000001B
   ```

9. Logical OR, XOR.

   OR and XOR are both infix. OR maps 0's in corresponding positions into 0, and 1's elsewhere in the result, for instance:

   ```
   11011001B OR 10011011B = 11011011B
   ```

   XOR maps corresponding bits equal in value into 0, and corresponding bits unequal in value into 1, for instance:

   ```
   10111011B XOR 11011101B = 01100110B
   ```

   If A is any assembly-time value, A XOR A = 0.

   A XOR B ≡ (A OR B) AND NOT(A AND B).

10. SHORT is defined in this chapter.

The EQU Directive

You can assign an assembly-time value to a symbol using EQU. The format is:

name EQU expression

where expression can be:

- A symbol, as in:

  ```
  A EQU PARAMETER_12
  ```

  In this special case only (the expression as a symbol), a symbol may be a forward reference.

- An indexing reference, as in:

  ```
  B EQU [BP + 8]
  ```

  You could then code:

  ```
  MOV AX, B.FOO
  ```

  and save a few keystrokes.

- The segment prefix operator ":" and its operands, as in:

  ```
  P8 EQU DS:[BP + 8]
  ```

  This sort of EQU is handy for retrieving items from the data segment with BP as base register, since BP defaults to the SS-addressed segment.

- Instruction names, as in:

  ```
  CBD   EQU   AAD     ; The instruction AAD ASCII adjust for division.
        CBD           ; Converts AX to binary-coded-decimal.
  ```

- Record expressions:

  ```
  BAUDOT  RECORD  A:5, B:5, C:5      ; 5-level triplet code.
    B333  EQU     BAUDOT<3, 3, 3>
    B505  EQU     BAUDOT<5, 0, 5>
          MOV     AX, B505 XOR B333
  ```

- Other assembly-time expressions, such as:

  ```
  E1  EQU  (MASK F1) XOR (0F0H AND MASK F2)
  E2  EQU  E1 MOD 10
  ```

CHAPTER 5
THE INSTRUCTION SET

The descriptors and notation explained below and used in this chapter are not all valid in source statements. They are used here as a shorthand, and explained in the English text that accompanies each instruction description. The code examples, however, are valid source statements.

Table 5-1. Symbols

MCS-86 Descriptor	Meaning
AX	Accumulator (16-bit) (8080 Accumulator holds only 8-bits)
AH	Accumulator (high-order byte)
AL	Accumulator (low-order byte)
BX	Register BX (16-bit) (8080 register pair HL), 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) (8080 register pair BC), 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) (8080) register DE) 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 (8080 Program Counter) (16-bit)
Flags	16-bit register space, in which nine flags reside. (Not directly equivalent to 8080 PSW, which contains five flags and the contents of the accumulator.)
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)
REG8	The name or encoding of an 8-bit CPU register location.
REG16	The name or encoding of a 16-bit CPU register location.
LSRC, RSRC	Refer to operands of an instruction, generally left source and right source when two operands are used. The leftmost operand is also called the destination operand, and the rightmost is called the source operand.
reg	A field which specifies REG8 or REG16 in the description of an instruction.
EA	Effective address (16-bit)

Table 5-1. Symbols (Cont'd.)

MCS-86 Descriptor	Meaning
r/m	Bits 2, 1, 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, 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 which 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

Instruction and Data Formats

The formats described briefly here reflect the assembly language processed by the Intel-supplied assembler, ASM-86, used with the Intellec development systems.

Assembly language instructions are written one per line. If a semicolon occurs other than in a string, then the remainder of that line is taken as a comment. If a line begins with an ampersand ("&"), it is considered a continuation of the previous line (instruction or directive, not comment).

Any instruction is made up of a series of tokens. Each token may be one of three types:

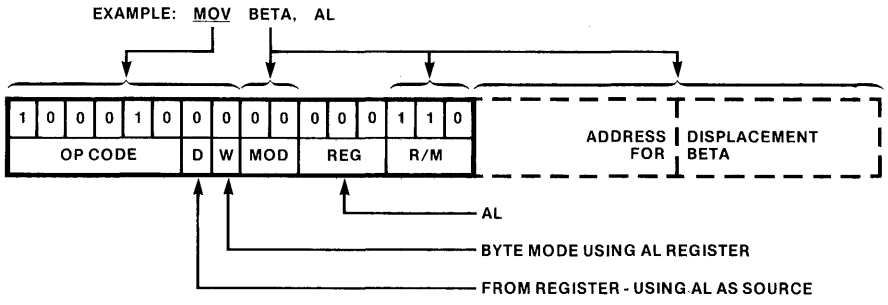
Name
Constant
Delimiter

If two consecutive tokens together might be interpreted as some other token, they must be separated by a space; if not, spaces have no meaning and may be omitted. However, extra spaces may be inserted if desired; the computer ignores them. Comments may be made any number of lines long, but a semicolon must start each line of a comment. The assembler ignores comments and blank lines. It does not distinguish between capitals and lower-case letters.

An exception to the above rules is the character string. The assembler recognizes all of the characters, spaces, and blanks that are contained within the string.

Instruction Set Encyclopedia

Page 5-157 is an alphabetical index to this chapter. In these lists, all instructions are referenced to the assembly-language mnemonics. Although there is not a unique mnemonic for each instruction code, there is enough information in the instruction, source, and destination mnemonics for the assembler to identify the correct code. This means, for example, that you don't have to keep in mind which of the different MOV codes is needed for different source and destination operands. When you write



The assembler takes care of:

D	(direction bit)
W	(word bit)
MOD	(mode field)
Displacement	

The assembler chooses the correct mode to perform your intended operation. This chapter describes how those codes function.

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 1).

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.

The mode field occupies the two most significant bits 7, 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 which holds the operand or can specify an addressing mode (in combination with the mod field) which points to the location of the operand in memory. The reg field occupies bits 5, 4, 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 (see also Chapter 6 and Appendix A).

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 which 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 having a high-order byte which is the sign extension of the low-order byte may be abbreviated to eight bits.

Three points about immediate operands should be made:

- 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.

Below each type of instruction, the following information is given:

1. A descriptive English name or phrase
2. The instruction's binary encoding
3. The time it takes, expressed in clock cycles (using a 5-MHz clock, one cycle is 200 nanoseconds; using an 8-MHz clock, one cycle is 125 nanoseconds)
4. A step-by-step operational description
5. A list of flags set to 1 or reset to 0 during the operation of this instruction (see also Appendix C).
6. A general description of when the instruction is used, how it works, defaults it may use or invoke, and points to remember about its interaction with other instructions or directives.

7. Examples

The times given for instructions depend on the nature of the operands. These times are fixed for register-to-register operations and for immediate-data-to-register operations, e.g.

```
MOV  DX,  AX   takes 2 cycles
MOV  DX,  444  takes 4 cycles, regardless of which register or what data.
```

Operands in memory take some extra time for calculating the Effective Address. These added cycles are indicated in the listed times by the term "+EA". The amount of time needed varies depending on 3 factors:

a. Which addressing mode was used in the address expression for the memory operand.

b. Whether a segment override prefix byte is needed.

c. For word operands, whether the first byte of the word resides at an even or odd address.

The list below shows the added cycles needed for each addressing mode to access either 8-bit memory operands or 16-bit memory operands (words) whose first byte is at an even address. Add 4 cycles for words residing at odd memory addresses. Add 2 cycles if a segment override is used.

	Addressing Mode	Add
1.	direct 16-bit offset address e.g., MOV BX, SIMPLE_NAME takes 8 + 6 or 14 cycles	6
2.	indirect through base or index register e.g., MOV CX, [BX] MOV CX, [SI] each takes 8 + 5 or 13 cycles	5
3.	indirect through base or index register with displacement constant e.g., MOV DX, SIMPLE_NAME [BX] MOV SIMPLE_NAME [DI] , CX each takes 8 + 9 or 17 cycles	9
4.	indirect through sum of one base and one index register e.g., MOV DX, [BX] [SI] MOV [BX] [DI] , CX each takes 8 + 7 or 15 cycles	7 or 8
5.	indirect through sum of base and index register plus displacement constant e.g., MOV DX, SIMPLE_NAME [BX] [SI] MOV SIMPLE_NAME [BX] [DI] ,CX each takes 8 + 11 or 19 cycles	11 or 12

If SIMPLE_NAME resides at an odd address, each of the above address expressions involving that variable would require 4 extra cycles. If a segment override were necessary (see ASSUME in Chapter 4), then an additional 2 cycles must be added. Thus the instruction MOV ES:SIMPLE_NAME, CX would require 16 instead of 14 cycles, and 20 cycles if the first byte of SIMPLE_NAME were at an odd address.

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:

Register or memory space to or from register
Immediate data to register or memory
Accumulator to or from registers, memory, or ports

Data Transfer

Data transfer operations are divided into four classes:

- general purpose
- accumulator-specific
- address-object
- 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 which 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 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; otherewise 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. Three division operations are 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 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 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.

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, which 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 which 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.

You should try to avoid using the two other prefix bytes with a repeat-prefixed string instruction, i.e., a segment prefix or the LOCK prefix. 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.

Primitive String Operations: Five primitive string operations are provided:

— MOVB (or MOVW) 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.

— CMPB (or CMPW) 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.

— SCAB (or SCAW) 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.

— LODB (or LODW) transfers a byte (or word) operand from the source operand to AL (or AX). This operation ordinarily would not be repeated.

— STOB (or STOW) 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.

The operand-less forms of the string instructions (MOVSB, MOVSW, etc.) are described under MOVS, etc., and in Chapter 2.

In all 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 a 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 which transfer control within the current code segment, and those which 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 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 5-2 shows the available instructions, the conditions associated with them, and their interpretation.

Table 5-2. 8086 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	((SP 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 intersegment 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.

There are 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).

— 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 which 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 reentered. 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.

AAA (ASCII adjust for addition)

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

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

Encoding:

0 0 1 1 0 1 1 1

Timing: 4 clocks

Example: AAA ;after the addition

Flags Affected: AF, CF.
Undefined: OF, PF, SF, ZF

Description: 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.

AAD (ASCII adjust for division)

Operation: The high byte (AH) of the accumulator is multiplied by 10 and added to the low byte (AL). The result is stored into AL. AH is zeroed out.

```
(AL) ← (AH)*OAH + (AL)
(AH) ← 0
```

Encoding:

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

Timing: 60 clocks

Example: AAD ;prior to the division

Flags Affected: PF, SF, ZF.
Undefined: AF, CF, OF

Description: AAD (Unpacked BCD (ASCII) adjust for division) performs an adjustment of the dividend in AL before a subsequent instruction divides two unpacked decimal operands, so that the result of the division will be an unpacked decimal quotient.

AAM (Ascii adjust for multiply)

Operation: The contents of AH are replaced by the result of dividing AL by 10. Then the contents of AL are replaced by the remainder of that division, i.e. by AL modulo 10.

```
(AH) ← (AL) / 0AH
(AL) ← (AL) % 0AH
```

Encoding:

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

Timing: 83 clocks

Example: AAM ;after the multiply

Flags Affected: PF, SF, ZF.
Undefined: AF, CF, OF

Description: 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.

AAS (ASCII adjust for subtraction)

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

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

Encoding:

0 0 1 1 1 1 1 1

Timing: 4 clocks

Example: AAS ;after the subtraction

Flags Affected: AF, CF.
Undefined: OF, PF, SF, ZF

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

ADC (Add with carry)

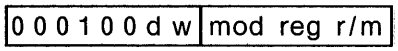
Operation: If the carry flag was set, ADC adds 1 to the sum of the two operands before storing the result into the destination (leftmost) operand. If the carry flag was not set, i.e. is zero, 1 is not added.

if (CF) = 1 then (DEST) ← (LSRC) + (RSRC) + 1
else (DEST) ← (LSRC) + (RSRC)

See note.

Encoding:

Memory or Register Operand with Register Operand:



if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks):

(a) register to register	3
(b) memory to register	9+EA
(c) register to memory	16+EA

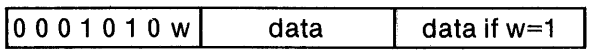
Examples:

```
(a) ADC   AX,   SI
    ADC   ,SI   ;same as above
    ADC   DI,   BX
    ADC   CH,   BL

(b) ADC   DX,   MEM_WORD
    ADC   AX,   BETA [SI]
    ADC   ,BETA [SI]   ;same as above
    ADC   CX,   ALPHA [BX] [SI]

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

Immediate Operand to Accumulator:



if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

Timing: 4 clocks

ADC

Examples:

```
ADC  AL,  3
ADC  AL,  VALUE_13_IMM
ADC  AX,  333
ADC  AX, IMM_VAL_777
ADC  ,IMM_VAL_777   ;same as above
```

Immediate Operand to Memory or Register Operand:

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate to memory 17+EA
(b) immediate to register 4

Examples:

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

(b) ADC  BX,  IMM_VAL_987
    ADC  DH,  65
    ADC  CX,  432
```

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

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: ADC (add with carry) performs an addition of the two operands, adds one if the CF flag is set, and returns the result to the destination (leftmost) operands.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

ADD (Addition)

Operation: The sum of the two operands is stored into the destination (leftmost) operand.

(DEST) ← (LSRC) + (RSRC)

See note.

Encoding:

Memory or Register Operand with Register Operand:

0 0 0 0 0 0 d w	mod reg r/m

if d = 0 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks):

(a)	register to register	3
(b)	memory to register	9+EA
(c)	register to memory	16+EA

Examples:

```
(a) ADD  AX,  BX
    ADD  ,BX  ;same as above
    ADD  CX,  DX
    ADD  DI,  SI
    ADD  BX,  BP

(b) ADD  CX,  MEM_WORD
    ADD  AX,  BETA [SI]
    ADD  ,BETA [SI]   ;same as above
    ADD  DX,  ALPHA [BX] [DI]

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

Immediate Operand to Accumulator:

0 0 0 0 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

Timing: 4 clocks

ADD

Examples:

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

Immediate Operand to Memory or Register Operand:

1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate to memory 17+EA
(b) immediate to register 4

Examples:

```
(a) ADD  MEM_WORD,  48
    ADD  GAMMA [DI],  IMM_84
    ADD  DELTA [BX] [SI],  IMM_SENSOR_5

(b) ADD  BX,  ORIG_VAL
    ADD  CX,  STANDARD_COUNT
    ADD  DX,  1776
```

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

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: ADD performs an addition of the two source operands and returns the result to the destination operands.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

AND (And: logical conjunction)

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

(DEST) ← (LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

See note.

Encoding:

Memory or Register Operand with Register Operand:

0 0 1 0 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks):

(a) register to register	3
(b) memory to register	9+EA
(c) register to memory	16+EA

Examples:

```
(a) AND  AX,  BX
    AND  ,BX  ;same as above
    AND  CX,  DI
    AND  BH,  CL

(b) AND  SI,  MEM_NAME_WORD
    AND  DX,  BETA [BX]
    AND  BX,  GAMMA [BX] [SI]
    AND  AX,  ALPHA [DI]
    AND  ,ALPHA [DI]  ;same as above
    AND  DH,  MEM_BYTE

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

Immediate Operand to Accumulator:

0 0 1 0 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

Timing (clocks): immediate to register 4

AND

Examples:

```
AND  AL,  7AH
AND  AH,  0EH
AND  AX,  IMM_VAL_MASK 3
```

Immediate Operand to Memory or Register Operand:

1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate to register 4
(b) immediate to memory 17+EA

Examples:

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

(b) AND  MEM_WORD,  7A46H
    AND  MEM_BYTE,  46H
    AND  GAMMA [DI],  IMM_MASK 14
    AND  CHI_BYTE [BX] [SI],  11100111B
```

Flags Affected: CF, OF, PF, SF, ZF.
Undefined: AF

Description: AND performs the bitwise logical conjunction of the two source operands and returns the result to one of the operands.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

CALL (Call a procedure)

Operation: If this is an intersegment call, the stack pointer is decremented by 2 and the contents of the CS register are pushed onto the stack. CS is then filled by the second word (segment) of the doubleword intersegment pointer.

Then the stack pointer is decremented by 2 and the contents of the Instruction Pointer are pushed onto the stack. The last step is to replace the contents of the IP by the offset of the target destination, i.e. the offset of the procedure's first instruction. An intra-segment or intra-group call does only steps 2, 3, and 4.

```
1) if Inter-Segment then
     (SP) ← (SP)−2
     ((SP)+1:(SP)) ← (CS)
     (CS) ← SEG
2) (SP) ← (SP)−2
3) ((SP)+1:(SP)) ← (IP)
4) (IP) ← DEST
```

See note.

Encoding:

Direct Intra-segment or Intra-group:

1 1 1 0 1 0 0 0	disp-low	disp-high

DEST = (IP) + disp

Timing: 13 + EA clocks

Examples:

```
CALL  NEAR_LABEL
CALL  NEAR_PROC
```

Inter-Segment Direct:

1 0 0 1 1 0 1 0	offset-low	offset-high	seg-low	seg-high

DEST = offset, SEG = seg

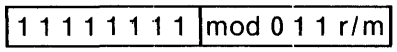
Timing: 20 clocks

Examples:

```
CALL  FAR_LABEL
CALL  FAR_PROC
```

CALL

Inter-Segment Indirect:



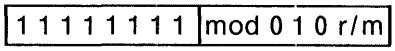
DEST = (EA), SEG = (EA + 2)

Timing: 29 + EA clocks

Examples:

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

Indirect Intra-Segment or Intra-Group



DEST = (EA)

Timing: 11 clocks

Examples:

```
CALL  WORD  PTR  [BX]
CALL  WORD  PTR  VARIABLE_NAME
CALL  WORD  PTR  [BX] [SI]
CALL  WORD  PTR  [DI]
CALL  WORD  PTR  VARIABLE_NAME [BP] [SI]
CALL  MEM_WORD
CALL  BX
CALL  CX
```

Flags Affected: None

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

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

As shown in the indirect-call examples above, calls through variables may use the PTR operator to indicate the intended use of one word for NEAR calls, or two words for calls to FAR labels or procedures. Indirect calls using word registers (within squarebrackets) are of necessity NEAR calls.

The implicit segment register used in a register-indirect call is DS, unless BP is used or an override is specified. The implicit segment register is used to construct the address which contains the offset (and segment, if a "long" call) of the call's target. If BP is used, SS is the segment register used. However, if a segment prefix byte is explicitly specified, e.g.,

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

then the segment register so specified is used (here ES). An implicit segment register for indirect calls through variables or address-expressions is determined by the address-expression in the source line and the applicable ASSUME directive (see Chapter 4).

When CALL is used to transfer control, a RETurn is implied. With indirect CALLS, you must carefully ensure that the type of the CALL matches the type of RETurn, or errors may result that are difficult to trace. The issue is whether CS is saved and restored. See RET and Appendix D.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

CBW

CBW (Convert byte to word)

Operation: If the lower byte of the accumulator (AL) is less than 80H, then AH is made zero. Otherwise, AH is set to FFH. This is equivalent to replicating bit 7 of AL all through AH.

Encoding:

1 0 0 1 1 0 0 0

Timing: 2 clocks

Example: CBW

Flags Affected: None

Description: CBW (convert byte to word) performs a sign extension of the AL register into the AH register.

CLC (Clear carry flag)

Operation: The carry flag is reset to zero.

(CF) ← 0

Encoding:

1 1 1 1 1 0 0 0

Timing: 2 clocks

Example: CLC

Flags Affected: CF

Description: CLC clears the CF flag.

CLD

CLD (Clear direction flag)

Operation: The direction flag is reset to zero.

(DF) ← 0

Encoding:

1 1 1 1 1 1 0 0

Timing: 2 clocks

Example: CLD

Flags Affected: DF.

Description: CLD clears the DF flag, causing the string operations to auto- increment the operand pointers.

CLI (Clear interrupt flag)

Operation: The interrupt flag is reset to zero.

(IF) ← 0

Encoding:

1 1 1 1 1 0 1 0

Timing: 2 clocks

Example: CLI

Flags Affected: IF

Description: CLI clears the IF flag, disabling maskable external interrupts, which appear on the INTR line of the 8086. (Nonmaskable interrupts, which appear on the NMI line are not disabled.)

CMC (Complement carry flag)

Operation: If the carry flag is zero, it is set to 1; if it is 1, it is reset to 0.

if (CF) = 0 then (CF) ← 1 else (CF) ← 0

Encoding:

1 1 1 1 0 1 0 1

Timing: 2 clocks

Example: CMC

Flags Affected: CF

Description: CMC complements the CF flag.

CMP(Compare two operands)

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

(LSRC)−(RSRC)

See note.

Encoding:

Memory or Register Operand with Register Operand:

0 0 1 1 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA
else LSRC = EA, RSRC = REG

Timing (clocks):

(a)	register with register	3
(b)	memory with register	9+EA
(c)	register with memory	9+EA

Examples:

```
(a) CMP  AX,  DX
    CMP  ,DX  ;same as above
    CMP  SI,  BP
    CMP  BH,  CL

(b) CMP  MEM_WORD,  SI
    CMP  MEM_BYTE,  CH
    CMP  ALPHA [DI],  DX
    CMP  BETA [BX] [SI],  CX

(c) CMP  DI,  MEM_WORD
    CMP  CH,  MEM_BYTE
    CMP  AX,  GAMMA [BP] [SI]
```

Immediate Operand with Accumulator:

0 0 1 1 1 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data
else LSRC = AX, RSRC = data

Timing (clocks):

	immediate with register	4

CMP

Examples:

```
CMP  AL,  6
CMP  AL,  IMM_VALUE_DRIVE 11
CMP  AX,  IMM_VAL_909
CMP  ,999
CMP  AX,  999  ;same as above
```

Immediate Operand with Memory or Register Operand:

1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data

Timing (clock): (a) immediate with register 4
(b) immediate with memory 17 + EA

Examples:

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

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

If an immediate-data-byte is being compared from a register-or-memory word, then that byte is sign-extended to 16 bits prior to the compare. For this situation the instruction byte is 83H (i.e., the s:w bits are both set).

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: CMP (compare) performs a subtraction of the two operands causing the flags to be affected but does not return the result.

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

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

CMPS (Compare byte string, compare word string)

Chapter 2 describes CMPSB and CMPSW.

Operation: The rightmost operand, using DI as an index into the extra segment, is subtracted from the leftmost operand, which uses SI as an index. (This is the only string instruction in which the DI-indexed operand appears as the rightmost operand.) Only the flags are affected, not the operands. SI and DI are then incremented, if the direction flag is reset (zero), or they are decremented, if DF=1. They thus point to the next element of the strings being compared. The increment is 1 for byte strings, 2 for word strings.

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

Encoding:

1 0 1 0 0 1 1 w

if w = 0 then LSRC = (SI), RSRC = (DI), DELTA = 1 (BYTE)
else LSRC = (SI) + 1:(SI), RSRC = (DI) + 1:(DI), DELTA = 2 (WORD)

Timing: 22 clocks

Example:

```
MOV  SI,  OFFSET  STRING1
MOV  DI,  OFFSET  STRING2
CMPS  STRING 1,  STRING2
;the operands named in the CMPS instruction are used only
;by the assembler to verify type and accessibility using current seg-
;ment register contents. CMPS actually uses only SI and DI to point to
;the locations whose contents are to be compared, without using the
;names given in the source CMPS line.
```

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: CMPS subtracts the byte (or word) operand addressed by DI from the operand addressed by SI 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.

Note that the operand indexed by DI is the rightmost operand in this instruction, and that this operand is addressed using the ES register only—this default CANNOT be overridden.

CWD

CWD (Convert word to doubleword)

Operation: The high order bit of AX is replicated throughout DX.

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

Encoding:

1 0 0 1 1 0 0 1

Timing: 5 clocks

Example: CWD

Flags Affected: None

Description: CWD (convert word to double word) performs a sign extension of the AX register into the DX register. See also DIV.

DAA (Decimal adjust for addition)

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

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

Encoding:

0 0 1 0 0 1 1 1

Timing: 4 clocks

Example: DAA

Flags Affected: AF, CF, PF, SF, ZF
Undefined: OF

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

DAS

DAS (Decimal adjust for subtraction)

Operation: If the lower nibble (4 bits) of AL is greater than 9 or if the auxiliary flag has been set, then 6 is subtracted from AL and AF is set. If AL is greater than 9FH or if the carry flag has been set, then 60H is subtracted from AL and CF is set.

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

Encoding:

0 0 1 0 1 1 1 1

Timing: 4 clocks

Example: DAS

Flags Affected: AF, CF, PF, SF, ZF.
Undefined: OF

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

DEC (Decrement destination by one)

Operation: The specified operand is reduced by 1.

(DEST) ← (DEST)−1

See note.

Encoding:

Register Operand: (Word)

0 1 0 0 1 reg

DEST = REG

Timing: 2 clocks

Examples:

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

Memory or Register Operand:

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

DEST = EA

Timing (clocks):

register	2
memory	15 + EA

Examples:

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

Flags Affected: AF, OF, PF, SF, ZF

Description: DEC (decrement) performs a subtraction of one from the operand and returns the result to that operand.

DEC

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

DIV (Division, unsigned)

Operation: If the division results in a value larger than can be held by the appropriate registers, an interrupt of type 0 is generated. The flags are pushed onto the stack, IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack and IP is then filled with the word at 0. This sequence thus includes a long call to the interrupt handling procedure whose segment and offset are stored respectively at locations 2 and 0.

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

```
(temp) ← (NUMR)
if (temp) / (DIVR) > MAX then the following, in sequence
        (QUO), (REM) undefined
        (SP) ← (SP)−2
        ((SP)+1:(SP)) ← FLAGS
        (IF) ← 0
        (TF) ← 0
        (SP) ← (SP)−2
        ((SP)+1:(SP)) ← (CS)
        (CS) ← (2)   i.e., the contents of memory locations 2 and 3
        (SP) ← (SP)−2
        ((SP)+1:(SP)) ← (IP)
        (IP) ←   (0) i.e., the contents of locations 0 and 1
else
        (QUO) ← (temp) / (DIVR), where / is unsigned division
        (REM) ← (temp) % (DIVR), where % is unsigned modulo
```

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

(a) if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM= AH,
MAX = FFH
(b) else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX,
MAX = FFFFH

Timing: (clocks): 8-bit 90 + EA
16-bit 155 + EA

DIV

Examples:

(a1) to divide a word by a byte

```
MOV  AX,  NUMERATOR_WORD
DIV  DIVISOR_BYTE
;quotient will be in AL, remainder in AH
```

(a2) to divide a byte by a byte

```
MOV  AL,  NUMERATOR_BYTE
CBW     ;converts byte in AL to word in AX
DIV  DIVISOR_BYTE
;quotient in AL, remainder in AH
```

(b1) to divide a double word by a word

```
MOV  DX,  NUMERATOR_HI_WORD
MOV  AX,  NUMERATOR_LO_WORD
DIV  DIVISOR_WORD
;quotient in AX remainder in DX
```

(b2) to divide a word by a word

```
MOV  AX,  NUMERATOR_WORD
CWD     ;converts word to doubleword
DIV  DIVISOR_WORD
;quotient in AX, remainder in DX
```

NOTE: Each memory operand above could be any variable or valid address- expression so long as its type were the same. For example, in (a1) above, NUMERATOR_WORD could be replaced by the expression

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

so long as ARRAY_NAME is of type WORD. Similarly DIVISOR_BYTE could be

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

so long as RATE_TABLE is of type BYTE.

Flags Affected: no valid flags result
Undefined: AF, CF, OF, PF, SF, ZF

Description: DIV (divide) performs an unsigned division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified source operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation). If the quotient is greater than MAX (as when division by zero is attempted) then QUO and REM are undefined, and a type 0 interrupt is generated. Flags are undefined in any DIV operation. Nonintegral quotients are truncated to integers.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

ESC (Escape)

Operation:

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

See note.

Encoding:

1 1 0 1 1 x	mod x r/m

Timing: 7 + EA clocks

Example:

```
ESC  EXTERNAL_OPCODE,  ADDRESS
; this opcode is a 6-bit number, which is split into the two 3-bit fields
; shown as x above.
```

Flags Affected: None

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.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

HLT (Halt)

Operation: None

Encoding:

1 1 1 1 0 1 0 0

Timing: 2 clocks

Example: HLT

Flags Affected: None

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

IDIV (Integer division, signed)

Operation: If the division results in a value larger than can be held by the appropriate registers, an interrupt of type 0 is generated. The flags are pushed onto the stack, IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack and IP is then filled with the word at 0. This sequence thus includes a long call to the interrupt handling procedure whose segment and offset are stored respectively at locations 2 and 0.

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

```
(temp) ← (NUMR)
if (temp) / (DIVR) > 0 and (temp) / (DIVR) > MAX
or (temp) / (DIVR) < 0 and (temp) / (DIVR) < 0–MAX–1
        then
        (QUO), (REM) undefined
        (SP) ← (SP)–2
        ((SP)+1:(SP)) ← FLAGS
        (IF) ← 0
        (TF) ← 0
        (SP) ← (SP)–2
        ((SP)+1:(SP)) ← (CS)
        (CS) ← (2)
        (SP) ← (SP)–2
        ((SP) + 1:(SP)) ← (IP)
        (IP) ← (0)
else
        (QUO) ← (temp) / (DIVR), where / is signed division
        (REM) ← (temp) % (DIVR), where % is signed modulo
```

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

(a) if w = 0 then NUMR = AX, DIVR = EA, QUO = AL, REM = AH, MAX = 7FH
(b) else NUMR = DX:AX, DIVR = EA, QUO = AX, REM = DX, MAX = 7FFFH

Timing (clocks): 8-bit 112+EA
16-bit 177+EA

Example:

```
(a) MOV   AX,   NUMERATOR_WORD [BX]
    IDIV  DIVISOR_BYTE [BX]

(b) MOV   DX,   NUM_HI_WORD
    MOV   AX,   NUM_LO_WORD
    IDIV  DIVISOR_WORD [SI]
```

SEE ALSO DIV.

Flags Affected: AF, CF, OF, PF, SF, ZF
Undefined: All

Description: IDIV (integer divide) performs a signed division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified source operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation). If the quotient is positive and greater than MAX or if the quotient is negative and less than (0−MAX−1), (as when division by zero is attempted) then QUO and REM are undefined, and a type 0 interrupt is generated. Flags are undefined in any divide operation. IDIV truncates nonintegral quotients and returns a remainder with the same sign as the numerator.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

IMUL
(Integer multiply accumulator by register-or-memory; signed)

Operation: The accumulator (AL if byte, AX if word) is multiplied by the specified operand. If the high-order half of the result is the sign-extension of the low-order half, the carry and overflow flags are reset, otherwise they are set.

```
(DEST) ← (LSRC) * (RSRC) where * is signed multiply
if (EXT) = sign-extension of (LOW) then (CF) ← 0
else (CF) ← 1;
(OF) ← (CF)
```

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

(a) if w = 0 then LSRC = AL, RSRC = EA, DEST = AX, EXT = AH, LOW =AL
(b) else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX, LOW = AX

Timing (clocks): 8-bit 90+EA
16-bit 144+EA

Example:

```
(a)  MOV  AL,  LSRC_BYTE
     IMUL  RSRC_BYTE  ;result in AX

(b1) MOV  AX,  LSRC_WORD
     IMUL  RSRC_WORD
     ;high-half result in DX, low-half in AX

(b2) to multiply a byte by a word
     MOV  AL,  MUL_BYTE
     CBW  ;converts byte in AL to word in AX
     IMUL  RSRC_WORD
     ;high-half result in DX, low-half in AX
```

NOTE: Any memory operand above could be an indexed address-expression of the correct TYPE, e.g., LSRC_BYTE could be ARRAY [SI] if ARRAY were of type BYTE, and RSRC_WORD could be TABLE [BX] [DI] if TABLE were of type WORD.

Flags Affected: CF, OF.
Undefined: AF, PF, SF, ZF

Description: IMUL (integer multiply) performs a signed 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, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

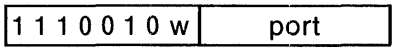
IN (Input byte and input word)

Operation: The contents of the accumulator are replaced by the contents of the designated port.

(DEST)←(SRC)

Encoding:

Fixed Port:



if w = 0 then SRC = port, DEST = AL
else SRC = port + 1:port, DEST = AX

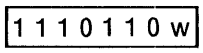
Timing: 10 clocks

Examples:

```
IN   AX, WORD_PORT   ;input word to AX
IN   AL, BYTE_PORT   ;input a byte to AL

;the destination for input must be AX or AL, and must be specified in
;order for the assembler to know the type of the input The port names
;must be immediate values between 0 and 255, as used above or literally
;the register name DX, which must be filled earlier with the requisite
;port location
```

Variable Port:



if w = 0 then SRC = (DX), DEST = AL
else SRC = (DX) + 1:(DX), DEST = AX

Timing: 8 clocks

Examples:

```
IN   AX, DX   ;input a word to AX
IN   AL, DX   ;input a byte to AL
```

Flags Affected: None

Description: 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.

INC (Increment destination by 1)

Operation: The specified operand is incremented by 1. There is no carry out of the most-significant bit.

(DEST) ← (DEST) + 1

See note.

Encoding:

Register Operand: (Word)

0 1 0 0 0 reg

DEST = REG

Timing: 2 clocks

Examples:

```
INC  AX
INC  DI
```

Memory or Register Operand:

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

DEST = EA

Timing (clocks): (a) register 2
(b) memory 15 + EA

Examples:

```
(a) INC  CX
    INC  BL

(b) INC  MEM_BYTE
    INC  MEM_WORD [BX]
    INC  BYTE  PTR [Bx]      ;byte in DATA Segment at offset [BX]
    INC  ALPHA [DI] [BX]
    INC  BYTE  PTR  [SI] [BP]   ;byte in Stack Segment at offset [SI + BP]
    INC  WORD PTR [BX]       ;increments the word in Data Segment at
    offset [BX], and thus can get carry into bit 8.
```

Flags Affected: AF, OF, PF, SF, ZF

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

INC

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

INT (Interrupt)

Operation: Stack Pointer is decremented by 2 and all flags are pushed into the stack. The interrupt and trap flags are then reset. SP is then decremented by 2 and the current contents of the CS register are pushed onto the stack. CS is then filled with the high-order word of the doubleword interrupt vector, i.e., the segment base-address of the interrupt handling procedure for this interrupt type.

SP is then decremented by 2 and the current contents of the Instruction Pointer are pushed onto the stack. IP is then filled with the low-order word of the interrupt vector, located at absolute address TYPE*4. This completes an intersegment ("long") call to the procedure which is to process this interrupt type.

See also PUSHF, INTO, IRET.

```
(SP) ← (SP) – 2
((SP) + 1:(SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) – 2
((SP) + 1:(SP)) ← (CS)
(CS) ← (TYPE * 4 + 2)
(SP) ← (SP) – 2
((SP) + 1:(SP)) ← (IP)
(IP) ← (TYPE * 4)
```

Encoding:

1 1 0 0 1 1 0 v	type if v=1

(a) if v = 0 then TYPE = 3
(b) else TYPE = type

Timing: 52 clocks

Examples:

```
(a) INT      3   ;one byte instruction, 11001100

(b) INT      2   ;two bytes: 11001101 00000010
    INT      67  ;two bytes: 11001101 01000011
    IMM_44   EQU 44
    INT      IMM_44  ;two bytes: 11001101 00101100
```

Note: The operand must be immediate data, not a register or a memory reference.

Flags Affected: IF, TF

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 (Interrupt if overflow)

Operation: If the overflow flag is zero, no operation occurs. If OF is 1, then Stack Pointer is decremented by 2 and all flags are saved onto the stack. The trap and interrupt flags are reset. SP is again decremented by 2 and the contents of CS are pushed into the stack. CS is then filled with the second word (segment) of the doubleword interrupt vector for a type 4 interrupt.

SP is again decremented by 2, and the current Instruction Pointer (pointing to the next instruction after INTO) is pushed onto the stack. IP is then filled with the first word of the type 4 doubleword interrupt vector, located at absolute location 16 (10H). This word is the offset of the procedure to handle type 4 interrupts. The segment base address was already placed in CS. Thus this completes a "long" call to the proper procedure.

See also INT, IRET, PUSHF.

```
if (OF) = 1 then
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← FLAGS
  (IF) ← 0
  (TF) ← 0
  (SP) ← (SP) – 2
  ((SP)) + 1:(SP)) ← (CS)
  (CS) ← (12H)
  (SP) ← (SP) – 2
  ((SP) + 1:(SP)) ← (IP)
  (IP) ← (10H)
```

Encoding:

1 1 0 0 1 1 1 0

Timing: 52 clocks

Example: INTO

Flags Affected: None

Description: 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 1OH) if the OF flag is set (trap on overflow). If the OF flag is clear, no operation takes place.

IRET (Interrupt return)

Operation: The instruction Pointer is filled with the word at the top of the stack. The Stack Pointer is then incremented by 2, and the CS register is filled with the word now at the top of the stack. This returns control to the point where the interrupt was encountered.

SP is again incremented by 2, and the flags are restored from the appropriate bits of the word now at the top of the stack. (See also POPF.) SP is again incremented by 2.

```
(IP) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
(CS) ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
FLAGS ← ((SP) + 1:(SP))
(SP) ← (SP) + 2
```

Encoding:

1 1 0 0 1 1 1 1

Timing: 24 clocks

Example: IRET

Flags Affected: All

Description: IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

JNBE and JA (Jump if not below nor equal, or jump if above)

Operation: If both the carry flag and the zero flag are zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a transfer. If (CF) = 1 or (ZF) = 1, no jump results.

```
IF (CF)|(ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JA   TARGET_LABEL
JNBE   TARGET_LABEL
```

Flags Affected: None

Description: JNBE (or JA) transfers control to the target operand on not below or equal (or above).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNB and JAE (Jump if not below, or jump if above or equal)

Operation: If the carry flag is zero, the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the transfer. If (CF) = 1, no jump results.

```
if (CF) = 0 then
(IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 0 1 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JNB  TARGET_LABEL
JAE  TARGET_LABEL
```

Flags Affected: None

Description: JNB (or JAE) transfers control to the target operand on not below (or above or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JB/JC

JB and JNAE (Jump if below, or jump if not above nor equal)
JC (Jump if carry)

Operation: If the carry flag is 1, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (CF) = 0, no jump occurs.

```
if (CF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 0 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JB   TARGET_LABEL
JNAE TARGET_LABEL
JC   TARGET_LABEL
```

Flags Affected: None

Description: JB (or JNAE) transfers control to the target operand on below (or not above or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

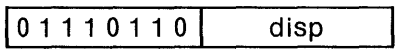
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JBE and JNA (Jump if below or equal, or jump if not above)

Operation: If either the carry flag or the zero flag is set, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If both (CF) = 0 and (ZF) = 0, no jump occurs.

```
if (CF)|(ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
JBE  TARGET_LABEL
JNA  TARGET_LABEL
```

Flags Affected: None

Description: JBE (or JNA) transfers control to the target operand on below or equal (or not above).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

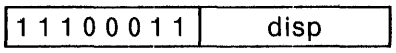
JCXZ

JCXZ (Jump if CX is zero)

Operation: If the count register (CX) is zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a transfer. If (CX) = 1, no jump occurs.

```
if (CX) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	9
Jump is not taken	5

Example: JCXZ TARGET_LABEL

Flags Affected: None

Description: JCXZ (jump on CX zero) transfers control to the target operand if the CX register is zero.

NOTE: The target label must be within −128 to +127 bytes of this instruction.

JE and JZ (Jump if equal, jump if zero)

Operation: If the last operation to affect the zero flag gave a result of zero, then (ZF) will be 1. If (ZF) = 1, then the distance from the end of this instruction to the target label will be added to the Instruction Pointer, effecting a transfer of control to that label. If (ZF) = 0, no operation occurs.

```
if (ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 0 0	disp

Timing (clocks: Jump is taken 8
Jump is not taken 4

Examples:

```
1)    CMP  CX,  DX
      JE   LAB2
      INC  CX
LAB2:
;the increment of CX will only occur if CX ≠ DX

2)    SUB  AX,  BX
      JZ   EXACT
      ;jump occurs if result was zero, i.e., AX = BX
      .
      .
      .
EXACT:
```

Flags Affected: None

Description: JE (or JZ) transfers control to the target operand on equal (or zero).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JG

JNLE and JG (Jump if not less nor equal, or jump if greater)

Operation: If the zero flag is reset and the sign flag equals the overflow flag (i.e., both zero or both 1), then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 1 or (SF) ≠ (OF), then no jump occurs.

```
if ((SF)‖(OF))‖(ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 1 1 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JG   TARGET_LABEL
JNLE  TARGET_LABEL
```

Flags Affected: None

Description: JNLE (or JG) transfers control to the target operand on not less or equal (or greater).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

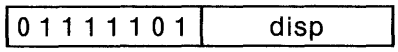
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNL and JGE (Jump if not less, or jump if greater or equal)

Operation: If the sign flag equals the overflow flag, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (SF) ≠ (OF), no jump results.

```
if (SF)‖(OF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JGE  TARGET_LABEL
JNL  TARGET_LABEL
```

Flags Affected: None

Description: JNL (or JGE) transfers control to the target operand on not less (or greater or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

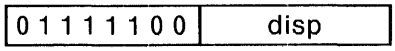
JL

JL and JNGE (Jump on less, or jump on not greater nor equal)

Operation: This jump occurs only if the sign flag is not equal to the overflow flag, i.e., (SF) ≠ (OF). (SF) exclusive-or (OF) = 1 means the same. If (SF) ≠ (OF), then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (SF) = (OF), no jump occurs.

```
if (SF)‖(OF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JL   TARGET_LABEL
JNGE LABEL_TARGET
```

Flags Affected: None

Description: JL (or JNGE) transfers control to the target operand on less (or not greater or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JLE and JNG (Jump if less or equal, or jump if not greater)

Operation: If the zero flag is set, or if the sign flag is unequal to the overflow flag, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 0, and (SF) = (OF), then no jump occurs.

```
if ((SF)‖(OF))|(ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 1 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JLE  TARGET_LABEL
JNG  TARGET_LABEL
```

Flags Affected: None

Description: JLE (or JNG) transfers control to the target operand on less or equal (or not greater).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JMP (Jump)

Operation: The Instruction Pointer is replaced by the target's offset in all intersegment jumps, and in intra-segment (or intra-group) indirect jumps.

When the jump is a direct intra-segment or intra-group, the distance from the end of this instruction to the target label is added to the IP.

Intersegment jumps first replace the contents of CS, using the second word following the instruction (direct) or using the second word following the indicated data address (indirect).

```
if Inter-Segment then (CS) ← SEG
(IP) ← DEST
```

See note.

Encoding:

Intra-Segment or Intra-Group Direct:

1 1 1 0 1 0 0 1	disp-low	disp-high

DEST = (IP) + disp

Timing: 7 clocks

Example: JMP NEAR_LABEL

Intra-Segment Direct Short:

1 1 1 0 1 0 1 1	disp

DEST = (IP) + disp sign extended to 16 bits

Timing: 1 clock

Examples:

```
JMP  TARGET_LABEL
JMP  SHORT  NEAR_LABEL
```

NOTE: The target label must be within −128 to +127 bytes of this instruction.

Inter-Segment Direct:

1 1 1 0 1 0 1 0	offset-low	offset-high	seg-low	seg-high

DEST = offset, SEG = seg

Timing: 7 clocks

Examples:

```
JMP  LABEL_DECLARED_FAR
JMP  FAR  PTR  LABEL_NAME
JMP  FAR  PTR  NEAR_LABEL
```

Inter-Segment Indirect:

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

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

Timing: 16 + EA clocks

Examples:

```
JMP  VAR_DOUBLEWORD
JMP  DWORD  PTR  [BX] [SI]
JMP  ALPHA [BP] [DI]
```

Intra-Segment or Intra-Group Indirect:

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

DEST = (EA)

Timing: 7 + EA clocks

Examples:

```
JMP  TABLE [BX]
JMP  WORD PTR  [BX] [DI]
JMP  BETA_WORD
JMP  AX
JMP  SI
JMP  BP
;these replace the Instruction Pointer by the contents of the named
;register. This causes a jump directly to the byte with that offset past
;CS. This is different from the direct intra-segment jumps, which are
;self-relative, causing an add to the IP.
```

Flags Affected: None

Description: JMP transfers control to the target operand.

The jump is always relative to the segment base address in the CS register. A direct jump directly uses the offset (and segment, if "long") bytes that follow the instruction byte. Indirect jumps use the contents of the location addressed by the bytes that follow the instruction byte.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

JNA and JBE (Jump if below or equal, or jump if not above)

Operation: If either the carry flag or the zero flag is set, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If both (CF) = 0 and (ZF) = 0, no jump occurs.

```
if (CF)|(ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JBE  TARGET_LABEL
JNA  TARGET_LABEL
```

Flags Affected: None

Description: JBE (or JNA) transfers control to the target operand on below or equal (or not above).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

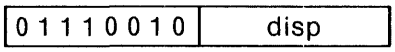
JNAE

JNAE and JB (Jump if below, or jump if not above nor equal)

Operation: If the carry flag is 1, then the distance from the end of this instructon to the target label is added to the Instruction Pointer, effecting the jump. If (CF) = 0, no jump occurs.

```
if (CF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JB   TARGET_LABEL
JNAE  TARGET_LABEL
```

Flags Affected: None

Description: JB (or JNAE) transfers control to the target operand on below (or not above or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNB and JAE (Jump if not below, or jump if above or equal)
JNC (Jump if no carry)

Operation: If the carry flag is zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (CF) = 1, no jump occurs.

```
if (CF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 0 1 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JNB  TARGET_LABEL
JAE  TARGET_LABEL
JNC  TARGET_LABEL
```

Flags Affected: None

Description: JNB (or JAE) transfers control to the target operand on not below (or above or equal).

NOTE: The target label must be within −128 to + 127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNBE (Jump if not below nor equal)

Operation: If neither the carry flag nor the zero flag is set, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (CF) = 1 or if (ZF) = 1, no jump occurs.

```
if (CF)|(ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 1 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JNBE  TARGET_LABEL
JA  TARGET_LABEL
```

Flags Affected: None

Description: JNBE (or JA) transfers control to the target operand on not below or equal (or above).

NOTE: The target label must be within −128 to + 127 bytes of this instruction.

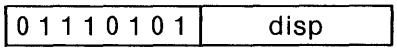
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNE and JNZ (Jump if not equal, or jump if not zero)

Operation: If the zero flag is reset, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 1, no jump occurs.

```
if (ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JNE  TARGET_LABEL
JNZ  TARGET_LABEL
```

Flags Affected: None

Description: JNE (or JNZ) transfers control to the target operand on not equal (or not zero).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNG

JNG and JLE (Jump if not greater, or jump if less or equal)

Operation: If the zero flag is set, or if the sign flag is unequal to the overflow flag, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 0, and (SF) = (OF), then no jump occurs.

```
if ((SF)‖(OF))|(ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 1 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
JLE  TARGET_LABEL
JNG  TARGET_LABEL
```

Flags Affected: None

Description: JLE (or JNG) transfers control to the target operand on less or equal (or not greater).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

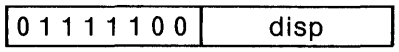
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JL and JNGE (Jump if less, or jump if not greater nor equal)

Operation: If the sign flag is unequal to the overflow flag, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (SF) = (OF), no jump occurs.

```
if (SF) || (OF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
JL   TARGET_LABEL
JNGE TARGET_LABEL
```

Flags Affected: None

Description: JL (or JNGE) transfers control to the target operand on less (or not greater or equal).

The target label must be within −128 to + 127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

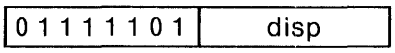
JNL

JNL and JGE (Jump if not less, or jump if greater or equal)

Operation: If the sign flag equals the overflow flag, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (SF) ≠ (OF), no jump results.

```
if (SF)‖(OF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
JGE  TARGET_LABEL
JNL  TARGET_LABEL
```

Flags Affected: None

Description: JNL (or JGE) transfers control to the target operand on not less (or greater or equal).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNLE and JG (Jump if not less nor equal, or jump if greater)

Operation: If the zero flag is reset and the sign flag equals the overflow flag (i.e., both zero or both 1), then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 1 or (SF) ≠ (OF), then no jump occurs.

```
if ((SF)‖(OF))|(ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 1 1 1	disp

Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
JG   TARGET_LABEL
JNLE  TARGET_LABEL
```

Flags Affected: None

Description: JNLE (or JG) transfers control to the target operand on not less or equal (or greater).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

JNO (Jump on not overflow)

Operation: If overflow flag is 1, no jump occurs. If (OF) = 0, the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a jump to that location.

```
if (OF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 0 0 1	disp

Timing (clocks):

Jump	is taken	8
Jump is not taken		4

Example: JNO TARGET_LABEL

Flags Affected: None

Description: JNO transfers control to the target operand on no overflow.

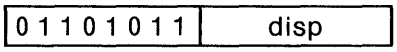
NOTE: The target label must be within −128 to +127 bytes of this instruction.

JNP and JPO (Jump on no parity or jump if parity odd)

Operation: If the parity flag is 1, meaning even parity resulted from the last operation to affect PF, then no jump occurs. If (PF) = 0, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a transfer to that location.

```
if (PF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding



Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
1) JNP  TARGET_LABEL
2) JPO  TARGET_LABEL
```

Flags Affected: None

Description: JNP (or JPO) transfers control to the target operand on not parity (or parity odd).

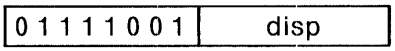
NOTE: The target label must be within −128 to +127 bytes off this instruction.

JNS (Jump on not sign, jump if positive)

Operation: If the sign flag is not set, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a transfer. If (SF) = 1, no jump occurs.

```
if (SF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Example: JNS TARGET_LABEL

Flags Affected: None

Description: JNS transfers controls to the target operand on not sign.

NOTE: The target label must be within −128 to +127 bytes of this instruction.

JNE and JNZ

Operation: If the new flag is reset, then the distance from the end of instruction to the target label is added to the Instruction Pointer, effecting the jump. If (ZF) = 1, no jump occurs.

```
if (ZF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 0 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
1) JNE  TARGET_LABEL
2) JNZ  TARGET_LABEL
```

Flags Affected: None

Description: JNE (or JNZ) transfers control to the target operand on not equal (or not zero).

JO (Jump on overflow)

Operation: If the overflow flag is 1, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (OF) = 0 no jump occurs.

```
if (OF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16 bits)
```

Encoding:

0 1 1 1 0 0 0 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Example: JO TARGET_LABEL

Flags Affected: None

Description: JO transfers control to the target operand on overflow.

NOTE: The target label must be within −128 to +127 bytes of this instruction.

JP and JPE (Jump on parity, or jump if parity even)

Operation: If the parity flag is 1, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (PF) = 0, no jump occurs.

```
if (PF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 0 1 0	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
1) JP   TARGET_LABEL
2) JPE  TARGET_LABEL
```

Flags Affected: None

Description: JP (or JPE) transfers control to the target operand on parity (or parity even).

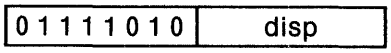
NOTE: The target label must be within −128 to +127 bytes of this instruction.

JP and JPE (Jump on parity, or jump if parity even)

Operation: If the parity flag is 1, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (PF) = 0, no jump occurs.

```
if (PF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
1) JP   TARGET_LABEL
2) JPE  TARGET_LABEL
```

Flags Affected: None

Description: JP (or JPE) transfers control to the target operand on parity (or parity even).

NOTE: The target label must be within −128 to +127 bytes of this instruction.

JNP and JPO

Operation: If the parity flag is 1, meaning even parity resulted from the last operation to affect PF, then no jump occurs. If (PF) = 0, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting a transfer to that location.

```
if (PF) = 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 0 1 1	disp

Timing (clocks):

Jump is taken	8
Jump is not taken	4

Examples:

```
1) JNP  TARGET_LABEL
2) JPO  TARGET_LABEL
```

Flags Affected: None

Description: JNP (or JPO) transfers control to the target operand on not parity (or parity odd).

NOTE: The target label must be within −128 to + 127 bytes of this instruction.

JS (Jump on sign)

Operation: If the sign flag is 1, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (SF) = 0, no jump occurs.

```
if (SF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 1 0 0 0	disp

Timing (clocks): Jump is taken 8
Jump is not taken 4

Example: JS TARGET_LABEL

Flags Affected: None

Description: JS transfers control to the target operand on sign.

NOTE: The target label must be within −128 to +127 bytes of this instruction.

JZ and JE (Jump if equal, jump if zero)

Operation: If the last operation to affect the zero flag gave a result of zero, then (ZF) will be 1. If (ZF) = 1, then the distance from the end of this instruction to the target label will be added to the Instruction Pointer, effecting a transfer of control to that label. If (ZF) = 0, no operation occurs.

```
if (ZF) = 1 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

0 1 1 1 0 1 0 0	disp

Timing (clocks): Jump is taken 8
Jump is not taken 4

Examples:

```
1) CMP  CX,  DX
   JE   LAB2
   INC  CX
   LAB2:
   ;the increment of CX will only occur if CX = DX

2) SUB  AX,  BX
   JZ   EXACT
   ;jump occurs if result was zero, i.e., AX = BX
   .
   .
   .
   EXACT:
```

Flags Affected: None

Description: JE (or JZ) transfers control to the target operand on equal (or zero).

NOTE: The target label must be within −128 to + 127 bytes on this instruction

LAHF

LAHF (Load AH from flags)

Operation: Specific bits of AH are filled from the following flags: The sign flag fills bit 7. The zero flag fills bit 6. The auxiliary carry flag fills bit 4. The parity flag fills bit 2. The carry flag fills bit 0. Bits 1, 3, and 5 of AH are indeterminate, i.e., they may on some occasions be 1 and at other times be 0.

(AH) ← (SF):(ZF):X:(AF):X:(PF):X:(CF)

Encoding:

1	0	0	1	1	1	1	1

Timing: 4 clocks

Example: LAHF

Flags Affected: None

Description: LAHF (Load AH with Flags) transfers the flag registers SF, ZF, AF, PF, and CF (which, when 8080 code is translated into 8086 code, are the 8080 flags) into specific bits of the AH register. The bits indicated "X" are unspecified.

LDS (Load data segment register)

Operation:

1) The contents of the specified register are replaced by the lower addressed word of the doubleword memory operand.

   (REG) ← (EA)

2) The contents of the DS register are replaced by the higher-addressed word of the doubleword memory operand.

   (DS) ← (EA + 2)

See note.

Encoding:

1 1 0 0 0 1 0 1	mod reg r/m

for mod ≠ 11 (if mod = 11 then undefined operation)

Timing: 16+EA clocks

Examples:

```
LDS  BX,  ADDR_TABLE [SI]
LDS  SI,  NEWSEG [BX]
```

Flags Affected: None

Description: 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 may be transferred to any 16-bit general, pointer, or index register you specify (not a segment register).

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions refering directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

LEA (Load effective address)

Operation: The contents of the specified register are replaced by the offset of the indicated variable or label or address-expression.

(REG) ← EA

See note.

Encoding:

1 0 0 0 1 1 0 1	mod reg r/m

for mod ≠ 11 (if mod = 11 then undefined operation)

Timing: 2+EA clocks

Examples:

```
LEA  BX,  VARIABLE_7
LEA  DX,  BETA [BX] [SI]
LEA  AX,  [BP] [DI]
```

Flags Affected: None

Description: 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 can be any 16-bit general, pointer, or index register. LEA allows the source to be subscripted. This is not allowed using the MOV instruction with the OFFSET operator. Also, the latter operation invariably uses the offset of the variable in the segment where it was defined. LEA, however, will take into account a group offset if the group is the only possible access route via the latest ASSUME directive.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

LES (Load extra-segment register)

Operation:

1) The contents of the specified register are replaced by the lower addressed word of the doubleword memory operand.

   (REG) ← (EA)

2) The contents of the ES register are replaced by the higher-addressed word of the doubleword memory operand.

   (ES) ← (EA + 2)

See note.

Encoding:

1 1 0 0 0 1 0 0	mod reg r/m

for mod ≠ 11 (if mod = 11 then undefined operation)

Timing: 16+EA clocks

Examples:

```
LES   BX,   ADDR_TABLE [SI]
LES   DI,   NEWSEG [BX]
```

Flags Affected: None

Description: LES (Load Pointer into ES) 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 ES segment register. The offset address may be transferred to a 16-bit general, pointer, or index register (not a segment register).

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the instruction bytes will be followed by 2 bytes giving the computed displacement from the segment-base-address.

LOCK

LOCK

Operation: None

Encoding:

1 1 1 1 0 0 0 0

Timing: 2 clocks

Example: LOCK

Flags Affected: None

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.)

LODS (Load byte or word string)

Chapter 2 describes LODSB and LODSW.

Operation: The source byte (or word) is loaded into AL (or AX). The Source Index is incremented by 1 (or 2, for word strings) if the Direction Flag is reset; otherwise SI is decremented by 1 (or 2).

```
(DEST) ← (SRC)
if (DF) = 0 then (SI) ← (SI) + DELTA
else (SI) ← (SI)−DELTA
```

Encoding:

1 0 1 0 1 1 0 w

1) if w = 0 then SRC = (SI), DEST = AL, DELTA = 1
2) else SRC = (SI) + 1:(SI), DEST = AX, DELTA = 2

Timing: 12 clocks

Examples:

```
1) CLD   ;clears direction flags so SI will be incremented
   MOV   SI,  OFFSET  BYTE_STRING
   LODS  BYTE_STRING      ;SI ← SI+1
      .
      .
      .

2) STD   ;sets DF so SI will be decremented
   MOV   SI,  OFFSET  WORD_STRING
   LODS  WORD_STRING      ;SI ← SI-2
   ;DF = 1 implies that the variable
   ;WORD_STRING names the last or
   ;highest-addressed word in the string. The operand named in the
   ;LODS instruction is used only by the assembler to verify type and
   ;accessibility using correct segment register contents. LODS
   ;actually uses only SI to point to the location whose contents are to
   ;be loaded into the accumulator, without using the name given in the
   ;source instruction
```

Flags Affected: None

Description: LODS transfers a byte (or word) operand from the source operand addressed by SI to accumulator AL (or AX) and adjusts the SI register by DELTA. This operation ordinarily would not be repeated.

LOOP
(Loop, or iterate instruction sequence until count complete)

Operation: The Count register (CX) is decremented by 1. If the new CX is not zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If CX = 0, no jump occurs.

```
(CX) ← (CX)−1
if (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

1 1 1 0 0 0 1 0

Timing (clocks): Jump is taken 9
Jump is not taken 5

Example: The following sequence will compute the 16-bit checksum of a non-null array:

```
(1)     MOV  CX,  LENGTH  ARRAY
        MOV  AX,  0
        MOV  SI,  AX

NEXT:ADD  AX,  ARRAY [SI]
        ADD  SI,  TYPE ARRAY
        LOOP  NEXT
        MOV  CKS,  AX

(2)     MOV  AX,  0
        MOV  BX,  1
        MOV  CX,  N  ;number of terms
        MOV  DI,  AX

FIB:    MOV  SI,  AX
        ADD  AX,  BX
        MOV  BX,  SI
        MOV  FIBONACCI [DI],  AX
        ADD  DI,  TYPE FIBONACCI

LL:     LOOP  FIB

        ;the instructions from FIB to LL will be executed N times
        ;and will store into the FIBONACCI array the first N terms of that sequence
        ;i.e., 1, 1, 2, 3, 5, 8, 13, 21, .....
```

Flags Affected: None

Description: LOOP decrements the CX (count) register by 1 and transfers control to the target operand (label) if CX is not zero.

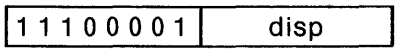
The target label must be within −128 to + 127 bytes of this instruction.

LOOPZ and LOOPE (Loop on equal, or loop on zero)

Operation: The Count register (CX) is decremented by 1. If the zero flag is set and (CX) is not yet zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. No jump occurs if (ZF) = 0 or if (CX) = 0.

```
(CX) ← (CX)−1
if (ZF) = 1 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks): Jump is taken 11
Jump is not taken 5

Example: The following sequence finds the first non-zero entry in a byte array:

```
          MOV  CX,  LENGTH  ARRAY
          MOV  SI,  −1

NEXT:     INC    SI
          CMP  ARRAY [SI],  0
          LOOPE  NEXT
          JNE  OKENTRY
          .                    ;arrive here if whole array is zero
          .
          .
OKENTRY:                       ;SI tells which entry is non-zero
```

Flags Affected: None

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

The target label must be within −128 to +127 bytes of this instruction.

LOOPNE

LOOPNZ and LOOPNE (Loop on not zero, or loop on not equal)

Operation: The Count register (CX) is decremented by 1. If the new (CX) is not zero and the zero flag is reset, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (CX) = 0 or if (ZF) = 1, then no jump occurs.

```
(CX) ← (CX)−1
if (ZF) = 0 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

1 1 1 0 0 0 0 0	disp

Timing (clocks): Jump is taken 11
Jump is not taken 5

Examples: The following sequence will compute the sum of 2 byte arrays, each of length N, only up to the point of encountering zero entries in both arrays at the same time. At that point the expression SI−1 will give the length of the non-zero sum arrays.

```
        MOV   AX,  0
        MOV   SI−1
        MOV   CX,  N
NONZER: INC   SI
        MOV   AL,  ARRAY1 [SI]
        ADD   ,ARRAY2 [SI]
        MOV   SUM [SI],  AX
        LOOPNZ  NONZER
```

The following sequence will search down a linked list for the last element. This will be the element with a zero in the word that normally contains the address of the next element. This word is always located the same number of bytes past each list element's beginning. LINK is the name for that absolute number of bytes, e.g.,

```
        LINK  EQU  7
        MOV   AX,  OFFSET HEAD_OF_LIST
        MOV   CX,  1000  ;search at most 1000 entries
NEXT:   MOV   BX,  AX
        MOV   AX, [BX] + LINK
        CMP   AX,  0
        LOOPNE  NEXT
```

Flags Affected: None

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

The target label must be within −128 to + 127 bytes of this instruction.

LOOPNZ and LOOPNE (Loop on not zero, or loop on not equal)

Operation: The Count register (CX) is decremented by 1. If the new (CX) is not zero and the zero flag is reset, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. If (CX) = 0 or if (ZF) = 1, then no jump occurs.

```
(CX) ← (CX)−1
if (ZF) = 0 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:

1 1 1 0 0 0 0 0	disp

Timing (clocks): Jump is taken 11
Jump is not taken 5

Examples: The following sequence will compute the sum of 2 byte arrays, each of length N, only up to the point of encountering zero entries in both arrays at the same time. At that point the expression SI−1 will give the length of the non-zero sum arrays.

```
        MOV   AX,  0
        MOV   SI−1
        MOV   CX,  N
NONZER: INC   SI
        MOV   AL,  ARRAY1 [SI]
        ADD   ,ARRAY2 [SI]
        MOV   SUM [SI],  AX
        LOOPNZ  NONZER
```

The following sequence will search down a linked list for the last element. This will be the element with a zero in the word that normally contains the address of the next element. This word is always located the same number of bytes past each list element's beginning. LINK is the name for that absolute number of bytes, e.g.,

```
        LINK  EQU  7
        MOV   AX,  OFFSET HEAD_OF_LIST
        MOV   CX,  1000  ;search at most 1000 entries
NEXT:   MOV   BX,  AX
        MOV   AX, [BX] + LINK
        CMP   AX,  0
        LOOPNE  NEXT
```

Flags Affected: None

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

The target label must be within −128 to + 127 bytes of this instruction.

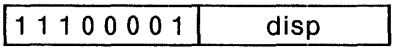
LOOPZ

LOOPZ and LOOPE (Loop on equal, or loop on zero)

Operation: The Count register (CX) is decremented by 1. If the zero flag is set and (CX) is not yet zero, then the distance from the end of this instruction to the target label is added to the Instruction Pointer, effecting the jump. No jump occurs if (ZF) = 0 or if (CX) = 0.

```
(CX) ← (CX)−1
if (ZF) = 1 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Encoding:



Timing (clocks): Jump is taken 11
Jump is not taken 5

Example: The following sequence finds the first non-zero entry in a byte array:

```
            MOV  CX,  LENGTH  ARRAY
            MOV  SI,  −1

NEXT:       INC   SI
            CMP  ARRAY[SI],  0
            LOOPZ  NEXT
            JNE  OKENTRY
                              ;arrive here if whole array is zero.
            .
            .
            .
OKENTRY:                      ;SI tells which entry is non-zero
```

Flags Affected: None

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

The target label must be within −128 to + 127 bytes of this instruction.

MOV (Move)

There are 7 separate types of move instructions, as shown below.

Each type has multiple uses and encodings depending on the type of data being moved and the location of that data. The assembler generates the correct encoding based on these 2 factors.

If the destination is a register, the bit shown as "d" will be 1, otherwise 0. If the type is a word, the bit shown as "w" will be 1, otherwise 0.

See note.

Type 1: TO Memory FROM Accumulator

1 0 1 0 0 0 1 w	addr-low	addr-high

If w=0 then SRC=AL, DEST=addr else SRC=AX, DEST=addr + 1: addr

Timing (clocks): 10 + EA

Examples:

```
MOV  ALPHA_MEM,  AX
MOV  GAMMA_BYTE,  AL

MOV  CS:DATUM_BYTE,  AL
MOV  ES:ARRAY [BX] [SI],  AX
     (prefix byte, e.g., ES:, will precede instruction byte; see
      beginning of this Chapter)
```

Type 2: TO Accumulator FROM Memory

1 0 1 0 0 0 0 w	addr-low	addr-high

If w=0 then SRC=addr, DEST=AL else SRC=addr + 1: addr, DEST=AX

Timing (clocks): 8 + EA

Examples:

```
MOV  AX,  BETA_MEM
MOV  AL,  GAMMA_BYTE

MOV  AX,  ES:ARRAY [BX] [SI]
MOV  AL,  SS:OTHER_BYTE
     (prefix byte, e.g., ES:, will precede instruction byte; see
     beginning of this Chapter).
```

Type 3: TO Segment Register FROM Memory-or-Register Operand

1 0 0 0 1 1 1 0	mod 0reg r/m

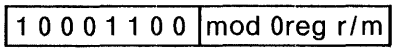
if reg ≠ 01 then SRC=EA, DEST=REG else undefined operation

Timing: register to register 2
memory to register 8 + EA

Examples:

```
MOV  ES,  DX
MOV  DS,  AX
MOV  SS,  BX
MOV  ES,  SS:NEW_WORD [DI]
Note: CS is illegal as a destination here.
```

Type 4: TO Memory-or-Register FROM Segment Register



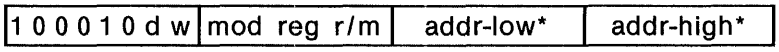
SRC=REG, DEST=EA,(DEST) ← (SRC)

Timing (clocks): memory to register 9 + EA
register to register 2

Examples:

```
MOV  DX,  DS
MOV  BX,  ES
MOV  ARRAY [BX] [SI],  SS
MOV  BETA_MEM_WORD,  DS
MOV  GAMMA, CS;  Note: CS is legal as a source here.
```

Type 5: (a) TO Register FROM Register
(b) TO Register FROM Memory-or-Register Operand
(c) TO Memory-or-Register Operand FROM Register



if d=1 then SRC=EA, DEST=REG else SRC=REG, DEST=EA

*these bytes omitted in register to register moves, i.e., when mod=11,

```
MOV    CX,  DX
```

and also when the address-expression to memory is register-indirect with no variable-name-displacement, i.e.,

```
MOV    [BX] [SI],  DX
MOV    AX, [BP] [DI]
```

Timing (clocks): (a) 2
(b) 8 + EA
(c) 9 + EA

Examples:

```
(a) MOV  AX,  BX
    MOV  CL,  DH
    MOV  CX,  DI

(b) MOV  AX,  MEM_VALUE
    MOV  DX,  ARRAY [SI]
    MOV  DI,  MEM [BX] [DI]

(c) MOV  ARRAY [DI],  DX
    MOV  MEM_VALUE,  AX
    MOV  [BX] [SI],  DI
```

Type 6: TO Register FROM Immediate-data

1 0 1 1 w reg	data	data-high*

SRC=data, DEST=REG

*present only if w = 1

Timing (clocks): 4

Examples:

```
MOV  AX,  77
MOV  BX,  VALUE_14_IMM
MOV  SI,  EQU_VAL_9
MOV  DI,  618
```

Type 7: TO Memory-or-Register Operand FROM Immediate-data

1 1 0 0 0 1 1 w	mod 000 r/m	data	data-high*

SRC=data, DEST=EA

*present only if w=1

Timing (clocks): 10 + EA

Examples:

```
MOV  ARRAY [BX] [SI],  DATA_4
MOV  MEM_BYTE,  IMM_BYTE_3
MOV  BYTE  PTR  [DI],  66
MOV  MEM_WORD,  1999
MOV  BX,  84
MOV  DS:MEM_WORD [BP],  3989
```

(prefix byte, e.g., DS:, of 00111110 will precede 1100011w above)

Flags Affected: None

MOV

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

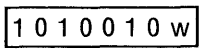
MOVS (Move byte string or move word string)

Chapter 2 describes MOVSB and MOVSW.

Operation: The source string whose offset is in the Source Index is moved into the location in the extra segment whose offset is in the Destination Index. SI and DI are then both incremented, if the direction flag is zero, or both decremented, if (DF) = 1. The increment or decrement is 1 for byte strings, 2 for word strings.

```
(DEST) ← (SRC)
if (DF) = 0 then
  (SI) ← (SI) + DELTA
  (DI) ← (DI) + DELTA
else
  (SI) ← (SI)–DELTA
  (DI) ← (DI)–DELTA
```

Encoding:



if w = 0 then SRC = (SI), DEST = (DI), DELTA = 1
else SRC = (SI) + 1:(SI), DEST = (DI) + 1:(DI), DELTA = 2

Timing: 17 clocks

Example:

```
        MOV  SI,  OFFSET  SOURCE
        MOV  DI,  OFFSET  DEST
        MOV  CX,  LENGTH  SOURCE
REP     MOVS  DEST,  SOURCE
        ;the above sequence moves the entire source string (in any
        ;segment reachable by current segment registers) into the
        ;destination locations in the Extra Segment (the ES register is
        ;always used for DI operands in string operations). See also
        ;REP. The operands named in the string operation are used
        ;only by the assembler to verify type and accessibility using
        ;current segment registers contents. MOVS actually moves the
        ;byte pointed at by SI to the byte pointed at by DI in ES, without
        ;using the names given in the source MOVS instruction.
```

Flags Affected: None

Description: MOVS transfers a byte (or word) operand from the source operand addressed by SI to the destination operand addressed by DI, and adjusts the SI and DI registers by DELTA. As a repeated operation this provides for moving a string from one location in memory to another.

MUL

MUL (Multiply accumulator by register-or-memory; unsigned)

Operation: The accumulator (AL if byte, AX if word) is multiplied by the specified operand. If the high order half of the result is zero, then the carry and overflow flags are reset, otherwise they are set.

```
(DEST) ← (LSRC) * (RSRC), where * is unsigned
  multiply
if (EXT) = 0 then (CF) ← 0
else (CF) ← 1;
(OF) ← (CF)
```

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

(a) if w = 0 then LSRC = AL, RSRC = EA, DEST = AX, EXT = AH
(b) else LSRC = AX, RSRC = EA, DEST = DX:AX, EXT = DX

Timing (clocks): 8-bit 71 + EA
16-bit 124 + EA

Example:

```
a)  MOV  AL,  LSRC_BYTE
    MUL  RSRC_BYTE  ;result in AX

b1) MOV  AX,  LSRC_WORD
    MUL  RSRC_WORD
    ;high-half result in DX, low-half in AX

b2) to multiply a byte by a word
    MOV  AL,  MUL_BYTE
    CBW  ;converts byte in AL to word in AX
    MUL  RSRC_WORD
```

NOTE: Any memory operand above could be an indexed addressed-expression of the correct TYPE, e.g., LSRC_BYTE could be ARRAY [SI] if ARRAY were of type BYTE, and RSRC_WORD could be TABLE [BX] [DI] if TABLE were of type WORD.

Flags Affected: CF, OF.
Undefined: AF, PF, SF, ZF

Description: MUL (multiply) 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, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is nonzero.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

NEG (Negate, or form 2's complement)

Operation: The specified operand is subtracted from all ones (0FFH for bytes, 0FFFFH for words), 1 is added, and the result stored back into the given operand.

(EA) ← SRC−(EA)
(EA) ← (EA) + 1 (affecting flags)

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

if w = 0 then SRC = 0FFH
else SRC = 0FFFFH

Timing (clocks): register 3
memory 16 + EA

Examples:

1) If AL contains 13H (00010011), then NEG AL causes AL to contain −13H or 0EDH (11101101).

2) If MEM_BYTE contains 0AFH (10101111), then NEG MEM_BYTE causes MEM_BYTE to contain −0AFH or 51H (01010001).

3) If SI contains 2FC3H, then NEG SI causes SI to contain 0D03DH.

(See also NOT.)

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: NEG (negate) performs a subtraction of the operand from zero, adds 1, and returns the result to the operand. This forms the 2's complement of the specified operand.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

NOP (No operation)

Operation: None

Encoding:

1	0	0	1	0	0	0	0

Timing: 3 clocks

Example: NOP

Flags Affected: None

Description: NOP causes no operation and takes 3 clocks. The next sequential instruction is then executed.

NOT (Not, or form 1's complement)

Operation: The specified operand is subtracted from 0FFH (or 0FFFFH, if a word) and the result is stored back into the given operand.

(EA) ← SRC−(EA)

See note.

Encoding:

1 1 1 1 0 1 1 w	mod 0 1 0 r/m

if w = 0 then SRC = 0FFH
else SRC = 0FFFFH

Timing (clocks):

register	3
memory	16 + EA

Examples:

1) If AH contains 13H (00010011), then NOT AH causes AH to contain 0ECH (11101100).

2) If MEM_BYTE contains 0AFH (10101111), then NOT MEM_BYTE causes MEM_BYTE to contain 50H (01010000).

3) If DX contains 2FC3H, then NOT DX causes DX to contain 0D03CH.

See also NEG.

Flags Affected: None

Description: NOT forms the ones complement of (inverts) the operand and returns the result to the operand. Flags are not affected.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

OR (Or, inclusive)

Operation: Each bit position in the destination (leftmost) operand becomes 1, unless it and the corresponding bit position of the source (rightmost) operand were both 0. Alternative phrasing: each bit position of the result has a 1 if either operand had a 1 in that position; if both had a 0, that position of the result has a zero. The carry and overflow flags are both reset.

(DEST) ← (LSRC)|(RSRC)
(CF) ← 0
(OF) ← 0

See note.

Encoding:

Memory or Register Operand with Register Operand:

0 0 0 0 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks):

(a)	register to register	3
(b)	memory to register	9+EA
(c)	register to memory	16+EA

Examples:

```
(a) OR   AH,   BL    ;result in AH,   BL unchanged
    OR   SI,   DX    ;result in SI,   DX unchanged
    OR   CX,   DI    ;result in CX,   DI unchanged

(b) OR   AX,   MEM_WORD
    OR   CL,   MEM_BYTE [SI]
    OR   SI,   ALPHA [BX] [SI]

(c) OR   BETA [BX] [DI],   AX
    OR   MEM_BYTE,   DH
    OR   GAMMA [DI],   BX
```

Immediate Operand to Accumulator:

0 0 0 0 1 1 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

Timing (clocks): immediate to register 4

OR

Examples:

```
a) OR   AL,  11110110B
   OR   AL,  0F6H

b) OR   AX,  23F6H
   OR   AX,  75Q
   OR   ,23F6H
```

Immediate Operand to Memory or Register Operand:

1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate to register 4
(b) immediate to memory 17 + EA

Examples:

```
a) OR   AH,  0F6H
   OR   CL,  37
   OR   DI,  23F5H

b) OR   MEM_BYTE,  3DH
   OR   GAMMA [BX] [DI],  0FACEH
   OR   ALPHA [DI],  VAL_EQUD_33H
```

Flags Affected: CF, OF, PF, SF, ZF.
Undefined: AF

Description: OR performs the bitwise logical inclusive disjunction of the two operands and returns the result to one of the operands.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

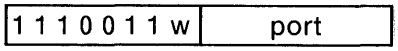
OUT (Output byte and output word)

Operation: The contents of the designated port are replaced by the contents of the accumulator.

(DEST) ← (SRC)

Encoding:

Fixed Port:



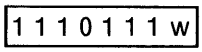
if w = 0 then SRC = AL, DEST = port
else SRC = AX, DEST = port + 1:port
(0 < port < 255)

Timing: 10 clocks

Examples:

```
OUT   BYTE_PORT_VAL,AL     ;outputs a byte from AL
OUT   WORD_PORT_VAL,AX     ;outputs a word from AX
OUT   44,AX      ;outputs a word from AX through port 44
```

Variable Port:



if w = 0 then SRC = AL, DEST = (DX)
else SRC = AX, DEST = (DX) + 1:(DX)

Timing: 8 clocks

Examples:

```
OUT   DX,AL      ;outputs a byte from AL through variable port in DX
OUT   DX,AX      ;outputs a word from AX through variable port in AX
```

Flags Affected: None

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

POP (Pop word off stack into destination)

There are 3 separate types of POP instructions, for different destinations.

See note.

Operation:

1) The contents of the destination are replaced by the word at the top of the stack
   (DEST) ← ((SP) + 1:(SP))

2) The stack pointer is incremented by 2.
   (SP) ← (SP) + 2

Flags Affected: None

Type 1:

Register Operand:

0 1 0 1 1 reg

DEST = REG

Timing: 8 clocks

Examples:

```
POP   CX
The assembler generates     0 1 0 1 1 0 0 1
POP   DX
The assembler generates     0 1 0 1 1 0 1 0
```

Type 2:

Segment Register:

0 0 0 reg 1 1 1

if reg ≠ 01 then DEST = REG
else undefined operation

Note: POP CS is not legal

Timing: 8 clocks

Examples:

```
POP   SS
The assembler generates     0 0 0 1 0 1 1 1
POP   DS
The assembler generates     0 0 0 1 1 1 1 1
```

Type 3:

Memory or Register Operand:

1 0 0 0 1 1 1 1	mod 0 0 r/m

DEST = EA

Timing (clocks): memory 17+EA
register 8

Examples:

```
                                    POP   ALPHA
The assembler generates  1 0 0 0 1 1 1 1  0 0 0 0 0 1 1 0  ALPHA addr-lo  ALPHA addr-hi

                                    POP   ALPHA [BX]
                         1 0 0 0 1 1 1 1  1 0 0 0 0 1 1 1  ALPHA addr-lo  ALPHA addr-hi
```

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

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes given the computed displacement from the segment-base-address.

POPF

POPF (Pop flags off stack)

Operation:

Flags ← ((SP) + 1:(SP))
(SP) ← (SP) + 2

The flag registers are filled from the appropriate bit positions of the word at the top of the stack, i.e.,

overflow flag	← bit 11
direction flag	← bit 10
interrupt flag	← bit 9
trap flag	← bit 8
sign flag	← bit 7
zero flag	← bit 6
auxiliary carry flag	← bit 4
parity flag	← bit 2
carry flag	← bit 0

Then the Stack Pointer is incremented by 2.

Encoding:

1 0 0 1 1 1 0 1

Timing: 8 clocks

Example: POPF

Flags Affected: All

Description: 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. See also PUSHF.

PUSH (Push word onto stack)

There are 3 separate types of PUSH instructions depending on the kind of operand supplied.

See note.

Operation:

1) The stack pointer (SP) is decremented by 2.
   (SP) ← (SP)−2

2) The contents of the specified operand are placed on the top of stack at the location pointed to by SP. The contents of SP are used as an offset to the stack's base address in register SS.
   ((SP+1):(SP)) ← (SRC)

Flags Affected: None

Type 1:

Register Operand (word)

0 1 0 1 0 reg

Timing (clocks): 10

Examples:

```
PUSH  AX  (generates:  0 1 0 1 0 0 0 0)
PUSH  SI  (generates:  0 1 0 1 0 1 1 0)
```

Type 2:

Segment Register

0 0 0 reg 1 1 0

Timing (clocks): 10

Examples:

```
PUSH  SS  (generates:  0 0 0 1 0 1 1 0)
PUSH  ES  (generates:  0 0 0 0 0 1 1 0)
PUSH  ES
Note: PUSH CS is legal.
```

Type 3:

Memory-or-Register Operand

1 1 1 1 1 1 1 1	mod 1 1 0 r/m

PUSH

Timing (clocks): memory 16 + EA
register 10

Examples:

```
                                    PUSH    BETA
1 1 1 1 1 1 1 1   00 110 110        Beta addr-lo      Beta addr-hi

                                  PUSH    BETA [BX]

1 1 1 1 1 1 1 1   10 110 111        Beta addr-lo      Beta addr-hi

                                PUSH    BETA [BX] [DI]

1 1 1 1 1 1 1 1   10 110 001        Beta addr-lo      Beta addr-hi
```

Description: PUSH decrements the stack pointer SP by 2 and then transfers a word from the service operand to the stack element currently addressed by SP.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

PUSHF (Push flags onto stack)

Operation: The Stack Pointer is decremented by 2, then the flags replace the appropriate bits of the word at the top of the stack (see POPF).

(SP) ← (SP)−2
((SP)+1:(SP)) ← Flags

Encoding:

1 0 0 1 1 1 0 0

Timing: 10 clocks

Example: PUSHF

Flags Affected: None

Description: PUSHF decrements the SP register by 2 and transfers all of the flag registers into specific bits of the word operand (stack element) addressed by SP.

RCL (Rotate left through carry)

Operation: The specified destination (leftmost) operand is rotated left through the carry flag a number of times (COUNT). That number is either exactly once, specified by an absolute number of value 1, or it is the number held in the CL register, specified by a right operand of CL.

The rotation continues until the COUNT is exhausted. CF is preserved and is rotated into bit 0 of the destination. The highest order bit of the destination is rotated into CF. If the COUNT was 1 and the 2 highest-order bits of the original destination value were unequal (one 0 and one 1), then the overflow flag is set. If they were equal, OF is reset. If the COUNT was not 1, OF is undefined and has no reliable value.

```
  (temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (tmpcf)
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 0 1 0 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15 + EA
(c)	variable-bit register	8 + 4/bit
(d)	variable-bit memory	20 + EA + 4/bit

Examples:

```
(a) RCL  AH,  1
    RCL  BL,  1
    RCL  CX,  1
    VAL_ONE  EQU  1
    RCL  DX,  VAL_ONE
    RCL  SI,  VAL_ONE

(b) RCL  MEM_BYTE,  1
    RCL  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    RCL  DH,  CL  ;rotates 3 bits left
    RCL  AX,  CL
```

```
(d) MOV  CL,  6
    RCL  MEM_WORD,  CL  ;rotates 6 times
    RCL  GANDALF_BYTE,  CL
    RCL  BETA [BX] [DI],  CL
```

Flags Affected: CF, OF

Description: RCL (rotate through carry flag left) rotates the operand left through the CF flag register by COUNT bits. See also ROL.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

RCR (Rotate right through carry)

Operation: The specified destination (leftmost) operand is rotated right through the carry flag a number of times (COUNT). That number is either exactly once, specified by an absolute number of value 1, or it is the number held in the CL register, specified by a right operand of CL.

The rotation continues until the COUNT is exhausted. CF is preserved and is rotated into the high order bit of the destination. The lowest order order bit of the destination is rotated into CF. If the COUNT was 1 and the 2 highest-order bits of the destination value are now unequal (one 0 and one 1), then the overflow flag is set. If they were equal, OF is reset. If the COUNT was not 1, OF is undefined and has no reliable value.

```
  (temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (tmpcf)
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 0 1 1 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15 + EA
(c)	variable-bit register	8 + 4/bit
(d)	variable-bit memory	20 + EA + 4/bit

Examples:

```
(a) RCR  AH,  1
    RCR  BL,  1
    RCR  CX,  1
    VAL_ONE  EQU  1
    RCR  DX,  VAL_ONE
    RCR  SI,  VAL_ONE

(b) RCR  MEM_BYTE,  1
    RCR  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    RCR  DH,  CL  ;rotates 3 bits right
    RCR  AX,  CL
```

```
(d) MOV  CL,  6
    RCR  MEM_WORD,  CL  ;rotates 6 times
    RCR  GANDALF_BYTE,  CL
    RCR  BETA [BX] [DI],  CL
```

Flags Affected: CF, OF

Description: RCR (rotate through carry flag right) rotates in EA operand right through the CF flag register by COUNT bits. See also ROR.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

REP/REPZ/REPE/REPNE/REPNZ (Repeat string operation)

Operation: The specified string operation is performed a number of times, i.e., until (CX) becomes 0. CX is decremented by 1 after each iteration.

The compare and scan string operations exit the loop when the zero flag is unequal to the value of bit 0 of this instruction byte.

```
do while (CX) ≠ 0
  service pending interrupt (if any)
  execute primitive string operation in succeeding byte
  (CX) ← (CX) −1
  if primitive operation is CMPS,
    or SCAS and (ZF) ≠ z then exit from while loop
```

Encoding:

1 1 1 1 0 0 1 z

Timing: 6 clocks/loop

Examples:

```
1) REP   MOVS   DEST,   SOURCE   ;see also MOVS

2) REPE   CMPS   DEST,   SOURCE
   ;loop will be exited prior to (CX)=0 only if
   ;(ZF)=1, i.e., only if the byte at (DI) is equal
   ;to the byte at (SI). See also CMPS.

3) REPNZ   SCAS   DEST   ;see also SCAS
   ;only if (ZF)=1, i.e., (AL) = DEST, will
   ;this loop be exited prior to (CX) = 0

4) REPNZ   (nonzero) = REPNE (not equal)
   REPNZ   (zero) = REPE (equal)
```

Flags Affected: See individual string operations.

Description: REP (repeat) causes the succeeding primitive string operation to be performed repeatedly while (CX) is not zero. In the case of CMPS and SCAS, if after any repetition of the primitive operation the ZF flag differs from the "z" bit of the repeat prefix, the repetition is terminated. This prefix may be combined with the segment override and/or LOCK prefixes, although with multiple prefixes, interrupts must be disabled, because the return from an interrupt returns control to the interrupted instruction or to at most one prefix byte before that instruction.

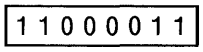
RET (Return from procedure)

Operation: The Instruction Pointer is replaced by the word at the top of the stack (offset of top is in Stack Pointer). SP is incremented by 2. For intersegment returns, the Code Segment register is replaced by the word now at the top of the stack, and SP is again incremented by 2. If an immediate value was specified on the RET statement, that value is now added to SP.

```
(IP) ← ((SP)+1:(SP))
(SP) ← (SP) + 2
if Inter-Segment then
  (CS) ← ((SP)+1:(SP))
  (SP) ← (SP) + 2
if Add Immediate to Stack Pointer then (SP) + data
```

Encoding:

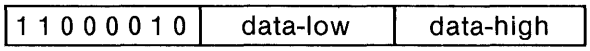
Intra-Segment



Timing: 8 clocks

Example: RET

Intra-Segment and Add Immediate to Stack Pointer:

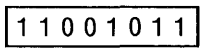


Timing: 12 clocks

Examples:

```
RET  4
RET  12
;these values cause 2 and 6 parameter
;words earlier stored on the stack to be
;discarded. Since most stack operations
;are on words, these values are usually
;even numbers (2 bytes per word).
```

Inter-Segment:



Timing: 18 clocks

RET

Example: RET

Inter-Segment and Add Immediate to Stack Pointer:

1 1 0 0 1 0 1 0	data low	data high

Timing: 17 clocks

Examples:

```
RET  2  ;intersegment returns restore IP first, then CS
RET  8
```

Flags Affected: None

Description: RET transfers control to the return address pushed by a previous CALL operation and optionally adds an immediate constant to the SP register so as to discard stack parameters. If this is an intersegment RET, i.e., it was assembled under a procedure labeled FAR, it will replace the IP AND the CS using the two words at the top of the stack. Otherwise, only the IP is replaced, using only one word from the top of the stack.

When using indirect CALLs, the programmer must carefully ensure that the type of CALL matches the type of RETurn in the procedure, e.g.

```
CALL WORD PTR [BX]
```

must not invoke a FAR procedure and

```
CALL DWORD PTR [BX]
```

must not invoke a NEAR procedure.

See also Appendix D.

ROL (Rotate left)

Operation: The specified destination (leftmost) operand is rotated left COUNT times. Its high order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move up" one position, e.g., the value of the third bit is replaced by the value of the second bit. The vacated bit-position-0 is filled by the new CF, i.e., the old high-order bit.

The rotation continues until the COUNT is exhausted. If COUNT was 1 and the new value of CF is not equal to the new high order bit, then the overflow flag is set; if (CF) does equal that high order bit, OF becomes 0. However, if COUNT was not 1, OF is not defined and has no reliable value.

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (CF)
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 0 0 0 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

Timing (clocks):

(a)	single bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/ bit

Examples:

```
(a) ROL  AH,  1
    ROL  BL,  1
    ROL  CX,  1
    VAL_ONE  EQU  1
    ROL  DX,  VAL_ONE
    ROL  SI,  VAL_ONE

(b) ROL  MEM_BYTE,  1
    ROL  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    ROL  DH,  CL  ;rotates 3 bits left
    ROL  AX,  CL

(d) MOV  CL,  6
    ROL  MEM_WORD,  CL  ;rotates 6 times
    ROL  GANDALF_BYTE,  CL
    ROL  BETA [BX] [DI],  CL
```

ROL

Flags Affected: CF, OF

Description: ROL (rotate left) rotates the operand left by COUNT bits. See also RCL.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

ROR (Rotate right)

Operation: The specified destination (leftmost) operand is rotated right COUNT times. Its low order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move down" one position, e.g., the value of the second bit is replaced by the value of the third bit. The vacated high order position is filled by the new CF, i.e., the old value of position 0.

The rotation continues until the COUNT is exhausted. If COUNT was 1 and the new high order value is not equal to the old high order value, then the overflow flag is set; if they are equal, (OF) = 0. However, if COUNT was not 1 then OF is undefined and has no reliable value.

```
(temp) ← COUNT
DO WHILE (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2
  high-order bit of (EA) ← (CF)
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 0 0 1 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/bit

Examples:

```
(a) ROR   AH,   1
    ROR   BL,   1
    ROR   CX,   1
    VAL_ONE  EQU  1
    ROR   DX,   VAL_ONE
    ROR   SI,   VAL_ONE

(b) ROR   MEM_BYTE,   1
    ROR   ALPHA [DI], VAL_ONE

(c) MOV   CL,   3
    ROR   DH,   CL   ;rotates 3 bits right
    ROR   AX,   CL
```

```
(d) MOV  CL,  6
    ROR  MEM_WORD,  CL  ;rotates 6 times
    ROR  GANDALF_BYTE,  CL
    ROR  BETA [BX] [DI],  CL
```

Flags Affected: CF, OF

Description: ROR (rotate right) rotates the source operand right by COUNT bits. See also RCR.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

SAHF

Operation: The five flags shown are replaced by specified bits from AH, high-order byte of the accumulator:

```
(SF) ← bit 7
(ZF) ← bit 6
(AF) ← bit 4        of AH
(PF) ← bit 2
(CF) ← bit 0

(SF):(ZF):X:(AF):X:(PF):X:(CF) ← (AH)
```

Encoding:

1 0 0 1 1 1 1 0

Timing: 4 clocks

Example: SAHF

Flags Affected: AF, CF, PF, SF, ZF

Description: SAHF transfers specific bits of the AH register to the flag registers SF, ZF, AF, PF, and CF. The bits of AH indicated by "X" in the operation are ignored.

SHL and SAL (Shift logical left and shift arithmetic left)

Operation: The specified destination (leftmost) operand is shifted left COUNT times. Its high order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move up" one position, e.g., the value of the third bit is replaced by the value of the second bit. The vacated low order bit-position is filled by 0.

The rotation continues until the COUNT is exhausted. If COUNT was 1 and the new value of CF is not equal to the new high order bit, then the overflow flag is set; if (CF) does equal that high order bit, OF becomes 0. However, if COUNT was not 1, OF is not defined and has no reliable value.

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2
  (temp) ← (temp)−1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 1 0 0 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/bit

Examples:

```
(a) SHL  AH,  1
    SHL  BL,  1
    SHL  CX,  1
    VAL_ONE  EQU  1
    SHL  DX,  VAL_ONE
    SHL  SI,  VAL_ONE

(b) SHL  MEM_BYTE,  1
    SHL  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    SHL  DH,  CL  ;rotates 3 bits left
    SHL  AX,  CL

(d) MOV  CL,  6
    SHL  MEM_WORD,  CL  ;rotates 6 times
    SHL  GANDALF_BYTE,  CL
    SHL  BETA [BX] [DI],  CL
```

Flags Affected: CF, OF, PF, SF, ZF
Undefined: AF

Description: SHL (shift logical left) and SAL (shift arithmetic left) shift the source operand left by COUNT bits, shifting in low-order zero bits.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

SAR (Shift arithmetic right)

Operation: The specified destination (leftmost) operand is shifted right COUNT times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move down" one position, e.g., the value of the second bit is replaced by the value of the third bit. The vacated high order position retains its old value i.e., if the original high order bit value was 0, zeroes are shifted in. If that value was 1, ones are shifted in.

The shift continues until the COUNT is exhausted. If COUNT was 1 and the high order value is not equal to the next-to-high order value, then the overflow flag is set; if they are equal, (OF) = 0. However, if COUNT was not 1 then OF is reset.

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2, where / is equivalent to signed
    division, rounding down
(temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
    else (OF) ← 0
else (OF) ← 0
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 1 1 1 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/bit

Examples:

```
(a) SAR   AH,  1
    SAR   BL,  1
    SAR   CX,  1
    VAL_ONE  EQU  1
    SAR   DX,  VAL_ONE
    SAR   SI,  VAL_ONE

(b) SAR   MEM_BYTE,  1
    SAR   ALPHA [DI], VAL_ONE

(c) MOV   CL,  3
    SAR   DH,  CL   ;rotates 3 bits right
    SAR   AX,  CL
```

```
(d) MOV  CL,  6
    SAR  MEM_WORD,  CL  ;rotates 6 times
    SAR  GANDALF_BYTE,  CL
    SAR  BETA [BX] [DI],  CL
```

Flags Affected: CF, OF, PF, SF, ZF.
Undefined: AF

Description: SAR (shift arithmetic right) shifts the destination operand right by COUNT bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

SBB (Subtract with borrow)

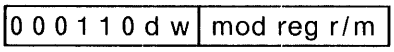
Operation: The source (rightmost) operand is subtracted from the destination (leftmost). If the carry flag was set, 1 is subtracted from the above result. The result replaces the original destination operand.

```
if (CF) = 1 then (DEST) ← (LSRC)−(RSRC)−1
else (DEST) ← (LSRC)−(RSRC)
```

See note.

Encoding:

Memory or Register Operand and Register Operand:



if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST =EA

Timing (clocks):

(a)	register from register	3
(b)	memory from register	9+EA
(c)	register from memory	16+EA

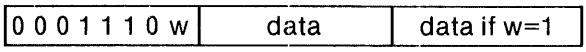
Examples:

```
(a) SBB  AX,  BX
    SBB  CH,  DL

(b) SBB  DX,  MEM_WORD
    SBB  DI,  ALPHA [SI]
    SBB  BL,  MEM_BYTE [DI]

(c) SBB  MEM_WORD,  AX
    SBB  MEM_BYTE [DI],  BX
    SBB  GAMMA [BX] [DI],  SI
```

Immediate Operand from Accumulator:



(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

Timing (clocks): immediate from register 4

Examples:

```
(a) SBB  AL,  4
    VAL_SIXTY  EQU  60
    SBB  AL,  VAL_SIXTY

(b) SBB  AX,  660
    SBB  AX,  VAL_SIXTY * 6
    SBB  ,6606
```

Immediate Operand from Memory or Register Operand:

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate from register 4
(b) immediate from memory 17+EA

Examples:

```
(a) SBB  BX,  2001
    SBB  CL,  VAL_SIXTY
    SBB  SI,  VAL_SIXTY * 9

(b) SBB  MEM_BYTE,  12
    SBB  MEM_BYTE [DI],  VAL_SIXTY
    SBB  MEM_WORD [BX],  79
    SBB  GAMMA  [DI] [BX],  1984
```

If an immediate-data-byte is being subtracted from a register-or-memory word, then that byte is sign-extended to 16 bits prior to the subtraction. For this situation the instruction byte is 83H (i.e., the s:w bits are both set).

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: SBB (subtract with borrow) performs a subtraction of the two source operands, subtracts one if the CF flag is set, and returns the result to one of the operands.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

SCAS/SCASB/SCASW

SCAS (Scan byte string or scan word string)

Chapter 2 describes SCASB and SCASW.

Operation: The string element specified by DI in the Extra Segment is subtracted from the value in the accumulator but the operation affects flags only. The Destination Index is then incremented (if the direction flag is zero) or decremented (if (DF) = 1) by 1 for byte strings or 2 for words.

```
(LSRC)–(RSRC)
if (DF) = 0 then (DI) ← (DI) + DELTA
else (DI) ← (DI)–DELTA
```

Encoding:

1 0 1 0 1 1 1 w

if w = 0 then LSRC = AL, RSRC = (DI), DELTA = 1
else LSRC = AX, RSRC = (DI)+1:(DI), DELTA = 2

Timing: 15 clocks

Examples:

```
1) CLD   ;clears DF, causes DI incrementing
   MOV   DI,  OFFSET  DEST_BYTE_STRING
   MOV   AL,  'M'
   SCAS  DEST_BYTE_STRING

2) STD   ;sets DF, causes DI decrementing
   MOV   DI,  OFFSET  WORD_STRING
   MOV   AX,  'MD'
   SCAS  WORD_STRING
   ;the operand named in the SCAS instruction is used only by the
   ;assembler to verify type and accessibility using current segment
   ;register contents. The actual operation of this instruction uses DI to
   ;point to the location to be scanned, without using the operand
   ;named in the source line.
```

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: SCAS subtracts the destination byte (or word) operand addressed by DI from AL (or AX) and affects the flags but does not return the result. As a repeated operation this provides for scanning for the occurrence of, or departure from, a given value in a string. See also REP.

SHL and SAL (Shift logical left and shift arithmetic left)

Operation: The specified destination (leftmost) operand is shifted left COUNT times. Its high order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move up" one position, e.g., the value of the third bit is replaced by the value of the second bit. The vacated low order bit-position is filled by 0.

The rotation continues until the COUNT is exhausted. If COUNT was 1 and the new value of CF is not equal to the new high order bit, then the overflow flag is set; if (CF) does equal that high order bit, OF becomes 0. However, if COUNT was not 1, OF is not defined and has no reliable value.

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF) then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 1 0 0 r/m

```
if v = 0 then COUNT = 1
else COUNT = (CL)
```

Timing (clocks):

(a)	single-bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/bit

Examples:

```
(a) SHL  AH,  1
    SHL  BL,  1
    SHL  CX,  1
    VAL_ONE  EQU  1
    SHL  DX,  VAL_ONE
    SHL  SI,  VAL_ONE

(b) SHL  MEM_BYTE,  1
    SHL  ALPHA [DI], VAL_ONE

(c) MOV  CL,  3
    SHL  DH,  CL  ;rotates 3 bits left
    SHL  AX,  CL

(d) MOV  CL,  6
    SHL  MEM_WORD,  CL  ;rotates 6 times
    SHL  GANDALF_BYTE,  CL
    SHL  BETA [BX] [DI],  CL
```

SHL

Flags Affected: CF, OF, PF, SF, ZF
Undefined: AF

Description: SHL (shift logical left) and SAL (shift arithmetic left) shift the source operand left by COUNT bits, shifting in low-order zero bits.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

SHR (Shift logical right)

Operation: The specified destination (leftmost) operand is shifted right COUNT times. Its low order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move down" one position, e.g., the value of the second bit is replaced by the value of the third bit. The vacated high order position is filled by 0.

The shift continues until the COUNT is exhausted. If COUNT was 1 and the new high order value is not equal to the next-to-high-order value, then the overflow flag is set; if they are equal, (OF) = 0. However, if COUNT was not 1 then OF is undefined and has no reliable value.

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA) / 2, where / is equivalent to unsigned division
  (temp) ← (temp)–1
if COUNT = 1 then
  if high-order bit of (EA) ≠ next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

See note.

Encoding:

1 1 0 1 0 0 v w	mod 1 0 1 r/m

if v = 0 then COUNT = 1
else COUNT = (CL)

Timing (clocks)

(a)	single-bit register	2
(b)	single-bit memory	15+EA
(c)	variable-bit register	8+4/bit
(d)	variable-bit memory	20+EA+4/bit

Examples:

```
(a) SHR   AH,  1
    SHR   BL,  1
    SHR   CX,  1
    VAL_ONE  EQU  1
    SHR   DX,  VAL_ONE
    SHR   SI,  VAL_ONE

(b) SHR   MEM_BYTE,  1
    SHR   ALPHA [DI], VAL_ONE

(c) MOV   CL,  3
    SHR   DH,  CL   ;rotates 3 bits right
    SHR   AX,  CL

(d) MOV   CL,  6
    SHR   MEM_WORD,  CL   ;rotates 6 times
    SHR   GANDALF_BYTE,  CL
    SHR   BETA [BX] [DI],  CL
```

SHR

Flags Affected: CF, OF, PF, SF, ZF
Undefined: AF

Description: SHR (shift logical right) shifts the source operand right by COUNT bits, shifting in high-order zero bits.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

STC (Set carry flag)

Operation: The carry flag is set to 1.

(CF) ← 1

Encoding:

1	1	1	1	1	0	0	1

Timing: 2 clocks

Example: STC

Flags Affected: CF

Description: STC sets the CF flag.

STD (Set direction flag)

Operation: The direction flag is set to 1.

(DF) ← 1

Encoding:

1	1	1	1	1	1	0	1

Timing: 2 clocks

Example: STD ;causes decrementing of DI (and SI) in string operations.

Flags Affected: DF.

Description: STD sets the DF flag, causing the string operations to auto-decrement the operand index(es).

STI (Set interrupt flag)

Operation: The interrupt flag is set to 1.

(IF) ← 1

Encoding:

1 1 1 1 1 0 1 1

Timing: 2 clocks

Example: STI ;enables interrupts

Flags Affected: IF

Description: STI sets the IF flag, enabling maskable external interrupts after the execution of the next instruction.

STOS/STOSB/STOSW

STOS (Store byte string or store word string)

Chapter 2 describes STOSB and STOSW.

Operation: The byte (or word) in AL (or AX) replaces the contents of the byte (or word) pointed to by DI in the Extra Segment. DI is then incremented if the direction flag is zero or decremented if DF=1. The change is 1 for bytes, 2 for words.

```
(DEST) ← (SRC)
if (DF) = 0 then (DI) ← (DI) + DELTA
else (DI) ← (DI)−DELTA
```

Encoding:

1 0 1 0 1 0 1 w

if w = 0 then SRC = AL, DEST = (DI), DELTA = 1
else SRC = AX, DEST = (DI) + 1:(DI), DELTA = 2

Timing: 10 clocks

Examples:

```
1) MOV  DI,  OFFSET  BYTE_DEST_STRING
   STOS  BYTE_DEST_STRING

2) MOV    DI ,OFFSET  WORD_DEST
   STOS    WORD_DEST
```

Flags Affected: None

Description: STOS transfers a byte (or word) operand from AL (or AX) to the destination operand addressed by DI and adjusts the DI register by DELTA. As a repeated operation (see REP) this provides for filling a string with a given value. The operand named in the STOS instruction is used only by the assembler to verify type and accessibility using current segment register contents. The actual operation of the instruction uses only DI to point to the location being stored into.

SUB (Subtract)

Operation: The source (rightmost) operand is subtracted from the destination (leftmost) operand and the result is stored in the destination.

(DEST) ← (LSRC)−(RSRC)

See note.

Encoding:

Memory or Register Operand and Register Operand:

0 0 1 0 1 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks):

(a)	register from register	3
(b)	memory from register	9+EA
(c)	register from memory	16+EA

Examples:

```
(a) SUB  AX,  BX
    SUB  CH,  DL

(b) SUB  DX,  MEM_WORD
    SUB  DI,  ALPHA [SI]
    SUB  BL,  MEM_BYTE [DI]

(c) SUB  MEM_WORD,  AX
    SUB  MEM_BYTE [DI],  BL
    SUB  GAMMA  [BX] [DI],  SI
```

Immediate Operand from Accumulator:

0 0 1 0 1 1 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data, DEST = AL
(b) else LSRC = AX, RSRC = data, DEST = AX

Timing (clocks): immediate from register 4

Examples:

```
(a) SUB  AL,  4
    VAL_SIXTY  EQU  60
    SUB  AL,  VAL_SIXTY

(b) SUB  AX,  660
    SUB  AX,  VAL_SIXTY * 6
    SUB  ,6606
```

SUB

Immediate Operand from Memory or Register Operand:

1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s:w=01

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): (a) immediate from register 4
(b) immediate from memory 17+EA

Examples:

```
(a) SUB  BX,  2001
    SUB  CL,VAL_SIXTY
    SUB  SI,  VAL_SIXTY * 9

(b) SUB  MEM_BYTE,  12
    SUB  MEM_BYTE [DI],  VAL_SIXTY
    SUB  MEM_WORD [BX],  79
    SUB  GAMMA [DI] [BX],  1984
```

If an immediate-data-byte is being subtracted from a register-or-memory word, then that byte is sign-extended to 16 bits prior to the subtraction. For this situation the instruction byte is 83H (i.e., the s:w bits are both set).

Flags Affected: AF, CF, OF, PF, SF, ZF

Description: SUB performs a subtraction of the source (rightmost) operand from the destination, and returns the result to the ddestination operand.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

TEST (Test, or logical compare)

Operation: The 2 operands are ANDed to affect the flags but neither operand is changed. The carry and overflow flags are reset.

(LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

See note.

Encoding:

Memory or Register Operand with Register Operand:

1 0 0 0 0 1 0 w	mod reg r/m

LSRC = REG, RSRC = EA

Timing (clocks): (a) register with register 3
(b) register with memory 9 + EA

Examples:

```
(a) TEST  AX,  DX
    TEST  ,DX  ;same as above
    TEST  SI,  BP
    TEST  BH,  CL

(b) TEST  MEM_WORD,  SI
    TEST  MEM_BYTE,  CH
    TEST  ALPHA [DI],  DX
    TEST  BETA [BX] [SI],  CX
    TEST  DI,  MEM_WORD
    TEST  CH,  MEM_BYTE
    TEST  AX,  GAMMA [BP] [SI]
```

Immediate Operand with Accumulator:

1 0 1 0 1 0 0 w	data	data if w=1

(a) if w = 0 then LSRC = AL, RSRC = data
(b) else LSRC = AX, RSRC = data

Timing (clocks): immediate with register 4

Examples:

```
TEST  AL,  6
TEST  AL,  IMM_VALUE_DRIVE11
TEST  AX,  IMM_VAL_909
TEST  ,999
TEST  AX,  999  ;same as above
```

TEST

Immediate Operand with Memory or Register Operand:

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w=1

LSRC = EA, RSRC = data

Timing (clocks): (a) immediate with register 4
(b) immediate with memory 10 + EA

Examples:

```
(a) TEST  BH,  7
    TEST  CL,  19_IMM_BYTE
    TEST  DX,  IMM_DATA_WORD
    TEST  SI,  798

(b) TEST  MEM_WORD,  IMM_DATA_BYTE
    TEST  GAMMA [BX],  IMM_BYTE
    TEST  [BP] [DI],  6ACEH
```

Flags Affected: CF, OF, PF, SF, ZF.
Undefined: AF

Description: TEST performs the bitwise logical conjuntion of the two source operands, causing the flags to be affected, but does not return the result.

The source (rightmost) operand must usually be of the same type, i.e., byte or word, as the destination operand. The only exception for TEST is testing an immediate-data byte with a memory word.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

WAIT (Wait)

Operation: None

Encoding:

1 0 0 1 1 0 1 1

Timing: 3 clocks

Example: WAIT

Flags Affected: None

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 reentered. The wait state is cleared and execution resumed when the TEST signal is asserted. Execution resumes without allowing external interrupts until after the execution of the next instruction. The instruction allows the processor to synchronize itself with external hardware.

XCHG (Exchange)

There are 2 forms of the XCHG instruction, one for switching the contents of the accumulator with those of some other general word register, and one for switching a register and a memory-or-register operand.

See note.

Operation:

1) The contents of the destination (leftmost operand) are temporarily stored in an internal work register

   (Temp) ← (DEST)

2) The contents of the destination are replaced by the contents of the source (leftmost) operand

   (DEST) ← (SRC)

3) The former contents of the destination are moved from the work register into the source operand

   (SRC) ← (Temp)

Flags Affected: None

Type 1:

Register Operand with Accumulator:

1 0 0 1 0 reg

SRC = REG, DEST = AX

Timing: 3 clocks

Examples:

```
XCHG  AX,  BX
XCHG  SI,  AX
XCHG  CX,  AX
```

Type 2:

Memory or Register Operand with Register Operand:

1 0 0 0 0 1 1 w	mod reg r/m

SRC = EA, DEST = REG

Timing (clocks):

memory with register		17 + EA
register with memory		4

Examples:

```
XCHG  BETA_WORD,  CX
XCHG  BX,  DELTA_WORD
XCHG  DH,  ALPHA_BYTE
XCHG  BL,  AL
```

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

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

XLAT/XLATB

XLAT (Translate)

Operation: The contents of the accumulator are replaced by a byte from a table. The table's starting address has been moved into register BX. The original contents of AL is the number of bytes past that starting address, where the desired translation byte is to be found. It replaces the contents of AL.

(AL) ← ((BX) + (AL))

Encoding:

1 1 0 1 0 1 1 1

Timing: 11 clocks

Example:
```
MOV   BX,  OFFSET  TABLE_NAME
XLAT  TABLE_ENTRY
;(see also example at LODS)
```

Flags Affected: None

Description: XLAT performs a table lookup byte translation. The AL register is used as an index into a table (256-bytes at most) addressed by the BX register. The byte operand so addressed is transferred to AL.

XOR (Exclusive or)

Operation: Each bit position in the destination (leftmost) operand is set to zero if the corresponding bit positions in both operands were equal. If they were unequal then that bit position is set to 1.

(DEST) ← (LSRC)‖(RSRC)
(CF) ← 0
(OF) ← 0

See note.

Encoding:

0 0 1 1 0 0 d w	mod reg r/m

if d = 1 then LSRC = REG, RSRC = EA, DEST = REG
else LSRC = EA, RSRC = REG, DEST = EA

Timing (clocks): (a) register to register 3
(b) memory to register 9 + EA
(c) register to memory 16 + EA

Examples:

```
(a) XOR  AH,  BL     ;result in AH,  BL unchanged
    XOR  SI,  DX     ;result in SI,  DX unchanged
    XOR  CX,  DI     ;result in CX,  DI unchanged

(b) XOR  AX,  MEM_WORD
    XOR  CL,  MEM_BYTE [SI]
    XOR  SI,  ALPHA [BX] [SI]

(c) XOR  BETA [BX] [DI],  AX
    XOR  MEM_BYTE,  DH
    XOR  GAMMA [DI],  BX
```

Immediate Operand to Accumulator:

0 0 1 1 0 1 0 w	data	data if w=1

if w = 0 then LSRC = AL, RSRC = data, DEST = AL
else LSRC = AX, RSRC = data, DEST = AX

Timing (clocks): immediate to register 4

Examples:

```
a) XOR  AL,  11110110B
   XOR  AL,  0F6H

b) XOR  AX,  23F6H
   XOR  AX,  75Q
   XOR  ,23F6H   ;AX destination
```

XOR

Immediate Operand to Memory or Register Operand:

1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w=1

LSRC = EA, RSRC = data, DEST = EA

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Examples:

a) XOR AH, 0F6H
XOR CL, 37
XOR DI, 23F5H

b) XOR MEM_BYTE, 3DH
XOR GAMMA [BX] [DI], 0FACEH
XOR ALPHA [DI], VAL_EQUD_33H

Flags Affected: CF, OF, PF, SF, ZF.
Undefined: AF

Description: XOR (exclusive Or) performs the bitwise logical exclusive disjunction of the source operands and returns the result to the destination operand.

NOTE: The early pages of this Chapter explain mod, reg, r/m, EA, DEST, and other notation. Included there are tables showing the encoding of addressing modes and registers. In most cases reg is a 3-bit field allowing any register except segment registers. Instructions referring directly to segment registers use a 2-bit field.

All references to memory implicitly use a segment register in constructing an address (see Chapter 2). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source line and on the ASSUME directive which applies to that line. See Chapter 2.

Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address. If a byte or word of immediate-data is also used, it will follow that displacement.

INDEX TO INSTRUCTION MNEMONICS

CHAPTER 6
CODEMACROS INTRODUCTION

This chapter describes codemacros, which define 8086 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, e.g.,:

```
        •
        •
        •
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 instruction above are codemacros. The assembler's entire instruction set is defined and implemented as a large number of codemacros. (The definitions are in APPENDIX A). Once you understand how this is done, you may add instructions, or even replace 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,

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 formals, each in the form

```
form_name:specifier_letter [modifier_letter] [range]
```

These 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 formals 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, or w. There follows an optional range specifier, which consists of a pair of parentheses enclosing either one number, or two numbers 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 macro, 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

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, e.g.,:

```
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. M meaning a memory address. This can be either a variable (with or without indexing) or a bracketed register expression.
6. R meaning a general Register only, not an address-expression, not a register in brackets, and not a segment register.
7. S meaning a Segment register only, either CS, DS, ES, or SS.
8. 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.
- 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. The specifier-modifier pair "Dd" is 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. 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 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.

The bit patterns for the override bytes are as follows:

```
00100110 for the ES override
00101110 for the CS override
00110110 for the SS override
00111110 for the DS override
```

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 imperative 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.

As 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 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, e.g.,

```
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
10000011  10000101  00000001  00000010
```

Example 4:

```
ADD BYTE PTR [BX] [SI], 4
10000001  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.

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.

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 = 00111001B
    opcode.MID3 = (000)00111
    R53 <11011B,opcode.mid3> becomes 1101 1111B
    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 1111   (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 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.

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 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.

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 matched "w".
Type DWORD matches "d".
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".

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. Resolution of forward references might cause a different codemacro to be matched.
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 word)
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              → 2
ADD  WORD_VAR,255            → 3
```

NOTE

Codemacros are limited to a maximum of 128 internal bytes, which is reached at approximately 60 bytes of generated object code.

CHAPTER 7
MACRO PROCESSING LANGUAGE (MPL)

This chapter describes MPL, the MCS-86 Macro Processing Language. (Macros should not be confused with codemacros, which pertain to individual machine instructions, and are discussed in Chapter 6.) Appendix L presents a more rigorous treatment of MPL.

MPL extends the MCS-86 Assembly Language to include these capabilities:

- Macro definition and invocation
- Macro-time string manipulation
- Macro-time expression evaluation
- Conditional assembly
- Macro-time console I/O

Conceptual Overview of Macro Processing

Understanding macro processing requires a different perspective from the way assembly languages and high-level procedural languages are understood as treating source files. When you invoke ASM86 to assemble your source file, all MPL statements in your source file are evaluated before the actual assembly process starts. Your MPL statements are either function definitions or function calls. The functions can be MPL's built-in functions or your own user-defined functions. You use the MPL built-in function DEFINE to define your own functions.

MPL deals in strings. If you think of your source file as one long string, then its MPL statements (function definitions and function calls) are substrings of that one long string. MPL replaces function definitions with the null string, and each function call with its value, which is always a string, and may be the null string. Similarly, any arguments present in function calls are given as strings, and may be interpreted by the function (depending on its definition) as integer values. Thus, depending on its context, the expression "86H" could represent the 3-character string '86H' or the 17-bit value +0000 0000 1000 0110B.

The following scheme illustrates these concepts:

1. Your source file as seen by the Macro Processor:
   (---------plaintext---------(macro-def)---------(macro-call)----------plaintext---------)
2. An internal, intermediate form after the macro-definition is stored:
   (---------plaintext----------o----------(macro-call)----------plaintext----------)
   Where 'o' represents the null string and 'macro-call' contains '86H'.
3. The macro called may then consider '86H' as a string or an integer value:
   (----------plaintext--------------------(86H)--------------------plaintext----------)
   (---------plaintext-----------------------(+0000 0000 1000 0110B)-----plaintext-----)
4. The resulting macro expansion then becomes input to the assembler-proper:
   (----------plaintext-- ----------)

What Is Macro-Time?

Macro-time is the term given to the time-frame within which the macro processor acts on your source file, copying it to an intermediate form for assembly, and processing your macro definitions and macro calls. No object code is created during macro-time. Macro-time is followed by:

- Assembly-time, when object code is created
- Link-time, when external references are resolved
- Locate-time, when code and data are bound to absolute addresses
- Run-time, when your program executes

Since MPL allows you to generate virtually any character string, which will then be assembled, linked, located, and run, macro processing influences the entire development cycle of your program. However, since the macro processor itself produces no object code, it cannot interrogate the assembly-time status of your program (such as referencing the assembler-proper's symbol table).

What Is a Macro?

A macro is a shorthand notation for a source text string. The shorthand notation is the macro name; the string it represents is the macro value. You define your own macros using the MPL function DEFINE, which has the format:

%*DEFINE (macro-call) (replacement-pattern)

Macro Expansions and Side Effects

A careful distinction must be made between the value of a macro or built-in function and its side-effects. At call-time, when the macro or built-in function is called, the macro processor replaces the call with the value (an ASCII string) of the macro or built-in function, as well as performing the operations inherent in the macro or built-in function.

The value of the DEFINE built-in function is the null (empty) string; therefore, when the call to DEFINE is made to define your user macro, the call is replaced by the null string. That is, the call is not copied from your source file to the intermediate file. The significance of the call to DEFINE is not its value, but its side-effect; that is, defining your user macro (entering it in the macro symbol table).

If, for example, you are coding a program which contains several calls to a procedure SUBROUTINE, and you want to push/pop registers ES, DS, AX, CX, DX, BX, SI, and DI before/after the call, you could first define the macro CALLSUBROUTINE as follows:

```
%*DEFINE (CALLSUBROUTINE) (
            PUSH ES
            PUSH DS
            PUSH AX
            PUSH CX
            PUSH DX
            PUSH BX
            PUSH SI
            PUSH DI
            CALL SUBROUTINE
            POP DI
            POP SI
            POP BX
            POP DX
            POP CX
            POP AX
            POP DS
            POP ES
)
```

Now wherever the macro call %CALLSUBROUTINE appears in your source file, the macro processor replaces it with the defined character string, including all carriage-returns, line-feeds, tabs, and blanks.

Two remarks are in order:

1. The definition of the macro begins with "%*DEFINE". (The asterisk (*) is termed the "call-literally" character, and means that no macro expansion is requested at this time. The macro processor is said to be in *literal* mode.)
2. Opening and closing carriage-return-line-feed's are included *inside* the replacement-pattern part of this macro definition so that the source file passed to the assembler-proper will not contain run-on lines.

What Is Macro Processing?

The macro processor, which is part of the MCS-86 Macro Assembler, copies your source file to an intermediate file to be assembled. During the copying process, the macro processor examines each character of your source file for a distinguished character called the metacharacter, which can be any ASCII character, but by default is the percent-sign (%). When the metacharacter is detected, the macroprocessor knows that what follows is intended for macro processing.

The metacharacter signals the macro processor that what follows is:

- A user macro definition, such as:

```
%*DEFINE (AR(NAME, TYPE, LENGTH) (%NAME D%TYPE %LENGTH DUP(?)
)
```

  This defines a macro AR with three parameters (NAME, TYPE, LENGTH), which, when called with actual arguments (strings or function calls which evaluate to strings) , expands to an assembly-language DB, DW, or DD directive defining an array with %LENGTH units (bytes, words, or doublewords) having the name %NAME. Notice that parameters are listed in the macro-name part of the definition without metacharacters, but in the replacement-pattern part of the definition each parameter is prefixed by the metacharacter %. Notice also

that the carriage-return (following DUP(?) is meant to be part of the macro definition, since we want the data definition directive to be on a line by itself.

- A user macro invocation (call), such as:

```
%AR(LASZLO, W, 500)
```

This call is replaced by its value, which according to the preceding definition is the following string, including the terminating carriage-return (and line-feed):

```
LASZLO DW 500 DUP(?)
```

Similarly, the call:

```
%AR(GONZO, B, 2048)
```

expands to:

```
GONZO DB 2048 DUP(?)
```

including the final carriage-return-line-feed.

- A user call to an MPL built-in function, such as:

```
%IF (%EQS(%ANSWER,YES)) THEN (%AR(LASZLO, W, 500)) FI
```

This call to the MPL built-in function IF evaluates to the first array definition above if the value of ANSWER (a user-defined function, presumably incorporating the MPL built-in functions IN and OUT) is exactly equal to the string 'YES', and evaluates to the null (empty) string otherwise.

These three types of MPL statements result in the respective actions:

1. If a macro definition follows, the macro processor saves the definition.
2. If a macro call follows, the macro processor retrieves the definition of the called macro, computes the value (an ASCII string) of the macro based on the call, and places it in the intermediate file at the point of call. This is called expanding the macro.
3. If a call to an MPL built-in function follows, the macro processor replaces the call with the value of the built-in function, much the same as in the previous case. Calls to MPL built-in functions will be discussed later; however, this section describes one such MPL built-in function—DEFINE, which you call to define your macros. Strictly speaking, then, the first item on the above list is really a special case of the third.

Aside from macro definitions and calls, the text of your source file has no meaning to the macro processor. The macro processor forms the "front-end" of the assembler, and as such, it cannot detect errors in your 8086 assembly language directives or instructions.

Why Use Macros?

Since a macro defines a string of text (called the macro value) that will replace a macro call, the usefulness of a macro depends on three characteristics:

- Its ability to represent a string of text using a shorter string
- Its ability to be used in different contexts; in a word, its flexibility
- Its side-effects; for instance, DEFINE, MATCH, and OUT.

The example CALLSUBROUTINE above has the first characteristic, but not the second; CALLSUBROUTINE is a "constant" macro—its value never changes, unless you redefine it. You can redefine your macros (but not MPL's built-in functions) any time you want (with the exception that a macro definition may not modify itself). At call-time, the macro processor refers to the most recent definition of each user macro.

Parameters and Arguments

A macro can also be defined so that part of it varies, depending on the information supplied to the macro in the form of arguments.

Returning to a previous example of the procedure call to SUBROUTINE, preceded by multiple PUSHes and followed by multiple POPs, we see that the macro CALLSUBROUTINE as defined has limited usefulness—we cannot use it for calls to other procedures besides ROUTINE.

We can code a macro to specify the same sequence of PUSHes, a call to any procedure (not just ROUTINE), and the same sequence of POPs, as follows:

```
%*DEFINE ( CALLSUB(ROUTINE) ) (
            PUSH ES
            PUSH DS
            PUSH AX
            PUSH CX
            PUSH DX
            PUSH BX
            PUSH SI
            PUSH DI
            CALL %ROUTINE
            POP DI
            POP SI
            POP BX
            POP DX
            POP CX
            POP AX
            POP DS
            POP ES
)
```

Now to generate a call to procedure AXOLOTL, for example (together with the preceding PUSHes and following POPs, as well as carriage-returns, line-feeds, tabs, and blanks), all you need to code is:

```
%CALLSUB(AXOLOTL)
```

In this example, ROUTINE is called a formal parameter, or simply a parameter. (It is also known as a "dummy" parameter, since its name in the definition of CALLSUB is irrelevant.)

When CALLSUB is called with a value for the formal parameter (ROUTINE), the actual value (AXOLOTL) is referred to as an argument.

In short, the parameter ROUTINE acts as a place-holder for the argument AXOLOTL.

In using macro definitions that have parameter lists, and corresponding macro calls that have argument lists:

- The parameter list of a macro definition is enclosed in parentheses following the macro name; parameters are separated by commas, as in:

```
%*DEFINE (BIGMAC(P1,P2,P3,P4,P5)) (text-string using %P1, %P2, %P3, %P4, %P5)
```

  When a parameter (to be replaced by an argument at call-time) appears in the replacement-string of the definition, be sure to prefix the metacharacter (%) to it.

- The argument list of a macro call is enclosed in parentheses following the macro name; arguments are separated by commas, as in:

```
%BIGMAC(CATSUP,MUSTARD,ONION,PICKLE,LETTUCE)
```

- The only occurrence of the metacharacter in the macro call is that prefixed to the macro-name, unless one or more arguments are macros. If you use a macro as an argument, then you prefix the metacharacter to the argument as well. For instance, if the macro YELLOWSTUFF is defined:

```
%*DEFINE (YELLOWSTUFF) (MUSTARD)
```

  Then you could call BIGMAC as follows:

```
%BIGMAC(CATSUP,%YELLOWSTUFF,ONION,PICKLE,LETTUCE)
```

  and obtain the same macro expansion.
- You can use any number of parameters/arguments.

This chapter describes a subset of MPL in which commas delimit parameters/arguments. More general constructs are possible, as described in Appendix L, Macro Processor Language: Full Capabilities

Evaluation of the Macro Call

The macro processor evaluates the call %CALLSUB(AXOLOTL) as follows:

1. The macro processor recognizes the metacharacter (%), and momentarily suspends copying your source file while it looks up the definition of CALLSUB in its macro symbol table.
2. Finding CALLSUB in the symbol table, the macro processor sees that CALLSUB is defined using one parameter, and hence needs one user-supplied argument in order to be expanded.
3. Upon finding the string 'AXOLOTL' in parentheses immediately following the %CALLSUB macro call, the macro processor picks up 'AXOLOTL' as the argument to the macro call.

4. Then, using the definition of CALLSUB as the string of PUSHes, POPs, the CALL, and all carriage-returns, line-feeds, tabs, and blanks in the definition, the macro processor computes the value of the call %CALLSUB(AXOLOTL) to be the ASCII string:

```
PUSH ES
PUSH DS
PUSH AX
PUSH CX
PUSH DX
PUSH BX
PUSH SI
PUSH DI
CALL AXOLOTL
POP DI
POP SI
POP BX
POP DX
POP CX
POP AX
POP DS
POP ES
```

5. The macro processor replaces the macro call with the value of the macro, exactly at the point of call.

A Comment-Generating Macro

Macro definitions and calls can be placed anywhere in your source file: as constant character strings (the first example), as operands to instructions (the second example), as in-line routines (the example following the next), as arguments to function calls, or simply as character strings that are more easily defined as macro functions and called as needed than rekeyed each time.

Consider this comment-generating macro, HEADER, which accepts 5 arguments, and is defined as follows:

```
%*DEFINE (HEADER(ROUTINE,DATE,NAME,PURPOSE,REGCLOB)) (
;*****************************************************************
;* ROUTINE NAME: %ROUTINE
;* DATE: %DATE
;* PROGRAMMER'S NAME: %NAME
;* PURPOSE OF ROUTINE: %PURPOSE
;* REGISTERS CLOBBERED: %REGCLOB
;*****************************************************************
)
```

Note that in the macro definition of HEADER above:

- The definition begins with %*DEFINE. This informs the macro processor that no expansion is to take place. (That is, this is a definition.)
- In the DEFINE function's pattern for parameterized macro definitions:

  %*DEFINE (macro-name(parameter-list)) (replacement-pattern)

- The metacharacter (%) does not appear in the macro-name or parameter-list fields.

- The metacharacter (%) does appear as a prefix to parameter names in the replacement-pattern, since the macro processor needs to know that the first 'ROUTINE' in 'ROUTINE NAME: %ROUTINE' is not itself a macro call, but the second is.
- The "hanging" left parenthesis at the right in the first line denotes that the macro body begins with a carriage-return. (Otherwise, the expanded macro might start in the middle of a line.) Similarly, the lone right-parenthesis which terminates the replacement-pattern denotes that the macro body ends with a carriage-return.

The macro call:

```
%HEADER(LASZLO,5/15/79,G. BOOLE,UPDATE NETWORK STRUCTURES,AX/SI/DI)
```

results in the expansion:

```
;*****************************************************************
;* ROUTINE NAME: LASZLO
;* DATE: 5/15/79
;* PROGRAMMER'S NAME: G. BOOLE
;* PURPOSE OF ROUTINE: UPDATE NETWORK STRUCTURES
;* REGISTERS CLOBBERED: AX/SI/DI
;*****************************************************************
```

A Macro to Move Word Strings at Run-Time

You can use macros for routines.For instance, your source file might require these three variants of the same code to move a word-string from a DS-based segment to an ES-based segment:

1. Move 5 Words from TABLE to FIELD

```
MOV CX,5            ;Register CX contains number of words to move.
LEA SI, TABLE       ;1st word to be moved is at DS:TABLE.
LEA DI, FIELD       ;1st word to be moved to ES:FIELD.
REP MOVSW           ;Loop here while CX decrements to 0.
```

2. Move LENGTH Words from ARRAY[BX] to ADTAB+8

```
MOV CX,LENGTH       ;Reg. CX contains number of words to move.
LEA SI, ARRAY[BX]   ;1st word to move is at DS:ARRAY[BX].
LEA DI, ADTAB+8     ;1st word to move to ES:ADTAB+8.
REP MOVSW           ;Loop here while CX>0.
```

3. Move AX Words from STRUC.WDS to [BX]

```
MOV CX,AX           ;Move count to CX.
LEA SI, STRUC.WDS   ;1st word to move is at DS:STRUC.WDS.
LEA DI, [BX]        ;1st word to move to ES:[BX].
REP MOVSW           ;Move a word at a time until CX=0.
```

By parameterizing the three operand fields that differ in these text-strings, we obtain the replacement-pattern of the macro we need to generate all three instances:

```
MOV CX, COUNT
LEA SI, SOURCE
LEA DI, DEST
REP MOVSW
```

Using the MPL built-in function DEFINE, we can name a macro representing the common form of the three separate instances:

```
%*DEFINE (MOVE(COUNT, SOURCE, DEST)) (
        MOV CX, %COUNT
        LEA SI, %SOURCE
        LEA DI, %DEST
        REP MOVSW
)
```

Note that in this macro definition, which conforms to the pattern for the DEFINE function:

%*DEFINE (macro-name) (replacement-pattern)

1. The metacharacter (%) and the call-literally character (*) are prefixed to DEFINE.

2. Neither the metacharacter (%) nor the call-literally character (*) occur in the macro-name field, but that

3. The metacharacter (%) is prefixed to each parameter-name in the replacement-pattern. The call-literally character does not appear in the replacement-pattern.

4. The replacement-pattern is defined by its appearance between the second pair of parentheses in the pattern:

```
%*DEFINE (macro-name) (replacement-pattern)
```

This means that MOVE consists of the opening and closing carriage-returns given in its replacement-pattern, as well as the text between them. Without these opening and closing carriage-returns, the first and last lines of the expanded macro would be run together with the last line before, and the first line after, the macro call.

Calling MOVE with Actual Arguments

Now with the MOVE macro defined for this assembly, it is unnecessary to code the sequence of instructions over again every time we wish to move a word-string. Our user macro MOVE can be invoked (called) using actual arguments in place of the formal parameters COUNT, SOURCE, and DEST. The formal parameters are simply place-holders until you supply actual values as arguments in macro calls.

For example, the macro calls:

```
%MOVE(5, TABLE, FIELD)

%MOVE(LENGTH, ARRAY[BX], ADTAB+8)

%MOVE(AX, STRUC.WDS, [BX])
```

expand to (1), (2), and (3) above, respectively.

A Macro to Move Both Byte- and Word-Strings

By introducing one additional parameter into the definition, we can generalize the MOVE macro (which moves word-strings only) to the MOVER macro (which moves both byte- and word-strings):

```
%*DEFINE MOVER(COUNT, SOURCE, DEST, TYPE)) (
        MOV CX, %COUNT
        LEA SI, %SOURCE
        LEA DI, %DEST
        REP MOVS%TYPE
)
```

Now the call:

```
%MOVER(100, TABLE, FIELD, W)
```

moves 100 words from DS:TABLE to ES:FIELD, since a MOVSW is generated in the expansion.

However, the call:

```
MOVER(100, TABLE, FIELD, B)
```

moves 100 bytes from DS:TABLE to ES:FIELD, since a MOVSB is generated in the expansion.

MPL Identifiers

MPL identifiers, used for function and parameter names, are different from your assembly-language identifiers. An MPL identifier has the following characteristics:

1. The first character must be an alphabetic character A through Z. Upper- and lower-case alphabetic characters are not distinguished.
2. Successive characters may be alphabetic, numeric (0 through 9), or the underscore (_) character, sometimes called the "break" character, and represented on some keyboards by a back-arrow. Its ASCII value is 5FH.
3. As with the assembly-language proper, identifiers may be any length but are considered unique only up to 31 characters.

Numbers As Strings in MPL

MPL maps ASCII strings in your source file into ASCII strings in an intermediate file to be assembled.

For instance, the MPL built-in function LEN accepts a string argument (or a macro whose value is a string), and has the string value 'xyH', where x and y are hexadecimal digits giving the length of the argument string.

Thus, the value of %LEN(ABC) is the ASCII string 03H. Similarly, the value of %LEN(ABCDEFGHIJ) is the ASCII string 0AH.

Furthermore, like other MPL built-in functions and user macros, LEN can accept a macro as an argument. In this case, the value of LEN is an ASCII string representing the length of the macro value string.

If, for example, ALPHA and DECIMAL are defined as follows:

```
%*DEFINE (ALPHA) (ABCDEFGHIJKLMNOPQRSTUVWXYZ)

%*DEFINE (DECIMAL) (0123456789)
```

then it follows that %LEN(%ALPHA) has the value 1AH, and %LEN(%DECIMAL) has the value 0AH. Note that %LEN(ALPHA) and %LEN(DECIMAL) are still meaningful, and have the values 05H and 07H, respectively.

Expression Evaluation; the EVAL Built-in Function

Since MPL deals in strings, the macro processor does not normally attempt to evaluate strings expressing numeric quantities. (Exceptions to this general rule are the built-in functions REPEAT, IF, WHILE, and SUBSTR, decribed below).

Thus, if you code:

```
%LEN(%ALPHA) + %LEN(%DIGIT)
```

the macro processor will treat the expression as a string, and will replace it with:

```
1AH + 0AH
```

without processing it any further.

If you want an expression to be evaluated, you can use the MPL built-in function EVAL function, which takes the form:

%EVAL(expression)

In this case, the desired evaluation is performed, and an ASCII string of hexadecimal digits is returned as the value of EVAL. For the example, we have:

```
%EVAL(%LEN(%ALPHA) + %LEN(%DIGIT))
```

which first reduces to:

```
%EVAL(1AH + 0AH)
```

and is then evaluated as an arithmetic expression to obtain the string:

```
24H
```

as the value of the call.

Arithmetic Expressions

Arithmetic operations are 17-bit, as used by the assembler proper. Note that dyadic (two-argument) operators are infix (as assembler-proper operators), unlike MPL's outfix operators, and that infix operators do not require the metacharacter preceding a call:

```
Infix:    %VALUE1 EQ 3          (compare numbers)
Outfix:   %EQS(%VALUE1, 3)      (compare strings)
```

Arithmetic expressions allow the following operators, in high-to-low order of precedence:

```
Parenthesized Expressions
HIGH,    LOW
Multiplication and Division: *, /, MOD, SHL, SHR
Addition and Subtraction: +, – (both unary and binary)
Relational: EQ, LT, LE, GT, GE, NE
Logical NOT
Logical AND
Logical OR,  XOR
```

Expressions are evaluated left-to-right, with operations of higher precedence performed first, unless precedence is overridden using parentheses.

Examples can be found at the end of Chapter 4. It is essential to remember that these arithmetic, relational, and Boolean operators are identical to the assembly-language operators of the same names. The difference between using these operators in the MPL context as opposed to the usual assembly-language context is that:

1. For the operations to be performed, MPL expressions must be enclosed within an %EVAL(expression) call.
2. Although the value returned by EVAL is always an ASCII string of hexadecimal digits, and not a "pure number", the hexadecimal string itself can be used as a number with arithmetic operators.
3. Assembly-time symbols such as those defined by EQU are *not* available at macro-time, and cannot be included in an argument to EVAL.

Range of Values

The permissible range of value is −0FFFFH (−65535) to 0FFFFH (65535).

The Length Function (LEN)

The MPL built-in LEN is called as follows:

%LEN(string)

and returns as a hexadecimal value the number of characters in the argument string.

For example, %LEN(A,B,C) = 5. %LEN(ABC) = 3.

If ABC has been defined, as in:

```
%*DEFINE (ABC) (ABRACADAVER)
```

then you would request the length ('0BH') of that string by calling as follows:

```
%LEN(%ABC)
```

The string 0BH would then replace the call to LEN, and would be interpreted as a number (by arithmetic operators) or a string (by MPL operators or functions).

String Comparator (Lexical-Relational) Functions

The string comparator functions are:

MPL Function	Answers the Question	With One Of
EQS	Are the strings lexically equal?	−1H(Yes), 00H (No)
NES	Are the strings lexically unequal?	−1H(Yes), 00H (No)
LTS	Does the first precede the second in their dictionary ordering?	−1H(Yes) 00H (No)
LES	Does the first precede or equal the second in their dictionary ordering?	−1H (Yes) 00H (No)
GES	Does the first follow or equal the second in their dictionary ordering?	−1H (Yes) 00H (No)
GTS	Does the first follow the second in their dictionary ordering?	−1H (Yes) 00H (No)

The value returned (−1H or 00H) is a character string, and not a "pure number".

Thus, the function call:

```
%LTS(101,101B)
```

returns the string '−1H', or "True", because the string '101' precedes the string '101B' in the lexical sense.

And the function call:

```
%EQS(0AH, 10)
```

returns the string '00H', or "False", because the two strings are not equal in the lexical sense (even though, if interpreted, they represent the same number).

Control Functions (IF, REPEAT, WHILE)

The functions IF, REPEAT, and WHILE are useful for controlling the expansion of macros depending on whether an expression evaluates to True (−1H, or any odd number) or False (00H, or any even number).

Unlike most instances of expressions in MPL (except for SUBSTR, described below), expressions in the first clause of IF, REPEAT, and WHILE are automatically interpreted as numbers, not strings. As a result, you do not need to code %EVAL(expr) as the first clause to the functions; the expression itself suffices.

The syntax for these expressions is as follows:

IF (expr) THEN (replacement-value) [ELSE (replacement-value)] FI

REPEAT (expr) (replacement-value)

WHILE (expr) (replacement-value)

where:

- "expr" must evalute to an integer. (Note that it is not necessary to code %EVAL(expr) for these three functions; the expression is automatically evaluated without your specifying EVAL.)
- "replacement-value" is an arbitrary string with balanced parentheses, and can contain macro calls.

The IF Function

If "expr" evaluates to an ODD integer, it is considered "True" and the value of the THEN-clause replaces the IF call. If macro calls appear in the THEN clause, the calls are made and replaced by their (string) values. Any side-effects inherent in the definition of the macro(s) called are performed.

If "expr" evaluates to an EVEN integer, it is considered "False" and the THEN-clause is ignored. The ELSE clause, if present, is then treated as if it were the THEN-clause in the "True" case.

For example, the call:

```
%IF (%LEN(ABC) EQ 3) THEN (%PROCESS) FI
```

Says, in effect:

1. Treat the expression %LEN(ABC) EQ 3 as a number, and evaluate it. (The IF built-in function, like several others, accepts an expression and treats it as a number, so you do not have to use EVAL here.)

2. If %LEN(ABC) EQ 3 evaluates False (00H), end processing of this call. (There is no ELSE clause in this particular instance.)

3. If %LEN(ABC) EQ 3 evaluates True (-1H), evaluate the call %PROCESS (a user-defined function). This means:
   - Replace the entire %IF call with the value of %PROCESS (possibly null).
   - Perform any side-effects indicated in the definition of %PROCESS.

Since the value of %LEN(ABC) EQ 3 is True (-1H), the call to PROCESS is made, %PROCESS is evaluated, and its value (a string) replaces the %IF call. Any side-effect processing inherent in the definition of process is also performed. (For instance, PROCESS may define a new user macro.)

If, on the other hand, the following IF call is made:

```
%IF (%EQS( %LEN(ABC), 3)) THEN (%PROCESS) FI
```

The IF-clause first reduces to:

```
%EQS( 03H, 3)
```

And since the string comparator function EQS does not regard '03H' as equal to '3', the expression evaluates to False, or 00H. Hence, PROCESS is not called.

As another example, the call:

```
%IF (%LEN(%STRING) GT 255) THEN (%TRUNC) ELSE (%CONCAT) FI
```

results in the following:

1. The user macro-call %STRING is evaluated and replaced by a (possibly null) string.

2. The length of the string is computed by LEN.

3. The relational expression:

   ```
   xyH GT 255
   ```

   is evaluated, where "xyH" represents the value of %LEN(%STRING).

4. If the hexadecimal value xyH returned by LEN is greater than 255, the user-macro TRUNC is evaluated, and any side-effects inherent in its definition are performed. The value of TRUNC replaces the IF call (in this case the line). The ELSE-clause is ignored.

5. If the hexadecimal value returned by LEN is less than or equal to 255, the expression %TRUNC is ignored, but the user macro CONCAT is called, expanded, and any side-effects are performed.

The REPEAT Function

The expression "expr" is evaluated only once; the "replacement-value" is then evaluated "expr" times, and becomes the value of the REPEAT function.

The format of the REPEAT function call is:

%REPEAT (expr) (string)

For example,

```
%REPEAT (10) (%REPEAT (4)(-)+)
```

generates the string:

```
----+----+----+----+----+----+----+----+----+----+
```

The WHILE Function

The WHILE function call has the format:

%WHILE (expr) (replacement-value)

where "expr" is evaluated until it is False (Even) as follows:

1. The expression "expr" is first evaluated to determine whether the second ("replacement-value") need be evaluated:
   - If "expr" evaluates to an odd ("True") number, then "replacement-value" is evaluated, including all macro calls and side-effects.
   - If "expr" evaluates to an even number ("False"), then no further processing is performed for the macro call.
2. At this point, if "expr" evaluated True, "expr" is reevaluated ("replacement-value" may have called a macro to change a value in the expression), and the two listed conditions again apply. This "looping" is continued until "expr" evaluates "False".

For example, the macro call:

```
%WHILE (%EQS(%ANSWER,YES)) (%CONTINUE)
```

Evaluates as follows:

1. %ANSWER (a user function) is evaluated, and lexically compared to the string 'YES'. (Presumably, the definition of ANSWER includes macro-time console I/O, defined below.)
2. If the strings compare equal, %CONTINUE (a user function) is evaluated, including side-effects. The value (a string) of %CONTINUE replaces the %WHILE call. Note that side-effects could include redefining ANSWER. Step 1 above is then repeated.
3. If the strings compare unequal, processing of this WHILE call stops. Any %CONTINUE values placed in the intermediate file remain.

The MATCH Function

The MATCH function allows you to manipulate lists. The syntax is:

MATCH (name1 , name2) (list)

where "list" is a string value consisting of a list of strings (none of which contains a comma) separated by commas. The value of the MATCH function is always null. MATCH is used for its side-effects, which are as follows:

- name1 is assigned as a value the substring of "list" preceding the first occurrence of a comma
- name2 is assigned as a value the substring of "list" following the first occurrence of a comma

Its primary use is to isolate and name substrings of a given string, as shown in the following example, and also in the final example under "Console I/O".

For example, the following call to WHILE:

```
%WHILE (%LEN(%LIST) NE 0) (%MATCH (ITEM, LIST) (%LIST) %PROCESS(%ITEM))
```

results in the following macro processing:

1. First the length of the string defined by the user-macro LIST is evaluated. If it is nonzero, the second clause of WHILE is evaluated. If it is zero, macro expansion stops.
2. MATCH in the second clause of WHILE looks for a comma in the string defined by LIST. If a comma is found, the substring of LIST preceding the comma is assigned as the value of ITEM, and LIST takes on as a new value its substring following the occurrence of the comma.
3. Processing at this point is still in the second clause of WHILE. Next, ITEM is evaluated (the substring just found preceding the comma) and is fed to PROCESS (a user-defined macro) as an argument. If PROCESS has a value, it is inserted in the intermediate file, replacing the WHILE call.
4. Now the second clause of WHILE has been processed, so the macro processor returns to the first clause to evaluate the condition. Here, this is the same as saying, "Go to Step 1 above." Note that LIST has been redefined.

As you can see, this represents a different perspective on algorithms from that usually encountered in assembly-languages and garden-variety procedural languages. The net effect of the preceding example is to filter through the list, stopping at each comma, and assigning each substring between commas (and the substring preceding the first comma, and the substring following the last comma) to ITEM, and then processing ITEM with the macro call to PROCESS. This represents an extremely powerful tool for programming any machine, and especially the 8086. Finally, when you consider that MPL permits virtually any character combination to be used as a delimiter-specifier (not just commas), you can appreciate the assembly-time processing power here.

NOTE

This is actually a simplified form of MATCH, using a comma as a delimiter to match against in a list. The MPL language and implementation permit delimiters of very nearly any character combination. An example below (under "Console I/O") shows a different use of MATCH, matching against the carriage-return and line-feed characters considered jointly as a single delimiter. Refer to Appendix L for the full definition of MATCH.

Console I/O; Interactive Macro Assembly

The MPL built-in functions IN and OUT perform macro-time console I/O.

IN reads one line (including line-feed and carriage-return) from the console input device. The value of IN is the string typed, including the terminating carriage-return and line-feed bytes (0D0AH). The syntax is:

IN

OUT writes a string to the console output device. OUT has one parameter, the string to be written. The syntax of OUT is:

OUT(string)

where "string" must have the same number of left- and right-parentheses. The value of OUT is the null string.

MATCH can be used to strip the terminating carriage-return and line-feed characters from the value of IN:

```
%MATCH (INPUT %CRLF NULL) (%IN)
```

where CRLF is defined as follows (note the embedded carriage-return):

```
%*DEFINE (CRLF) (
)
```

If this is done, the above call to MATCH assigns the input text to INPUT and the null string (i.e. the string following the carriage-return-line-feed) to NULL.

The following example, when included in your source file and submitted for assembly, will prompt you for information to define a record array in which each record contains three fields. The prompt character is ">"):

```
%*DEFINE (REC(F)) LOCAL RECORDNAME (
%RECORDNAME RECORD %ITEM %REPEAT (%F-1) (, %ITEM)
%ARRAYNAME %RECORDNAME %EVAL(%NUMREC) DUP (<>)
)
%*DEFINE (ITEM) (%FLDNAME : %FLDWIDTH = %FLDVAL)
%*DEFINE (FLDNAME) (%OUT(NAME OF FIELD? ) %GET)
%*DEFINE (FLDWIDTH) (%OUT(WIDTH OF FIELD? ) %GET)
%*DEFINE (FLDVAL) (%OUT(INITIAL VALUE OF FIELD? ) %GET)
%*DEFINE (ARRAYNAME) (%OUT(NAME OF RECORD ARRAY?) %GET)
%*DEFINE (NUMREC) (%OUT(NUMBER OF RECORDS IN ARRAY?) %GET)
%*DEFINE (GET) (%MATCH (LINE %(
) NULL) (%IN) %LINE)
%REC(3)
```

If you want five fields instead, for example, change the call from %REC(3) to %REC(5). Or, you can define a function prompting you (or a user) for the number of record fields. Once you have some facility with MPL, you'll see vast possibilities. For instance, by inserting calls to EVAL in the definitions, you can increase the capability of the program to include expression (rather than constant) input.

The SET Function

The SET function allows you to assign a macro-time numeric value to a macro-time variable. The format is:

%SET (name, value)

where:

name is an MPL identifier

value is an expression acceptable to EVAL

For instance,

```
%SET (LINES, 10)
%SET (MAX, 80 - %LEN(%STRING))
%SET (CHARS, %MAX*%LINES)
```

You can use SET to redefine the same macro-time variable.

For example,

```
%SET (LINES, 10)
        o
        o
        o
%SET (LINES, 15)
        o
        o
        o
%SET (LINES, %LINES+1)
```

the last statement increments the macro-time variable LINES by 1.

Unlike the other MPL built-in functions, the SET function *can* be redefined (but this is not recommended).

For example,

```
%*DEFINE(SET(X)(%DEFINE(%X)(-H))
```

The SUBSTR Function

You can isolate a substring of a string or string expression using the SUBSTR built-in function. The format is:

SUBSTR(string-expr,expr1, expr2)

where:

string-expr is a string or an MPL expression which evaluates to a string.

expr1 evaluates to a string constant representing a number. This number is taken to be the character number of the beginning of the selected substring of the value of **string-expr**. The first character of the argument string is character number 1.

expr2 evaluates to a string representing a number. This number is taken to be the length of the selected substring.

SUBSTR evaluates to a null string if:

- **expr1** = 0 or **expr1** > %LEN(string-expr)
- **string-expr** evaluates to the null string
- **expr2** = 0

If **expr2** > %LEN(**string-expr**) - **expr1** + 1, then the selected substring begins at character number **expr1** and ends at character number %LEN (**string-expr**).

SUBSTR Examples

```
%SUBSTR(ABC,1,2) = AB
%SUBSTR(A B C, 1, 3) = A B
%SUBSTR(ABC, 0, 1) = (null)
%SUBSTR(ABC, 4, 1) = (null)
%SUBSTR(ABC, 2, 2) = BC
%SUBSTR(ABC, 2, 3) = BC
%SUBSTR(ABC, 3, 1) = C
%SUBSTR(%(A,B,C), 1, 2) = A,
```

CHAPTER 8
MODELS OF COMPUTATION: RECOMMENDED PRACTICES

Recommendations

1. Place EQUates to registers and numbers at the top of your program.
2. Place data segments before code segments.
3. Within code segments, place definitions of any variables early, meaning as near as you can to the first segment directive defining that segment.
4. Where possible, make modules private (non-combinable) and paragraph-aligned.
5. Try to consolidate the use of public symbols in modules assembled separately from those which neither need them as externals nor supply them as publics.

The basic unit of assembly program in this language is a module. Within modules the basic unit of contiguous code or data is a segment. Memory layout and addressability (via base addresses in the "segment registers") require the use of segments. Segments can be placed anywhere in memory by the LOCATE facility. Their order in an assembly is thus not necessarily their sequence in memory during execution.

To the assembler, however, certain orderings are distinctly preferable in the interest of creating optimal code using minimal memory. These orderings prevent most of the ambiguities and possible errors associated with forward referencing.

Forward-referencing refers to the assembler working with a variable or label whose definition has not yet been scanned. In this situation the assembler must reserve enough room for the address or number to come. It chooses either the most probable case of 1 word (based on these recommendations) or the "worst" case of 2 words, i.e., room for both the offset and segment (paragraph-number). In the absence of user-supplied data, it chooses 2 bytes. Given the definitions of segments and variables prior to their use in instructions, the assembler can choose the optimal 8086 machine instruction to generate and can reserve only the minimum bytes needed.

A one word offset is adequate to access any byte within the 64K bytes above a base-address in one of the segment registers, and 64K is the maximum size of a segment. The assumption is that the definition will be found later in this assembly. Otherwise you would have already supplied it in a segment scanned earlier, or in an EXTRN directive, which gives its attribute and says not to expect its full definition in this segment. If this reasoning fails because you supply no definition at all, then an error is flagged for you to resolve before re-assembling.

When a 2-word space is reserved, it is always adequate. If the forward-reference is ultimately defined in this segment, 1 word suffices and the other is unused. This is safe but non-optimal. If the forward-reference is never defined even in an EXTRN, an error is flagged.

The recommendation that code be placed in segments which are non-combinable and paragraph-aligned allows faster assembly, linkage, and relocation as well as increased optimization of code. When a one-module program has no groups, no need for external variables or labels, and provides no publics, then linking can be skipped entirely. The program is ready for direct relocation into absolute addresses.

Forward Referencing

This is a 2-pass assembler, meaning it goes through a representation of your source code twice.

By placing data segments early in the module, and variables early within code segments, you enable the assembler to recognize the attributes (type, segment, offset) of these operands in the code it sees later. Armed with this knowledge, it produces the tighest code it can, by using 1 byte instead of 2, or 2 bytes instead of 4, wherever possible. References to data always use a 2 byte offset, but transfers of control (jumps or calls) can vary requiring 1 or 2 or 4 bytes depending on the context of definition and usage.

In the absence of special coding, the assembler assumes forward references require 1 word, with no implicit segment override to be discovered later. You may code an explicit segment override, and in some cases cause a double-word space or a byte space to be reserved for the forward reference instead of the usual word. Registers may not be forward-referenced, i.e., if a forward-reference is later found to be defined as a register, you get an error.

Variables and Labels

For a forward-reference variable, e.g.,

```
MOV     AL, FRVAR
```

which could be defined anywhere beyond this reference, the assembler reserves a full word for the offset of the variable. For a forward-reference label, e.g.,

```
JMP   FRLAB
```

the assumption is that FRLAB will be typed NEAR later in this segment, hence 1 word is sufficient for its address.

However, if FRLAB is found to be declared FAR, or not in the current segment or group then you get an error. Such a reference would require an operand of 2 words, the first being the offset address of the label in its segment, the second being the segment-base-address for insertion into CS. Thus 2 words are needed but only 1 was reserved after the JMP instruction word.

If it turns out in pass 2 that a smaller operand is sufficient, the remainder of the space in pass 1 is no-op instructions (90H). This is usually of little concern if forward-references are kept to a minimum, by following the above recommended practices.

In some cases you know that when the reference is ultimately resolved, it will fit in less space (or more) than the assembler can assume. You may override the default by using the attribute-changing operators of the language. For example, if you know that FRLAB was to be defined within the next 127 bytes of this segment, you could write:

```
JMP   SHORT   FRLAB
```

causing the assembler to reserve only 1 byte for this forward-reference, instead of the normal 2 bytes. (A segment-override may be required, as discussed below.)

Similarly, if you know FRLAB is a label defined later in a different code segment, you may write:

```
JMP  FAR  PTR  FRLAB
```

causing the assembler to reserve 2 words instead of only 1.

There are some further issues mentioned below which are discussed in greater detail under the ASSUME and GROUP directives.

Segments

A forward-reference in an ASSUME directive, e.g.,

```
ASSUME  DS:FORWREF, ES:SEG2
```

will be taken to be a segment name. If FORWREF turns out later to be a group-name, or anything else other than a segment name, you will get an error message and must re-assemble.

A forward-referenced variable might need a segment prefix byte. If so, you must code it or refer to it explicitly, e.g.,

```
MOV   AL, ES:FRVAR
MOV   AL, SEG2:FRVAR
```

Otherwise the assembler leaves no room for that prefix byte in pass 1, and if it turns out in pass 2 to be necessary, you will get an error message and must re-assemble. Note that this use of SEG2 above generates a prefix byte only because in the prior ASSUME, SEG2 is not in DS.

PLM86 Linking Conventions

GROUPs are necessary to link ASM86 and PLM86 programs and procedures in some cases. There are established conventions for passing data, parameters, or addresses between programs written in these languages.

These cases and conventions are described in detail in the *MCS-86 Macro Assembler Operating Instructions for ISIS-II Users*.

APPENDIX A
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, RF10:3

  CodeMacro     AAA
    DB          37H
    EndM

  CodeMacro     AAD
    DW          0AD5H
    EndM

  CodeMacro     AAM
    DW          0AD4H
    EndM

  CodeMacro     AAS
    DB          3FH
    EndM

  CodeMacro     Adc   dst:Eb, src:Db
    Segfix      dst
    DB          80H
    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          10H
    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          80H
    ModRM       0, dst
    DB          src
    EndM

  CodeMacro     Add   dst:Ew,src:Db
    Segfix      dst
    DB          81H
    ModRM       0, dst
    DW          src
    EndM

  CodeMacro     Add   dst:Ew, src:Db(-128,127)
    Segfix      dst
    DB          83H
    ModRM       0, dst
    DB          src
    EndM
```

```
CodeMacro   Add   dst:Ew, src:Dw
  Segfix    dst
  DB        81H
  ModRM     0, dst
  DW        src
  EndM

CodeMacro   Add   dst:Ab, src:Db
  DB        04H
  DB        src
  EndM

CodeMacro   Add   dst:Aw, src:Db
  DB        05H
  DW        src
  EndM

CodeMacro   Add   dst:Aw, src:Dw
  DB        05H
  DW        src
  EndM

CodeMacro   Add   dst:Eb, src:Rb
  Segfix    dst
  DB        0
  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, scr:Ew
  Segfix    src
  DB        3
  ModRM     dst,src
  EndM

CodeMacro   And   dst:Eb, src:Db
  Segfix    dst
  DB        80H
  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        20H
  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

CodeMacro   Call  addr:Ew
  Segfix    addr
  DB        0FFH
  ModRM     2, addr
  EndM

CodeMacro   Call  addr:Ed
  Segfix    addr
  DB        0FFH
  ModRM     3, addr
  EndM
```

```
CodeMacro   Call  addr:Cd
  DB        9AH
  DD        addr
  EndM

CodeMacro   Call  addr:Cb
  DB        0E8H
  RelW      addr
  EndM

CodeMacro   Call  addr:Cw
  DB        0E8H
  RelW      addr
  EndM

CodeMacro   CBW
  DB        98H
  EndM

CodeMacro   CLC
  DB        0F8H
  EndM

CodeMacro   CLD
  DB        0FCH
  EndM

CodeMacro   CLI
  DB        0FAH
  EndM

CodeMacro   CMC
  DB        0F5H
  EndM

CodeMacro   Cmp   dst:Eb, src:Db
  Segfix    dst
  DB        80H
  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:Eb, DI_ptr:Eb
  NoSegfix  ES, DI_ptr
  Segfix    SI_ptr
  DB        0A6H
  EndM

CodeMacro   CmpS  SI_ptr:Ew, DI_ptr:Ew
  NoSegfix  ES, DI_ptr
  Segfix    SI_ptr
  DB        0A7H
  EndM

CodeMacro   CmpSB
  DB        0A65H
  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

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   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
```

```
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

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  Sl_ptr:Mb
  Segfix    Sl_ptr
  DB        0ACH
  EndM

CodeMacro   LodS  Sl_ptr:Mw
  Segfix    Sl_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        0C7H
  ModRM     0, dst
  DW        src
  EndM
```

```
CodeMacro   Mov  dst:Rb, src:Db
  R53       <10110B,dst>
  DB        src
  EndM

CodeMacro   Mov  dst:Rw, src:Db
  R53       <10111B,dst>
  DW        src
  EndM

CodeMacro   Mov  dst:Rw, src:Dw
  R53       <10111B,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        08CH
  ModRM     src, dst
  EndM

CodeMacro   Mov  dst:S(ES), src:Ew
  Segfix    src
  DB        08EH
  ModRM     dst, src
  EndM

CodeMacro   Mov  dst:S(SS,DS), src:Ew
  Segfix    src
  DB        08EH
  ModRM     dst, src
  EndM

CodeMacro   Mov  dst:Ab, src:Xb
  Segfix    src
  DB        0A0H
  DW        src
  EndM

CodeMacro   Mov  dst:Aw, src:Xw
  Segfix    src
  DB        0A1H
  DW        src
  EndM

CodeMacro   Mov  dst:Xb, src:Ab
  Segfix    dst
  DB        0A2H
  DW        dst
  EndM

CodeMacro   Mov  dst:Xw, src:Aw
  Segfix    dst
  DB        0A3H
  DW        dst
  EndM

CodeMacro   MovS  SI_ptr:Mb, DI_ptr:Mb
  NoSegfix  ES, SI_ptr
  Segfix    DI_ptr
  DB        0A4H
  EndM

CodeMacro   MovS  SI_ptr:Mw, DI_ptr:Mw
  NoSegfix  ES, SI_ptr
  Segfix    DI_ptr
  DB        0A5H
  EndM

CodeMacro   MovSB
  DB        0A4H
  EndM

CodeMacro   Mul  mplier:Eb
  Segfix    mplier
  DB        0F6H
  ModRM     4, mplier
  EndM

CodeMacro   Mul  mplier:Ew
  Segfix    mplier
  DB        0F7H
  ModRM     4, mplier
  EndM

CodeMacro   Neg  dst:Eb
  Segfix    dst
  DB        0F6H
  ModRM     3, dst
  EndM

CodeMacro   Neg  dst:Ew
  Segfix    dst
  DB        0F7H
  ModRM     3, dst
  EndM

CodeMacro   Nil
  EndM
```

```
CodeMacro   Nop
  DB        90H
  EndM

CodeMacro   Not  dst:Eb
  Segfix    dst
  DB        0F6H
  ModRM     2, dst
  EndM

CodeMacro   Not  dst:Ew
  Segfix    dst
  DB        0F7H
  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

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

CodeMacro   PopF
  DB        9DH
  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

CodeMacro   PushF
  DB        9CH
  EndM

CodeMacro   RCL  dst:Eb, count:DB(1)
  Segfix    dst
  DB        0D0H
  ModRM     2, dst
  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:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     3, dst
  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:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     0, dst
  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:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     1, dst
  EndM

CodeMacro   ROR   dst:Ew, count:Db(1)
  Segfix    dst
  DB        0D1H
  ModRM     1, dst
  EndM

CodeMacro   ROR   dst:Eb, count:Rb(CL)
  Segfix    dst
  DB        0D2H
  ModRM     1, dst
  EndM

CodeMacro   ROR   dst:Ew, count:Rb(CL)
  Segfix    dst
  DB        0D3H
  ModRM     1, dst
  EndM

CodeMacro   SAHF
  DB        9EH
  EndM

CodeMacro   SAL   dst:Eb, count:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     4, dst
  EndM

CodeMacro   SAL   dst:Ew, count:Db(1)
  Segfix    dst
  DB        0D1H
  ModRM     4, dst
  EndM

CodeMacro   SAL   dst:Eb, count:Rb(CL)
  Segfix    dst
  DB        0D2H
  ModRM     4, dst
  EndM

CodeMacro   SAL   dst:Ew, count:Rb(CL)
  Segfix    dst
  DB        0D3H
  ModRM     4, dst
  EndM

CodeMacro   SAR   dst:Eb, count:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     7, dst
  EndM

CodeMacro   SAR   dst:Ew, count:Db(1)
  Segfix    dst
  DB        0D1H
  ModRM     7, dst
  EndM

CodeMacro   SAR   dst:Eb, count:Rb(CL)
  Segfix    dst
  DB        0D2H
  ModRM     7, dst
  EndM

CodeMacro   SAR   dst:Ew, count:Rb(CL)
  Segfix    dst
  DB        0D3H
  ModRM     7, dst
  EndM

CodeMacro   Sbb   dst:Eb, src:Db
  Segfix    dst
  DB        80H
  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        0AEH
  EndM

CodeMacro   ScaS  DI_ptr:Mw
  NoSegfix  ES, DI_ptr
  DB        0AFH
  EndM

CodeMacro   ScaSB
  DB        0AEH
  EndM

CodeMacro   ScaSW
  DB        0AFH
  EndM

            SHL  Equ SAL

CodeMacro   SHR  dst:Eb, count:Db(1)
  Segfix    dst
  DB        0D0H
  ModRM     5, dst
  EndM

CodeMacro   SHR  dst:Ew, count:Db(1)
  Segfix    dst
  DB        0D1H
  ModRM     5, dst
  EndM

CodeMacro   SHR  dst:Eb, count:Rb(CL)
  Segfix    dst
  DB        0D2H
  ModRM     5, dst
  EndM

CodeMacro   SHR  dst:Ew, count:Rb(CL)
  Segfix    dst
  DB        0D3H
  ModRM     5, dst
  EndM

CodeMacro   STC
  DB        0F9H
  EndM

CodeMacro   STD
  DB        0FDH
  EndM

CodeMacro   STI
  DB        0FBH
  EndM

CodeMacro   StoS  DI_ptr:Mb
  NoSegfix  ES, DI_ptr
  DB        0AAH
  EndM

CodeMacro   StoS  DI_ptr:Mw
  NoSegfix  ES, DI_ptr
  DB        0ABH
  EndM

CodeMacro   StoSB
  DB        0AAH
  EndM

CodeMacro   StoSW
  DB        0ABH
  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        0F6H
  ModRM     0, dst
  DB        src
  EndM

CodeMacro   Test  dst:Ew, src:Db
  Segfix    dst
  DB        0F7H
  ModRM     0, dst
  DW        src
  EndM

CodeMacro   Test  dst:Ew, src:Dw
  Segfix    dst
  DB        0F7H
  ModRM     0, dst
  DW        src
  EndM

CodeMacro   Test  dst:Ab, src:Db
  DB        0A8H
  DB        src
  EndM

CodeMacro   Test  dst:Aw, src:Db
  DB        0A9H
  DW        src
  EndM

CodeMacro   Test  dst:Aw, src:Dw
  DB        0A9H
  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        09BH
  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       <10010B,dst>
  EndM

CodeMacro   Xchg  dst:Aw, src:Rw
  R53       <10010B,src>
  EndM

CodeMacro   Xlat  table:Mb
  Segfix    table
  DB        0D7H
  EndM

CodeMacro   XlatB
  DB        0D7H
  EndM

CodeMacro   Xor   dst:Eb, src:Db
  Segfix    dst
  DB        80H
  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

Purge R53,R323,R233,R413
Purge RF1,RF2,RF3,RF4,RF5
Purge RF6,RF7,RF8,RF9
Purge RF10,Mid3


END
```

APPENDIX B
MEMORY ORGANIZATION

The location of an operand in an 8086 register or in memory is specified in many instructions by up to three fields. 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. Any DISPlacement bytes (1 or 2) come last.

The mod field occupies the two most significant bits of the byte, and specifies how the r/m field is to be used.

The reg field occupies the next three bits following the mod field, and can specify either an 8-bit register or a 16-bit register to be the location of an operand. In some instructions it can further specify the instruction encoding instead of naming a register.

The r/m field either can be the location of the operand (if in a register) or can specify how the 8086 will locate the operand in memory, in combination with the mod field as shown below.

These fields are set automatically by the assembler in generating your code. They are discussed in greater detail in Chapter 7 on Code macros.

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 or data-low	disp-high or data-high

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

APPENDIX C
FLAG OPERATIONS

FLAG REGISTERS

Flags are used to distinguish or denote certain results of data manipulation. The 8086 provides the four basic mathematical operations (+, -, *, /) in a number of different varieties. Both 8- and 16-bit operations and both signed and unsigned arithmetic are provided. Standard two's complement representation of signed values is used. The addition and subtraction operations serve as both signed and unsigned operations. In these cases the flag settings allow the distinction between signed and unsigned operations to be made (see Conditional Transfer instructions in Chapter 9).

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 clear 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 destination of the last operation to affect PF. Instructions that affect the parity flag set the flag to one for even parity and reset the flag to zero to indicate odd parity.

Auxiliary Carry Flag. The auxiliary carry flag indicates a carry out of bit 3 of the accumulator. You cannot test this flag directly in your program; it is present to enable the Decimal Adjust instructions to perform their function.

The auxiliary carry flag is affected by all add, subtract, increment, decrement, compare, and all logical AND, OR, and XOR instructions.

APPENDIX D
EXAMPLES

In this Appendix, several sample problems are presented, each with several solutions.

Each code example has comments and is followed by explanatory paragraphs. Inevitably there will still be a few undefined words and less-than-crystal concepts. You may prefer to look them up in the index as soon as you encounter them. This thoroughness will increase your depth of understanding but will also slow your use of this chapter.

Another way to go about it is to note unclear items on a pad as you read—but to continue reading, leaving the detailed exploration and analysis until later. Many early questions will be answered by later examples and text; twice through this chapter might build familiarity that could save time in studying the manual as a whole.

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).

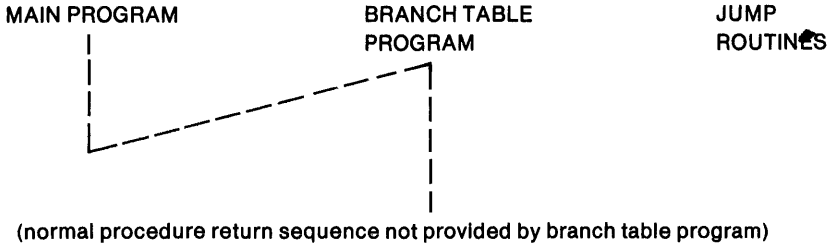
Examples 3, 4, and 5 discuss additional methods of passing data and parameters to procedures, illustrating the use of both the registers and the stack for passing parameters. Examples 6 and 7 cover multibyte addition and subtraction. Interrupt procedures and timing loops are described in examples 8 and 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



(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:

The 8086 assembly language mnemonics used below can be read and (briefly) interpreted as follows:

ASSUME	tells assembler what you intend to put in the segment registers during execution (required)
MOV	moves 2nd operand ("source") into 1st operand ("destination")
CMP	compares 2 operands by subtracting 2nd from 1st
JE	jumps to label given if comparison said "equal"
SHR	shifts operand 1 bit to the right, putting lowest order bit into carry flag
JNB	jumps to label given if carry flag is zero
JMP	jumps to label, if given; or jumps-indirect to address held as contents of the given variable or register, as here
ADD	adds source into destination
TYPE	means how many bytes in each entry. The branch table is in words, each 2 bytes

```
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_TABLE_2     DW  PROCESS_31
                             DW  PROCESS_61
                             DW  PROCESS_81
                             .
                             .
                             .
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).
          JNB       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 point 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:", JNB NOT_YET, reads "jump if not below", 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 1 and this conditional jump JNB 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 cell 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 cell in the data segment, a cell 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.

Only BX, BP, SI, and DI are permitted within square brackets.

BRANCH_TABLE_2 is unused in this example. It is shown only as an indication that data segments may contain multiple tables referenced at different times by different code segments.

NOTES

Note that the ASSUME statement is necessary, to identify what the run-time contents of CS and DS will be.

If you have already looked at the Attributes of Symbols section in Chapter 2, then it should be noted also that all routines whose labels are in the branch table must be defined under the same CS:assumption as the code that transfers to them. The reason is that in this example, they are to be NEAR jumps, using only the one word offset. They need not necessarily be in the same segment, but the same contents of CS must be ASSUMEd. This is also indicated by the phrase WORD PTR preceding [BX], indicating the intent to use one word from the table as an offset.

This restriction does not apply to FAR jumps or calls. Thus it would not be necessary to ensure that the same ASSUME CS:name in fact applied to each branch table entry, if the requirements for FAR jumps were coded. These requirements are given below:

In the above example, it would be necessary to change the JMP WORD PTR [BX] to read JMP DWORD PTR [BX]. It would also be necessary to change each BRANCH_TABLE entry into an entry of the form

```
DD  ROUTINE_1
```

so that the transfer would replace the contents of CS as well as IP. Attributes of symbols, such as NEAR and FAR, are discussed in Chapters 2, 4, and 5. PTR is also discussed in Chapter 5 on Expressions. Jumps and calls are further explained later in this appendix and in Chapter 6. ASSUME is in Chapter 4.

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__TABLE__2        DW  PROCESS_31
                             DW  PROCESS_61
                             DW  PROCESS_81
                              .
                              .
                              .
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
                 JNB    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 AL 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
```

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.

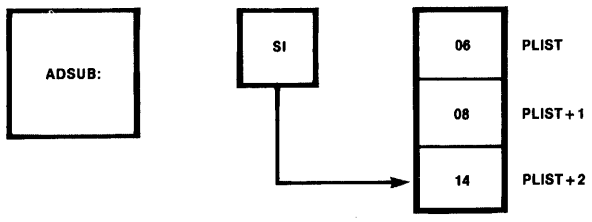
Example 3 shows the use of memory to pass parameters. Registers are used for this in Example 4. Example 5 uses a stack.

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:

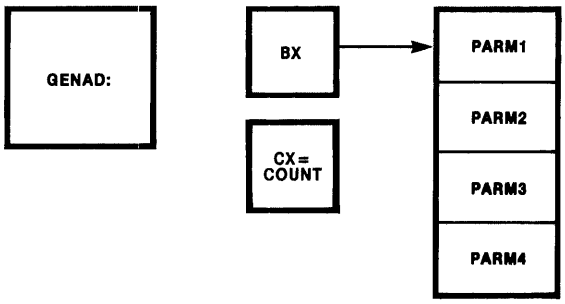
```
PARAMS  SEGMENT
PLIST       DB      6
            DB      8
            DB      ?
LIST2       DB      10
            DB      35
            DB      ?
              .
              .
              .
PARAMS  ENDS
STACK   SEGMENT
        DW  4  DUP  (?)
STACK_TOP  LABEL WORD
STACK   ENDS
ADDING  SEGMENT
ASSUME  CS:ADDING,    DS:PARAMS, SS:STACK
START:      MOV       AX,PARAMS
            MOV       DS,AX
            MOV       AX,STACK
            MOV       SS,AX
            MOV       SP,OFFSET STACK_TOP
            MOV       SI,OFFSET PLIST
CALL1:      CALL      ADSUB
RET1:         .
              .
              .
            LEA       SI,LIST2
CALL2:      CALL      ADSUB
 RET2:
              .
              .
              .
              .
              .
ADSUB       PROC
            MOV       AL,[SI]
            ADD       AL,[SI+1]
            MOV       [SI+2],AL
            RET
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 5). 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 the 8086 assembly language. 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


GPR         EQU     GENERAL_PROCEDURES
PR1         EQU     INITIAL_PARAMETERS

GENERAL_PROCEDURES      SEGMENT
            ASSUME  CS:GPR,   DS:PR1     ; uses short synonyms from EQUs


; The procedure is placed first, to avoid forward referencing
; the FAR procedure GENAD86. Note that the program start address
; is after the procedure, at label "START".


GENAD86     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.

GENAD86   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 GENAD86 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  GENAD86
          MOV   RESULT, AL             ; Sum is returned in AL


          HLT                          ; ******* end of program *******
GENERAL_PROCEDURES    ENDS
          END   START
```

In Example 4 the general guidelines for the 8086 Assembly Language are followed by coding first the data segments and EQUs, followed by the code segments which refer to these program elements. The EQUs enable names to be used in place of numeric values, or shorter synonyms instead of longer names.

A forward reference to an EQU is allowed. An EQU may refer to a later-defined simple name, (but not to a later-defined full address-expression).

In GENAD86, 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 GENAD86, 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.

As mentioned earlier, BX, BP, SI, and DI are the only registers permitted within square brackets. Such usage is further restricted: in any one expression you may use BX or BP, but not both, and SI or DI, but not both. Thus [BX][SI] is valid but [BX][BP] is not. This is discussed in greater detail in Chapter 5.

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 Appendix B of the Operator's Guide 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 TOS, 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_pass
```

```
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 + 6]  ; get count of number of bytes in array
          MOV    BX, [BP + 8]  ; 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, OFFSEST 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
```

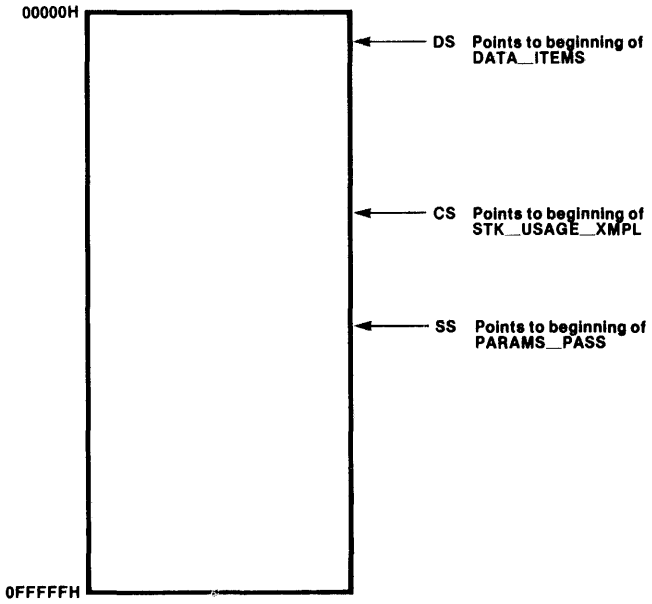


Figure 1

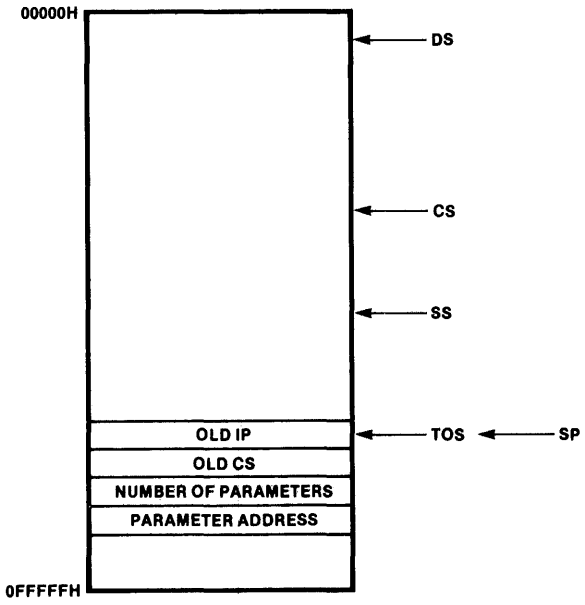


Figure 2

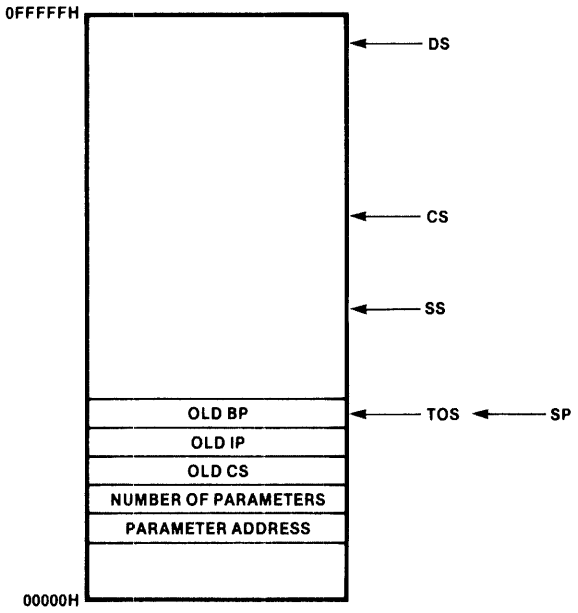


Figure 3

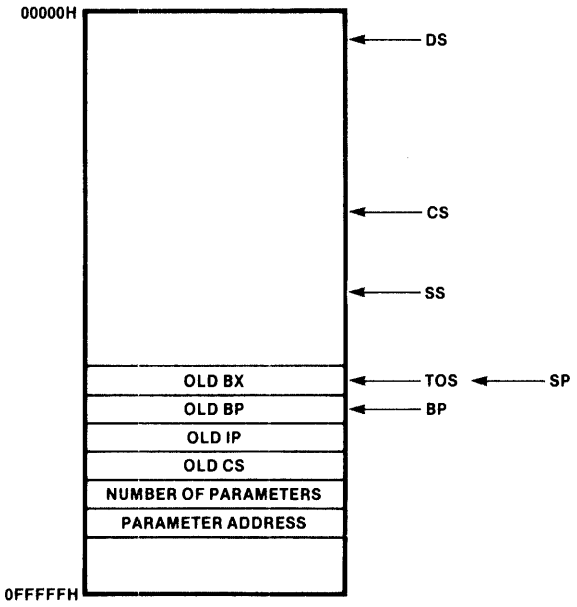


Figure 4

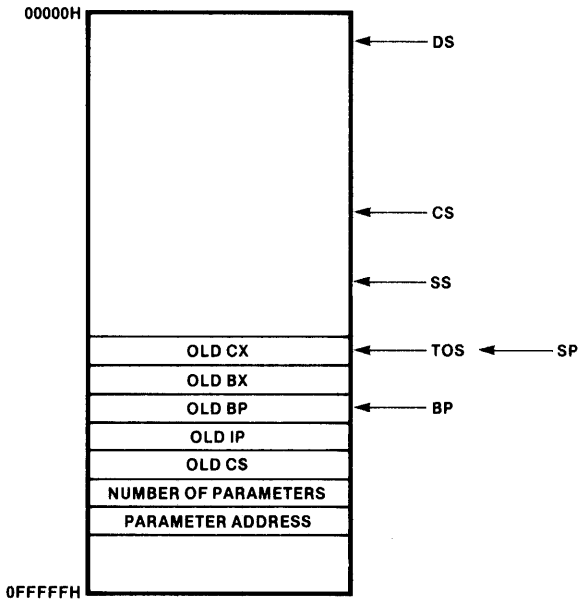


Figure 5

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.

After three calls, the old six parameter words would use up half the stack. The first would be in LAST_WORD-2, next in LAST_WORD-4, LAST_WORD-6, etc. to LAST_WORD-12. The fourth use of the procedure would put on the two parameters, and then the return address would go in LAST_WORD-18 and LAST_WORD-20. The procedure's PUSHes of original register contents would fill LAST_WORD-22 and LAST_WORD-24, and then two words outside PARAMS_PASS. (Those two words would be at offsets of +0FFFE and +0FFFC, since address arithmetic is done modulo 64K. That is, the offset of LAST_WORD is 24, so the location whose offset is 26 less than 24 has offset 0FFFE.)

```
LAST_WORD-2   1st  param       +22
            -4   2nd  param     +20
            -6   3rd  param     +18
            -8   4th  param     +16
           -10   5th  param     +14
           -12   6th  param     +12
           -14   7th  param     +10
           -16   8th  param     +8
           -18   instr pointer  +6
LAST_WORD-20  old CS           +4
LAST_WORD-22  old BP           +2
LAST_WORD-24  old BX           0
```

```
  0000
 -0000
 -----
  FFFE
 -0002
 -----
  FFFC
```

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.

When location DONE is reached, bytes FIRST through FIRST+2 will contain 1A6AB7H, which is the sum shown at the beginning of this section, from low-order byte to high-order byte.

If you change the ADC instruction to an SBB instruction, the routine becomes a multibyte subtraction process. It will then subtract the number beginning at SECOND from that at FIRST, placing the result at FIRST. (Different length numbers are not handled.)

Example 6:

```
ADDDATA         SEGMENT

FIRST     DB    8AH,0AFH, 32H
SECOND    DB    90H,0BAH,84H
ADDDATA         ENDS

MULTIBYTE_ADD     SEGMENT

ASSUME    CS:MULTIBYTE_ADD,
&         DS:ADDDATA
START:    MOV  AX,ADDDATA
          MOV  DS,AX
          MOV  CX,LENGTH FIRST   ; Number of bytes in each
                  ; addend.Controls # of loop iterations.
          MOV  SI,0
          CLC              ; Clears any prior carry.
LOOPER:   MOV  AL, SECOND [SI]   ; Each successive byte
                           ; replaces AL
          ADC  FIRST [SI], AL   ; Parallel byte added with carry.
          INC  SI          ; Index incremented by 1
          LOOP  LOOPER ; CX = CX-1. Repeat till CX=0,
                           ; then fall thru to DONE

DONE:
          .                ;
          .                ;
          .                ;
MULTIBYTE_ADD     ENDS
          END     START
```

The two numbers could be of different lengths, e.g., one 5 bytes long and the other 3 bytes long. If so, the routine below would perform the multibyte addition. However, a carry out of the highest byte of the longer number would be lost. This could be handled by additional code to check the flags, or by the expedient of an extra high-order byte on the longer number.

```
  ADD_DATA_2    SEGMENT

FIRST  DB  11,22,33
NUM1   DW  LENGTH FIRST

SECOND  DB  99,88,77,66,55
NUM2   DW  LENGTH SECOND

ADD_DATA_2    ENDS

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.

        ORG  80H
; 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.

                        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 device_1
                MOV     INPUT_1_VAL, AL   ; 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 occurs,
                PUSH    AX      ; 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 occurs,
            PUSH    AX      ; only the lower byte at the port is
            MOV     DX, DEVICE_3_PORT       ; 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 provides
            PUSH    AX      ; a value which needs immediate
            MOV     DX, DEVICE__4__PORT
                            ; conversion by another procedure
            IN      AL,DX   ; before this interrupt-handler can
            MOV     CL, AL  ; allow it to be used at input_4_val

            CALL    CONVERT_VALUE       ; converts input value in CL
            MOV     INPUT_4_VAL, AL     ; 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
```

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 in 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     @1          ; if so, return TRUE

    MOV     AL, 0       ; if not, return FALSE: AL=0
    POP     BP          ; restore old value
    RET                 ; done, no char waiting

@1:

    MOV     AL, 0FFH    ; 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 0FFH
       FALSE  EQU 0H
STATUS_PORT   EQU 0FFF2H
```

```
DATA_PORT   EQU  0FFF0H
ASCII_MASK  EQU  7FH
CONTROL_S   EQU  13H
CONTROL_Q   EQU  11H
CARR_RET    EQU  0DH

SIO_CHAR_RDY2    PROC    NEAR

        PUSH  BX                        ; save old BX value
        MOV   BL, TRUE                  ; prepare for one reesult
        MOV   DX, STATUS_PORT           ; check the facts
        IN    AL,DX                     ; char waiting???
        TEST  AL, 2H      ; if 2nd bit ON, char is waiting
        JNZ   RESULT      ; hence skip over FALSE set-up
        MOV   BL, FALSE   ; here if 2nd bit was OFF,
                          ; hence no char waiting
RESULT: MOV   AL, BL      ; AL receives whichever result
        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   @117                  ; 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   @117                  ; if this halt-display signal
                                    ; is not rec'd, continue
                                    ; output at @117

@115:                               ; if control-S rec'd, must
                                    ; await its release
        CMP   CHAR, CONTROL_Q       ; Control-Q received?
        JZ    @117                  ; 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 @115
        JNB   @115                  ; 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     @115                    ; skip this instruction, which
                                        ; loops to await control-Q, and
        JMP     NEXTCOMMAND             ; go to NEXTCOMMAND

@117:
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      @117                    ;

        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)

    @121:

        CMP   BYTE PTR ES: [BX] [SI], 0
    ; terminator character is ASCII null = all zeroes.
        JZ    @122    ; 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   @121
    @122:
        POP   BP
        RET   4     ; after return, resets
                    ; SP behind former parameters

    SIO_OUT_STRING   ENDP
```

APPENDIX E
INSTRUCTIONS IN HEXADECIMAL ORDER

00	00000000	MOD REGR/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01	00000001	MOD REGR/M	ADD	EA,REG	WORD ADD (REG) TO EA
02	00000010	MOD REGR/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03	00000011	MOD REGR/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 REGR/M	OR	EA,REG	BYTE OR (REG) TO EA
09	00001001	MOD REGR/M	OR	EA,REG	WORD OR (REG) TO EA
0A	00001010	MOD REGR/M	OR	REG,EA	BYTE OR (EA) TO REG
0B	00001011	MOD REGR/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 REGR/M	ADC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11	00010001	MOD REGR/M	ADC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12	00010010	MOD REGR/M	ADC	REA,EA	BYTE ADD (EA) W/ CARRY TO REG
13	00010011	MOD REGR/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 REGR/M	SBB	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19	00011001	MOD REGR/M	SBB	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A	00011010	MOD REGR/M	SBB	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B	00011011	MOD REGR/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 REGR/M	AND	EA,REG	BYTE AND (REG) TO EA
21	00100001	MOD REGR/M	AND	EA,REG	WORD AND (REG) TO EA
22	00100010	MOD REGR/M	AND	REG,EA	BYTE AND (EA) TO REG
23	00100011	MOD REGR/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 REGR/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29	00101001	MOD REGR/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A	00101010	MOD REGR/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B	00101011	MOD REGR/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 REGR/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31	00110001	MOD REGR/M	XOR	EA,REG	WORD XOR (REG) TO EA
32	00110010	MOD REGR/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33	00110011	MOD REGR/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 REGR/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39	00111001	MOD REGR/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A	00111010	MOD REGR/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B	00111011	MOD REGR/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	BX	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		(not used)		
61 01100001		(not used)		
62 01100010		(not used)		
63 01100011		(not used)		
64 01100100		(not used)		
65 01100101		(not used)		
66 01100110		(not used)		
67 01100111		(not used)		
68 01101000		(not used)		
69 01101001		(not used)		
6A 01101010		(not used)		
6B 01101011		(not used)		
6C 01101100		(not used)		
6D 01101101		(not used)		
6E 01101110		(not used)		
6F 01101111		(not used)		
70 01110000		JO	DISP8	JUMP ON OVERFLOW
71 01110001		JNO	DISP8	JUMP ON NOT OVERFLOW
72 01110010		JB/JNAE	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73 01110011		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
85 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 REGR/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85 10000101	MOD REGR/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86 10000110	MOD REGR/M	XCHG	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87 10000111	MOD REGR/M	XCHG	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88 10001000	MOD REGR/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89 10001001	MOD REGR/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A 10001010	MOD REGR/M	MOV	REG,EA	BYTE MOVE (EA) TO REG
8B 10001011	MOD REGR/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 REGR/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	DST8SRC8	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
AA10101010		STOS	DST8	BYTE STORE, STRING OP
AB10101011		STOS	DST16	WORD STORE, STRING OP
AC10101100		LODS	SRC8	BYTE LOAD, STRING OP

AD 10101101		LODS	SRC16	WORD LOAD, STRING OP
AE 10101110		SCAS	DIPTR8	BYTE SCAN, STRING OP
AF 10101111		SCAS	DIPTR16	WORD SCAN, STRING OP
B0 10110000		MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001		MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010		MOV	DL,DATA8	BYTE MOVE DATA TO REG DL
B3 10110011		MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100		MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101		MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110		MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111		MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000		MOV	AX,DATA16	WORD MOVE DATA TO REG AX
B9 10111001		MOV	CX,DATA16	WORD MOVE DATA TO REG CX
BA 10111010		MOV	DX,DATA16	WORD MOVE DATA TO REG DX
BB 10111011		MOV	BX,DATA16	WORD MOVE DATA TO REG BX
BC 10111100		MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD 10111101		MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE 10111110		MOV	SI,DATA16	WORD MOVE DATA TO REG SI
BF 10111111		MOV	DI,DATA16	WORD MOVE DATA TO REG DI
C0 11000000		(not used)		
C1 11000001		(not used)		
C2 11000010		RET	DATA16	INTRA SEGMENT RETURN, ADD DATA TO REG SP
C3 11000011		RET		INTRA SEGMENT RETURN
C4 11000100	MOD REGR/M	LES	REG,EA	WORD LOAD REG AND SEGMENT REG ES
C5 11000101	MOD REGR/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		(not used)		
C9 11001001		(not used)		
CA 11001010		RET	DATA16	INTER SEGMENT RETURN, ADD DATA TO REG SP
CB 11001011		RET		INTER SEGMENT RETURN
CC 11001100		INT	3	TYPE 3 INTERRUPT
CD 11001101		INT	TYPE	TYPED INTERRUPT
CE 11001110		INTO		INTERRUPT ON OVERFLOW
CF 11001111		IRET		RETURN FROM INTERRUPT
D0 11010000	MOD 000 R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD 001 R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD 010 R/M	RCL	EA,1	BYTE ROTATE EA LEFT THRU CARRY 1 BIT
D0 11010000	MOD 011 R/M	RCR	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD 100 R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD 101 R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD 110 R/M	(not used)		
D0 11010000	MOD 111 R/M	SAR	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD 000 R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT
D1 11010001	MOD 001 R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1 11010001	MOD 010 R/M	RCL	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1 11010001	MOD 011 R/M	RCR	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1 11010001	MOD 100 R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1 11010001	MOD 101 R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1 11010001	MOD 110 R/M	(not used)		
D1 11010001	MOD 111 R/M	SAR	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT
D2 11010010	MOD 000 R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2 11010010	MOD 001 R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2 11010010	MOD 010 R/M	RCL	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2 11010010	MOD 011 R/M	RCR	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2 11010010	MOD 100 R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2 11010010	MOD 101 R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2 11010010	MOD 110 R/M	(not used)		
D2 11010010	MOD 111 R/M	SAR	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS

D3 11010011	MOD 000 R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3 11010011	MOD 001 R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS
D3 11010011	MOD 010 R/M	RCL	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3 11010011	MOD 011 R/M	RCR	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3 11010011	MOD 100 R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3 11010011	MOD 101 R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3 11010011	MOD 110 R/M	(not used)		
D3 11010011	MOD 111 R/M	SAR	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4 11010100	00001010	AAM		ASCII ADJUST FOR MULTIPLY
D5 11010101	00001010	ADD		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
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
EA11101010		JMP	DISP16,SEG16	DIRECT INTER SEGMENT JUMP
EB11101010		JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC11101010		IN	AL,DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED11101010		IN	AX,DX	WORD INPUT FROM PORT (DX) TO REG AX
EE11101010		OUT	DX,AL	BYTE OUTPUT (AL) TO PORT (DX)
EF11101010		OUT	DX,AX	WORD OUTPUT (AX) TO PORT (DX)
F0 11110000		LOCK		BUS LOCK PREFIX
F1 11110001		(not used)		
F2 11110010		REPNZ		REPEAT WHILE (CX)≠0 AND (ZF)=0
F3 11110011		REPN		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)

8086 INSTRUCTION

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								
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	XCHG AX	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**	**MOVS**	**CMPS**	**CMPS**
B	MOV i → AL	MOV i → CL	MOV i → DL	MOV i → BL	MOV i → AH	MOV i → CH	MOV i → DH	MOV i → BH
C			RET, (i+SP)	RET	LES	LDS	MOV b,i,r/m	MOV w,i,r/m
D	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	**IN b**	**IN w**	**OUT b**	**OUT w**
F	LOCK		REP	REP z	HLT	CMC	Grp 1 b,r/m	Grp 1 w,r/m

where:

mod□r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	—	SAR
Grp 1	TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2	INC	DEC	CALL id	CALL l,id	JMP id	JMP l,id	PUSH	—

SET MATRIX

Hi \ Lo	8	9	A	B	C	D	E	F
0	OR b,f,r/m	OR w,f,r/m	OR b,t,r/m	OR w,t,r/m	OR b,i	OR w,i	PUSH CS	
1	SBB b,f,r/m	SBB w,f,r/m	SBB b,t,r/m	SBB w,t,r/m	SBB b,i	SBB w,i	PUSH DS	POP DS
2	SUB b,f,r/m	SUB w,f,r/m	SUB b,t,r/m	SUB w,t,r/m	SUB b,i	SUB w,i	SEG CS	DAS
3	CMP b,f,r/m	CMP w,f,r/m	CMP b,t,r/m	CMP w,t,r/m	CMP b,i	CMP w,i	SEG DS	AAS
4	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6								
7	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	MOV b,f,r/m	MOV w,f,r/m	MOV b,t,r/m	MOV w,t,r/m	**MOV sr,f,r/m**	LEA	**MOV sr,t,r/m**	POP r/m
9	CBW	CWD	CALL l,d	WAIT	PUSHF	POPF	SAHF	LAHF
A	**TEST b,l,a**	**TEST w,l,a**	**STOS**	**STOS**	**LODS**	**LODS**	**SCAS**	**SCAS**
B	MOV i → AX	MOV i → CX	MOV i → DX	MOV i → BX	MOV i → SP	MOV i → BP	MOV i → SI	MOV i → DI
C			RET, I,(i+SP)	RET I	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

b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed. to accum.
id = indirect
is = immed. byte, sign ext.
l = long ie. intersegment
m = memory
r/m = EA is second byte
si = short intrasegment
sr = segment register
t = to CPU reg
v = variable
w = word operation
z = zero

APPENDIX F
PREDEFINED NAMES

DUAL FUNCTION KEYWORD/SYMBOLS

AND	NOT	OR	SHL	SHR	XOR	

SYMBOLS

A	CLD	DX	JC	JNZ	MOV	ROR
AAA	CLI	ES	JE	JO	MOVS	SAHF
AAD	CMC	ESC	JE	JP	MOVSB	SAL
AAM	CMP	FAC	JGE	JPE	MOVSW	SAR
AAS	CMPS	FALC	JL	JPO	MUL	SBB
ADC	CMPSB	HLT	JLE	JS	NEG	SCAS
ADD	CMPSW	IDIV	JMP	JZ	NIL	SCASB
AH	CS	IMUL	JNA	LAHF	OUT	SCASW
AL	CWD	IN	JNAE	LDS	POP	SI
AX	CX	INC	JNB	LEA	POPF	SP
BH	DAA	INT	JNBE	LES	PUSH	SS
BL	DAS	JB	JNC	LOCK	PUSHF	STC
BP	DEC	INTO	JNE	LODS	RCL	STD
BX	DH	IRET	JNG	LODSB	RCR	STI
CALL	DI	JA	JNGE	LODSW	REPE	STOS
CBW	DIV	JAE	JNLE	LOOP	REPNE	STOSB
CH	DL	JB	JNO	LOOPE	REPNZ	STOSW
CL	ES	JBCZ	JNP	LOOPNZ	REPZ	SUB
CLC	DS	JBE	JNS	LOOPZ	RET	TEST
						WAIT
						XCHG
						XLAT
						XLATB
						??SEG

NON-CONFLICTING KEYWORDS

DEBUG	NOPRINT
EJECT	NOSYMBOLS
ERRORPRINT	NOXREF
GEN	OBJECT
GENONLY	PAGELENGTH
INCLUDE	PAGEWIDTH
LIST	PAGING
MEMORY	PRINT
NODEBUG	RESTORE
NOERRORPRINT	SAVE
NOGEN	STACK
NOLIST	SYMBOLS
NOOBJECT	TITLE
NOPAGING	WORKFILES

HANDS-OFF KEYWORDS

ABS	ENDS	LOW	PREFX	STACK
ASSUME	EQ	LT	PROC	THIS
AT	EQU	MASK	PROCLEN	TYPE
BYTE	EVEN	MEMORY	PTR	WIDTH
COMMON	EXTRN	MOD	PUBLIC	WORD
CODEMACRO	FAR	MODRM	PURGE	?
DB	GE	NAME	RECORD	
DD	GROUP	NE	RELB	
DUP	GT	NEAR	RELW	
DW	HIGH	NOTHING	SEG	
DWORD	INPAGE	OFFSET	SEGFIX	
END	LABEL	ORG	SEGMENT	
ENDM	LE	PAGE	SHORT	
ENDP	LENGTH	PARA	SIZE	

APPENDIX G
RELOCATION

Address expressions and numeric expressions may have results which cannot be known until the program has been positioned in memory. These expressions are relocatable. The following rules define (1) when an expression is relocatable and (2) what kind of arithmetic is allowable with relocatable numbers and relocatable address expressions.

NOTE

Associated with every relocatable value is a set of relocation attributes. The assembler tells the R & L system how to calculate the final absolute value via these attributes.

Relocatable Expressions

The following rules define when an expression is relocatable. The EQU facility of the assembler allows a symbol to have as its value the results of a relocatable expression. Therefore, any relocatable expressions may be embodied in a single symbol.

1. Segments and Groups. A segment is considered "non-relocatable" if
   a. It has either PARA or PAGE alignment type and it is not a PUBLIC or STACK segment
   b. It is absolute (i.e., defined via "AT exp").

   A non-relocatable segment has the property that the run-time offset of any byte in the segment is known at assembly time.

   The name of a segment or group may be used in an expression. The name then stands for the paragraph number in 8086 memory space where the segment or group will be located. If a segment is defined via "At exp", then this number is known at assembly time and is "absolute". Otherwise, the paragraph number will not be known until the program has been located by LOC86 (or QRL86) and is "base" relocatable
2. The offset of a variable or label is known at assembly time (called an "absolute" offset) if it meets both these tests:
   a. its containing segment is non-relocatable
   b. it was defined by appearing as a statement label, or to the left of a DB, DW, DD, or LABEL directive, or by an expression of the kind "THIS type"

      The variable's offset is NOT known at assembly time, i.e., is "offset" relocatable, if it fails either (a) or (b). Variables or labels defined by an EXTRN statement always have relocatable offsets, (i.e., are "offset" relocatable).
3. Numbers. A symbol is a number if
   a. it was defined in an EXTRN statement with type ABS, or
   b. it is the name of a group or segment, or
   c. it is defined by EQUating it to an expression evaluating to a number.

   Numbers defined by (a) are always "offset" relocatable, numbers as in (b) are either absolute or "base" relocatable as described in 1 above. Numbers defined via (c) receive the relocation attributes of the expression. Rules governing relocation of expressions are discussed below.

   A number whose value is known at assembly time is called an "absolute" number.

4. Expressions. Expressions evaluate to either a number or an address expression. The rules governing expression evaluation are given in Chapter 5. The following rules define how relocation affects expression evaluation.

   a. The SHORT operator does not affect relocation.

   b. The operators OR, XOR, AND, and NOT may only operate on absolute numbers. The result of one of these operations is always an absolute number.

   c. The relational operators EQ, NE, GT, GE, LT, and LE may have operands which are

      i. both absolute numbers.

      ii. both relocatable numbers. The numbers must have exactly the same relocation attributes.

      iii. Variables and/or Labels. The operands must have exactly the same relocation attributes

   The result of a relational operation is always an absolute Number.

   d. The operators + and −. Two relocatable expressions may never be added. A relocatable expression may appear to the right of "−" if a relocatable expression is on the left, and the two expressions relate as in c above. In this case, the result is always an absolute number. An absolute number may be added to or subtracted from a relocatable expression. A relocatable number may be added to an indexing register. The result has an offset which is identical to the number.

   e. The operators *, /, MOD, SHL, SHR only operate on absolute numbers and the result is always an absolute number.

   f. HIGH and LOW accept either a number or a variable or label as an operand. If the operand is an absolute number, the result is an absolute number. If the operand is a variable or label with an absolute offset, then the result is an absolute number. If the variable or label has a relocatable offset, then the offset is treated as a relocatable number and the following rules apply:

      Let RN be a relocatable number with relocation type

      i. "low", then LOW RN = RN and HIGH RN = 0

      ii. "high", then LOW RN = RN and HIGH RN = 0

      iii. "offset" then LOW RN = RN', which is "low" relocatable and HIGH RN = RN', which is "high" relocatable

      iv. "base" then HIGH and LOW are illegal.

   g. TYPE always returns an absolute number. The operand to the OFFSET operator must be an expression evaluating to a variable or label. If the operand has a relocatable offset, then the result is a relocatable number with the same relocation attributes as the offset. If the operand has an absolute offset, then the result is an absolute number. In either case, the value of the result will equal the operand's offset (i.e., as described in (2) above or PTR and ":" below).

The SEG operator operates on any legal address expression and returns the paragraph number (or segment register) of that address expression. The resulting number is relocatable or absolute as defined in 1 above.

The PTR operator can be used in two ways; the simplest just changes the type attribute of an address expression. No relocation attributes are affected by this action. The other aspect of the PTR operator is to create a variable or label from its two operands. One of the operands must be an absolute number (the type). The other operand represents the offset of the new quantity. This operand may be absolute or any legal relocatable number. Note that this includes "low", "high", or "base"

relocatable numbers, as well as "offset" relocatable numbers. The result of this usage of PTR is a variable or label with no segment part and an offset part which is exactly equal to the offset of the operand, including any relocatable attributes that operand has. This result is not valid in any context except as an operand to the segment override operator, the OFFSET operator, or the TYPE operator.

The SEGMENT OVERRIDE operator, ":" is interpreted as follows:

i. The left operand is restricted to be one of
   1. A segment register
   2. A segment name defined in this module
   3. A group name

ii. The right operand must be an address expression.

iii. If the left operand is a segment register or segment name, the OFFSET of the right operand is first determined and then a new address expression is formed as the result. The segment portion of the result is the left operand. The offset of the result is the OFFSET of the right operand, which may be relocatable.

iv. If the left operand is a group name, then the result has the group as its segment part. The number of bytes from the base of the group to the right operand is the offset of the result. This offset is always relocatable.

The following notation will allow a more exact description of the results from using PTR and the SEGMENT OVERRIDE operator ":". "Vsada" will stand for an address expression. The "s" is its segment part (segment name, group name, segment register, or 0 for undefined). The "d" is its offset part, and "a" is the type (BYTE, WORD, DWORD, NEAR, FAR).

Let d be any number (absolute or relocatable) and a be any valid type. Then

a ptr d = V0da,

an address expression whose offset is exactly equal to d and whose type is a. The segment part is undefined, i.e., the paragraph number to which the offset must be added to obtain a valid 8086 memory address.

Let s be a segment name, let r be a segment register and let g be a group name. Then

s : Vs'da = Vsd'a and
r : Vs'da = Vrd'a

where d' = OFFSET(Vs'da). Morever,

g : Vs'da = Vgd'a,

where d' = OFFSET(Vs'da) + (s' - g) * 16. In this case, d must be either absolute or "offset"-relocatable FROM s'. Furthermore,

g : V0da = Vgda,

which represents an offset of d from the base of g.

A symbol is an absolute number if one of the following applies:

1. it represents the paragraph number of a segment defined by "At exp".
2. it is equated to an expression involving only absolute numbers.
3. it is the OFFSET of a variable or label defined in a non-relocatable segment.
4. it is equated to the comparison or difference of two expressions.
5. it is equated to any expression whose result is always an absolute number, regardles of the types of operands in the expression (e.g., TYPE, LENGTH, SIZE, WIDTH, BYTE, WORD, DWORD, NEAR, FAR)
6. it is equated to the HIGH or LOW of an absolute number or a variable or label as described in 2 above.

A symbol is a relocatable number if and only if it is a number and not absolute.

Assume that Nabs is an absolute number, Nrel is a relocatable number, Vabs is a variable or label whose offset is absolute, Vrel is a variable or label whose offset is relocatable, s is the name of a segment whose paragraph number is not known at assembly time, and g is a group name. The expressions shown in the following list are the only expressions which yield a relocatable result:

EXPRESSION	VALUE
Nrel	Nrel
Vrel	Vrel
g	Nrel
s	Nrel
Nabs + Nrel	Nrel
Nabs + Vrel	Vrel
Nabs + s	Nrel
Nabs + g	Nrel
Nrel + Nabs	Nrel
Vrel + Nabs	Vrel
s + Nabs	Nrel
g + Nabs	Nrel
Nrel – Nabs	Nrel
Vrel – Nabs	Vrel
s – Nabs	Nrel
g – Nabs	Nrel
HIGH Vrel	Nrel
HIGH Nrel	Nrel
LOW Vrel	Nrel
LOW Nrel	Nrel
s : Nabs PTR Nabs'	Vrel
g : Nabs PTR Nabs'	Vrel
s : Nabs PTR Nrel	Vrel
g : Nabs PTR Nrel	Vrel
s : Vabs	Vrel
g : Vabs	Vrel
SEG Vrel	Nrel
OFFSET Vrel	Nrel

The expressions shown in the following list are the expressions which involve relocatable quantities, but always yield an absolute result:

```
Vrel – Vrel
Nrel – Nrel
Vrel r Vrel
Nrel r Nrel
```

where r is one of

```
EQ
NE
GT
GE
LT
LE
```

The result is always Nabs.

APPENDIX H
GETTING STARTED

The primary purpose of this appendix is to show a simple way to get started using ASM86. The information is given in four sections, corresponding to four cases of the size or program code and data (including stack).

Each section summarizes the required and recommended declarations to simplify coding and references to data. Linking with PLM86 is described in the ASM86 Operator's Manual.

The four cases are:

1. total code size less than 64K bytes, total data size less than 64K bytes, total stack size less than 64K bytes
2. total code size greater than 64K, total data and stack each less than 64K
3. total code and stack size each less than 64K, total data size greater than 64K
4. code and data greater than 64K, stack less than 64K

These numbers refer to the sizes after all modules have been linked.

Code, Data, and Stack Sizes Each Less Than 64K

Segments

For the first case, there are 3 types of segments:

1. code, which contains the instructions the program will execute,
2. data, which contains the data being manipulated and
3. stack, which will contain temporary data, procedure parameters, return addresses, etc.

In this case, it is advisable that the final program have only one code segment, one data segment, and one stack segment. There can, however, be many modules which ultimately become linked into this final program.

This situation is completely handled by declaring the segments to have the PUBLIC attribute, as shown below for each type of segment.

Code

The declaration is

```
CODE    SEGMENT     PUBLIC
          o
          o         ; ASM86 instructions
          o
CODE    ENDS
```

The PUBLIC attribute is used to combine all segments of the same name, defined in different modules, into a single final segment for execution.

Data

The declaration is

```
DATA    SEGMENT         PUBLIC
          o
          o           ; data declarations
          o
DATA    ENDS
```

Again, the PUBLIC attribute is used to insure that there will be only one such segment in the final program. This segment will be composed of all segments named DATA from all modules linked together.

Stack

The declaration is

```
STACK   SEGMENT         STACK
          DW            N  DUP  (?)
STACK   ENDS
```

N is the maximum number of stack words used by this module at any one time, e.g., the maximum depth of procedure nesting plus all parameters used by these procedures, plus any data stored temporarily on the stack by any such procedure in this module. The STACK attribute automatically makes this segment public as well.

If this is the main module, then the line preceding this ENDS should read

```
STACK_TOP     LABEL    WORD
```

This enables this main module to initialize the SS and SP registers with the following code:

```
          o
          o
MOV       AX,  STACK
MOV       SS,  AX
MOV       SP,  OFFSET  STACK_TOP
          o
          o
          o
          o
```

This code belongs only in the main module. (LINK86 and LOC86 or QRL86 will correctly adjust the offset of STACK_TOP.) Any other module which uses the stack must use the declaration above, but it is not necessary to declare STACK_TOP. Such a module should not reinitialize SS and SP.

ASSUME Directive

There need be only one ASSUME per module:

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

The ES, DS, and SS registers must be explicitly loaded by your code. Therefore a sample main module skeleton is as follows:

```
ASSUME    CS:CODE,  DS:DATA,  ES:DATA,  SS:STACK
STACK     SEGMENT     STACK
           DW   10   DUP   (?)
STACK_TOP LABEL   WORD
STACK     ENDS
DATA      SEGMENT     PUBLIC
            o
            o
            o
DATA      ENDS

CODE      SEGMENT     PUBLIC
  START:   MOV   AX,  DATA    ; Paragraph # of Data segment to AX
           MOV   DS,  AX      ; then to DS
           MOV   ES,  AX      ; and ES
           MOV   AX,  STACK   ; Paragraph # of Stack segments to AX
           MOV   SS,  AX      ; then to SS
           MOV   SP,  OFFSET  STACK_TOP
                              ; offset of the top of the stack
                              ; to the SP
            o
            o
            o
            o
CODE      ENDS

          END     START
```

Code Greater Than 64K, Data and Stack Each Less Than 64K

The data and stack segments are treated the same as in case 1. There are many optimal methods to organize the code segments. One example would be for each module to have a private code segment. This means each such code segment must have a unique name and must omit the PUBLIC attribute on the segment directive.

Example:

```
    (In module A)
A_CODE      P1  PROC  FAR
                  o
                  o
                  o
A_CODE      ENDS

    (In module B)
B_CODE      SEGMENT
                  o
                  o
            P1  PROC  FAR
                  o
                  o
                  o
                RET
            P1  ENDP
                  o
                  o
                  o
B_CODE      ENDS
```

This will result in all intermodule *jumps* and *calls* being "long" (i.e., FAR). Therefore if a procedure is going to be called from another module it should be FAR.

Total Code and Stack Sizes Each Less Than 64K, Data Size Greater Than 64K

In this case, code and stack segments are handled exactly as they were in case 1. Data segments, however, should be constructed to minimize changing the contents of the DS and ES registers.

This is usually a problem-specific optimization. As an example, the ES register could contain the paragraph number of a segment containing global data, which is referenced from many modules. The ES would remain fixed throughout the program. On the other hand, the DS register could point to a segment containing data local to a module or group of modules. As program control switches to a new module, the DS register would change to point to the local data segment of the new module. The following (non-main) module skeleton is an example of this:

```
ASSUME CS: CODE,  DS: L_DATA,  ES: G_DATA,  SS: STACK

STACK  SEGMENT  STACK
       DW  8  DUP  (?)
            ; Maximum of 8 words of stack used at any one time by this module
STACK  ENDS

PUBLIC    BUFFER,  B_COUNT       ; Global data declared in this module

G_DATA  SEGMENT  PUBLIC
BUFFER          DB    80  DUP  (' ')      ;Buffer initialized to 80 blanks
B_COUNT         DB    ?
G_DATA  ENDS

L_DATA  SEGMENT
                o
                o            ; Data structures local to this module
                o
L_DATA  ENDS

PUBLIC    P                  ; P is a public procedure

CODE  SEGMENT  PUBLIC

P  PROC  NEAR
   PUSH   DS               ; Save the old DS register contents
   MOV    AX, L_DATA       ; Paragraph number of L_data to AX
   MOV    DS, AX           ; and then to DS
    o
    o
    o
   POP    DS               ; restore the DS contents
   RET                     ; return to caller
P  ENDP

CODE  ENDS
      END
```

The segments CODE and G_DATA are public, they will be combined with the other CODE and G_DATA segments respectively from the other modules which comprise the total program. This will also happen for the segment STACK. The segment L_DATA is not public, since the data in that segment is only referenced in this module.

The DS register is saved when this module is entered (presumably by a call to the public procedure P) and restored when this module is exited.

Code and Data and Stack Each Possibly Greater Than 64K Bytes

Code segments will be private as in case 2. Data segments would be handled as above in case 3. Since it is desirable to reduce the overhead of switching segment registers frequently, the design of programs this large should emphasize modularity.

APPENDIX J
INSTRUCTION SET REFERENCE DATA

Table J-1. 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

With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings given in table 2-21. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions, however, is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the SDK-86 or the iSBC 86/12™ board.

Key to Flag Codes:

1 = unconditionally set
0 = unconditionally cleared
X = altered to reflect operation result
U = undefined (mask it out)
R = replaced from memory (e.g., SAHF)
b = (blank) unaffected

AAA	**AAA** (no operands) ASCII adjust for addition			Flags O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAA

AAD	**AAD** (no operands) ASCII adjust for division			Flags O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	60	—	2	AAD

AAM	**AAM** (no operands) ASCII adjust for multiply			Flags O D I T S Z A P C U X X U X U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	83	—	1	AAM

AAS	**AAS** (no operands) ASCII adjust for subtraction			Flags O D I T S Z A P C U U U X U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	AAS

ADC	**ADC** destination,source Add with carry			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADC AX, SI
register, memory	9 + EA	1	2-4	ADC DX, BETA [SI]
memory, register	16 + EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17 + EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

ADD	**ADD** destination,source Addition			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	ADD CX, DX
register, memory	9 + EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16 + EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory, immediate	17 + EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

AND	**AND** destination,source Logical and			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	AND AL,BL
register, memory	9 + EA	1	2-4	AND CX,FLAG_WORD
memory, register	16 + EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	—	3-4	AND CX0,F0H
memory, immediate	17 + EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

CALL	**CALL** target Call a procedure			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FAR_PROC
memptr 16	21 + EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37 + EA	4	2-4	CALL [BX].TASK [SI]

CBW	**CBW** (no operands) Convert byte to word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CBW

CLC	**CLC** (no operands) Clear carry flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLC

CLD	**CLD** (no operands) Clear direction flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLD

CLI	**CLI** (no operands) Clear interrupt flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLI

CMC	**CMC** (no operands) Complement carry flag			Flags O D I T S Z A P C X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CMC

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

CMP

CMP destination,source
Compare destination to source

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

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	CMP BX, CX
register, memory	9+EA	1	2-4	CMP DH, ALPHA
memory, register	9+EA	1	2-4	CMP [BP+2], SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10+EA	1	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

CMPS

CMPS dest-string,source-string
Compare string

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

Operands	Clocks	Transfers*	Bytes	Coding Example
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9+22/rep	2/rep	1	REPE CMPS ID, KEY

CWD

CWD (no operands)
Convert word to doubleword

Flags: O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	5	—	1	CWD

DAA

DAA (no operands)
Decimal adjust for addition

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

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	DAA

DAS

DAS (no operands)
Decimal adjust for subtraction

Flags: O D I T S Z A P C
U X X X X X

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	4	—	1	DAS

DEC

DEC destination
Decrement by 1

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

Operands	Clocks	Transfers*	Bytes	Coding Example
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15+EA	2	2-4	DEC ARRAY [SI]

DIV

DIV source
Division, unsigned

Flags: O D I T S Z A P C
U U U U U U

Operands	Clocks	Transfers*	Bytes	Coding Example
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) +EA	1	2-4	DIV ALPHA
mem16	(150-168) +EA	1	2-4	DIV TABLE [SI]

ESC

ESC external-opcode,source
Escape

Flags: O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
immediate, memory	8+EA	1	2-4	ESC 6,ARRAY [SI]
immediate, register	2	—	2	ESC 20,AL

HLT

HLT (no operands)
Halt

Flags: O D I T S Z A P C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	HLT

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer

IDIV	**IDIV** source Integer division			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) +EA	1	2-4	IDIV DIVISOR BYTE [SI]
mem16	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL	**IMUL** source Integer multiplication			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) +EA	1	2-4	IMUL RATE_BYTE
mem16	(134-160) +EA	1	2-4	IMUL RATE_WORD [BP] [DI]

IN	**IN** accumulator,port Input byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed8	10	1	2	IN AL, 0FFEAH
accumulator, DX	8	1	1	IN AX, DX

INC	**INC** destination Increment by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI] [BX]

INT	**INT** interrupt-type Interrupt			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8 (type = 3)	52	5	1	INT 3
immed8 (type ≠ 3)	51	5	2	INT 67

INTR	**INTR** (external maskable interrupt) Interrupt if INTR and IF=1			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	61	7	N/A	N/A

INTO	**INTO** (no operands) Interrupt if overflow			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	53 or 4	5	1	INTO

IRET	**IRET** (no operands) Interrupt Return			Flags O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	24	3	1	IRET

JA/JNBE	**JA/JNBE** short-label Jump if above/Jump if not below nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JA ABOVE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

JAE/JNB	JAE/JNB short-label Jump if above or equal/Jump if not below			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JAE ABOVE_EQUAL

JB/JNAE	JB/JNAE short-label Jump if below/Jump if not above nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JB BELOW

JBE/JNA	JBE/JNA short-label Jump if below or equal/Jump if not above			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNA NOT ABOVE

JC	JC short-label Jump if carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JC CARRY SET

JCXZ	JCXZ short-label Jump if CX is zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	JCXZ COUNT DONE

JE/JZ	JE/JZ short-label Jump if equal/Jump if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JZ ZERO

JG/JNLE	JG/JNLE short-label Jump if greater/Jump if not less nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JG GREATER

JGE/JNL	JGE/JNL short-label Jump if greater or equal/Jump if not less			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JGE GREATER EQUAL

JL/JNGE	JL/JNGE short-label Jump if less/Jump if not greater nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JL LESS

JLE/JNG	JLE/JNG short-label Jump if less or equal/Jump if not greater			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNG NOT GREATER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer

JMP	JMP target Jump			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN SEGMENT
far-label	15	—	5	JMP FAR LABEL
memptr16	18+EA	1	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24+EA	2	2-4	JMP OTHER_SEG [SI]

JNC	JNC short-label Jump if not carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ short-label Jump if not equal/Jump if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNE NOT_EQUAL

JNO	JNO short-label Jump if not overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNO NO_OVERFLOW

JNP/JPO	JNP/JPO short-label Jump if not parity/Jump if parity odd			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPO ODD_PARITY

JNS	JNS short-label Jump if not sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNS POSITIVE

JO	JO short-label Jump if overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JO SIGNED_OVRFLW

JP/JPE	JP/JPE short-label Jump if parity/Jump if parity even			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPE EVEN_PARITY

JS	JS short-label Jump if sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JS NEGATIVE

LAHF	LAHF (no operands) Load AH from flags			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	LAHF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

LDS	LDS destination,source Load pointer using DS			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LDS SI,DATA.SEG [DI]

LOCK	LOCK (no operands) Lock bus			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	LOCK XCHG FLAG.AL

LODS	LODS source-string Load string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string (repeat) source-string	12 9+13/rep	1 1/rep	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	LOOP short-label Loop			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17/5	—	2	LOOP AGAIN

LOOPE/LOOPZ	LOOPE/LOOPZ short-label Loop if equal/Loop if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	LOOPE AGAIN

LOOPNE/LOOPNZ	LOOPNE/LOOPNZ short-label Loop if not equal/Loop if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	—	2	LOOPNE AGAIN

LEA	LEA destination,source Load effective address			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem16	2+EA	—	2-4	LEA BX, [BP][DI]

LES	LES destination,source Load pointer using ES			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem32	16+EA	2	2-4	LES DI, [BX].TEXT_BUFF

NMI	NMI (external nonmaskable interrupt) Interrupt if NMI = 1			Flags O S I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

MOV	**MOV** destination,source Move			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX,CX
register, memory	8+EA	1	2-4	MOV BP, STACK_TOP
memory, register	9+EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10+EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9+EA	1	2-4	MOV [BX].SEG_SAVE, CS

MOVS	**MOVS** dest-string,source-string Move string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9+17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** (no operands) Move string (byte/word)			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	18	2	1	MOVSB
(repeat) (no operands)	9+17/rep	2/rep	1	REP MOVSW

MUL	**MUL** source Multiplication, unsigned			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	70-77	—	2	MUL BL
reg16	118-133	—	2	MUL CX
mem8	(76-83) +EA	1	2-4	MUL MONTH [SI]
mem16	(124-139) +EA	1	2-4	MUL BAUD_RATE

NEG	**NEG** destination Negate			Flags O D I T S Z A P C X X X X X 1*
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NEG AL
memory	16+EA	2	2-4	NEG MULTIPLIER

*0 if destination = 0

NOP	**NOP** (no operands) No Operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3	—	1	NOP

NOT	**NOT** destination Logical not			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NOT AX
memory	16+EA	2	2-4	NOT CHARACTER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

OR	**OR** destination,source Logical inclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	OR AL, BL
register, memory	9+EA	1	2-4	OR DX, PORT ID[DI]
memory, register	16+EA	2	2-4	OR FLAG BYTE, CL
accumulator, immediate	4	—	2-3	OR AL,0110110B
register, immediate	4	—	3-4	OR CX,01FH
memory, immediate	17+EA	2	3-6	OR [BX].CMD WORD,0CFH

OUT	**OUT** port,accumulator Output byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8, accumulator	10	1	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP	**POP** destination Pop word off stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17+EA	2	2-4	POP PARAMETER

POPF	**POPF** (no operands) Pop flags off stack			Flags O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	8	1	1	POPF

PUSH	**PUSH** source Push word onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16+EA	2	2-4	PUSH RETURN CODE[SI]

PUSHF	**PUSHF** (no operands) Push flags onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	10	1	1	PUSHF

RCL	**RCL** destination,count Rotate left through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCL CX, 1
register, CL	8+4/bit	—	2	RCL AL, CL
memory, 1	15+EA	2	2-4	RCL ALPHA, 1
memory, CL	20+EA+4/bit	2	2-4	RCL [BP].PARM, CL

RCR	**RCR** designation,count Rotate right through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCR BX, 1
register, CL	8+4/bit	—	2	RCR BL, CL
memory, 1	15+EA	2	2-4	RCR [BX].STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	RCR ARRAY[DI], CL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

REP	**REP** (no operands) Repeat string operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REP MOVS DEST, SRCE

REPE/REPZ	**REPE/REPZ** (no operands) Repeat string operation while equal/while zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPE CMPS DATA, KEY

REPNE/REPNZ	**REPNE/REPNZ** (no operands) Repeat string operation while not equal/not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPNE SCAS INPUT LINE

RET	**RET** optional-pop-value Return from procedure			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(intra-segment, no pop)	8	1	1	RET
(intra-segment, pop)	12	1	3	RET 4
(inter-segment, no pop)	18	2	1	RET
(inter-segment, pop)	17	2	3	RET 2

ROL	**ROL** destination,count Rotate left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Examples**
register, 1	2	—	2	ROL BX, 1
register, CL	8+4/bit	—	2	ROL DI, CL
memory, 1	15+EA	2	2-4	ROL FLAG BYTE [DI], 1
memory, CL	20+EA+4/bit	2	2-4	ROL ALPHA , CL

ROR	**ROR** destination,count Rotate right			Flags O D I T S Z A P C X X
Operand	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	ROR AL, 1
register, CL	8+4/bit	—	2	ROR BX, CL
memory, 1	15+EA	2	2-4	ROR PORT STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	ROR CMD WORD, CL

SAHF	**SAHF** (no operands) Store AH into flags			Flags O D I T S Z A P C R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	SAHF

SAL/SHL	**SAL/SHL** destination,count Shift arithmetic left/Shift logical left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
register,1	2	—	2	SAL AL,1
register, CL	8+4/bit	—	2	SHL DI, CL
memory,1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20+EA+4/bit	2	2-4	SAL STORE_COUNT, CL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

SAR	**SAR** destination,source Shift arithmetic right			Flags O D I T S Z A P C X X X U X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SAR DX, 1
register, CL	8+4/bit	—	2	SAR DI, CL
memory, 1	15+EA	2	2-4	SAR N BLOCKS, 1
memory, CL	20+EA+4/bit	2	2-4	SAR N BLOCKS, CL

SBB	**SBB** destination,source Subtract with borrow			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SBB BX, CX
register, memory	9+EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16+EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS	**SCAS** dest-string Scan string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9+15/rep	1/rep	1	REPNE SCAS BUFFER

SHR	**SHR** destination,count Shift logical right			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SHR SI, 1
register, CL	8+4/bit	—	2	SHR SI, CL
memory, 1	15+EA	2	2-4	SHR ID BYTE [SI] [BX], 1
memory, CL	20+EA+4/bit	2	2-4	SHR INPUT WORD, CL

SINGLE STEP	**SINGLE STEP** (Trap flag interrupt) Interrupt if TF = 1			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

STC	**STC** (no operands) Set carry flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STC

STD	**STD** (no operands) Set direction flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STD

STI	**STI** (no operands) Set interrupt enable flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STI

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

STOS	**STOS** dest-string Store byte or word string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

SUB	**SUB** destination,source Subtraction			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SUB CX, BX
register, memory	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]
memory, register	16+EA	2	2-4	SUB [BP+2], CL
accumulator, immediate	4	—	2-3	SUB AL, 10
register, immediate	4	—	3-4	SUB SI, 5280
memory, immediate	17+EA	2	3-6	SUB [BP].BALANCE, 1000

TEST	**TEST** destination,source Test or non-destructive logical and			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	TEST SI, DI
register, memory	9+EA	1	2-4	TEST SI, END_COUNT
accumulator, immediate	4	—	2-3	TEST AL, 00100000B
register, immediate	5	—	3-4	TEST BX, 0CC4H
memory, immediate	11+EA	—	3-6	TEST RETURN_CODE, 01H

WAIT	**WAIT** (no operands) Wait while $\overline{\text{TEST}}$ pin not asserted			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3 + 5n	—	1	WAIT

XCHG	**XCHG** destination,source Exchange			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, reg16	3	—	1	XCHG AX, BX
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX
register, register	4	—	2	XCHG AL, BL

XLAT	**XLAT** source-table Translate			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-table	11	1	1	XLAT ASCII_TAB

XOR	**XOR** destination,source Logical exclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	XOR CX, BX
register, memory	9+EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16+EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

APPENDIX K
SAMPLE PROGRAM

```
MCS-86 MACRO ASSEMBLER    SAMPLE                                                PAGE     1


ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE SAMPLE
OBJECT MODULE PLACED IN :F1:SAMPLE.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:SAMPLE.A86 PL(255) XREF

LOC  OBJ                  LINE     SOURCE

                             1     ;*************************************************************************
                             2     ;
----                         3     DATA2          SEGMENT
                             4     ;
0000 ????                    5             TEMP     DW    ?              ; 1 word, indeterminate contents
0002 (100                    6             FOO      DW    100 DUP (?)    ; 100 words, indeterminate contents
     ????
     )
                             7     ;
                             8     ;
                             9     ;-------------------------------------------------------------------------
                            10     ;                       Defining a STRUCTURE
                            11     ;-------------------------------------------------------------------------
                            12     ; Define a structure template PANACHE with fields FLDA, FLDB, ..., LFDD,
                            13     ; such that their types are ordered BYTE, WORD, WORD, BYTE.
----                        14             PANACHE  STRUC                ; Template def'n., no storage reserved
0000                        15               FLDA   DB     ?            ; Reference to .FLDA gives 0
0001                        16               FLDB   DW     ?            ; Reference to .FLDB gives 1
0003                        17               FLDC   DW     ?            ; Reference to .FLDC gives 3
0005                        18               FLDD   DB     ?            ; Reference to .FLDD gives 5
----                        19             PANACHE  ENDS                 ; End of structure template definition
                            20     ;
                            21     ;-------------------------------------------------------------------------
                            22     ;                       Initializing a STRUCTURE ARRAY
                            23     ;-------------------------------------------------------------------------
                            24     ; Now use structure template name PANACHE as an operator to initialize
                            25     ; 100 copies of 6-byte structure ELAN = ELAN[0], ELAN[6], ... , ELAN[594]
                            26     ; each with contents initialized in the order shown
                            27     ;
00CA (100                   28             ELAN     PANACHE  100 DUP (<?, , 5*5+4, 0FEH>)
     ??
     ????
     1D00
     FE
     )
                            29     ;
                            30     ; -------------------------------------------------------------------------
                            31     ;                       Defining a RECORD
                            32     ;-------------------------------------------------------------------------
                            33     ; Define a record, VERVE, 16 bits long with fields of 4 bits, 5 bits, 7 bits
                            34     ;
   #                        35             VERVE    RECORD   DIBSA:4, DIBSB:5, DIBSC:7  ; No storage reserved
                            36     ;
                            37     ;-------------------------------------------------------------------------
                            38     ;                       Initializing a RECORD ARRAY
                            39     ;-------------------------------------------------------------------------
                            40     ; Now use record definition name VERVE as operator to initialize
                            41     ; 100 copies of 2-byte record ESPRIT = ESPRIT[0], ... , ESPRIT[198]
                            42     ; Initialize fields in each 2-byte copy to 13, 29, and 101...
                            43
0322 (100                   44             ESPRIT   VERVE    100 DUP (<13, 29, 101>)     ; 200 bytes reserved
     E5DE
     )
                            45     ;
                            46     ;-------------------------------------------------------------------------
----                        47     DATA2          ENDS                                   ; End of segment DATA2
                            48     ;    (CONTINUED ON NEXT PAGE -- K-2)
                            49 +1  $EJECT
```

```
MCS-86 MACRO ASSEMBLER    SAMPLE                                                PAGE    2


LOC  OBJ                  LINE     SOURCE

                            50     ;   (CONTINUED FROM PRECEDING PAGE -- K-1)
                            51
                            52     ;*************************************************************************
                            53     ;
----                        54     STACK3          SEGMENT
0000 (1000                  55                       DW     1000 DUP (?)  ; Reserve 1000 uninitialized words
     ????
     )
07D0                        56          STACK_BTM   LABEL WORD            ; Stack grows toward low memory
----                        57     STACK3          ENDS                   ; End of stack segment
                            58     ;
                            59     ;*************************************************************************
                            60     ;
----                        61     STACK_INIT      SEGMENT
                            62                     ASSUME   CS:STACK_INIT   ; 16*CS + IP is current instr. addr.
0000                        63     INITIALIZE      PROC     FAR                    ;
0000 B8----          R      64                     MOV      AX, STACK3             ; Load AX with stack seg. base
0003 8ED0                   65                     MOV      SS, AX                 ; Load SS seg-reg with seg. base
0005 BCD007                 66                     MOV      SP, OFFSET STACK_BTM   ; Initialize stack pointer SP
0008 CB                     67                     RET
                            68     INITIALIZE      ENDP                           ; End of procedure INITIALIZE
----                        69     STACK_INIT      ENDS                           ; End of segment STACK_INIT
                            70     ;
                            71     ;*************************************************************************
                            72     ;
                            73                   EXTRN    DIDDLE:FAR
----                        74     ILLUSTRATION    SEGMENT
                            75                     ASSUME   CS:ILLUSTRATION,       ; 16*CS+IP=curr. instr. addr.
                            76     &                       DS:DATA2        ; 16*DS + Offset is data item addr.
                            77     ;
                            78     ;-------------------------------------------------------------------------
                            79     ; COMPUTE is a procedure (callable from another segment) which adds
                            80     ; the 3rd word in array FOO,
                            81     ; to the sum of all the FLDCs in ELAN,
                            82     ; then subtracts the sum of all the record-fields DIBSB in record array ESPRIT
                            83     ;
0000                        84          COMPUTE    PROC  FAR               ; FAR means callable from another seg.
                            85     ;
                            86     ;-------------------------------------------------------------------------
                            87     ;
                            88     ; Put 3rd word from array FOO into AX as 1st addend
                            89     ;
0000 A10600                 90                     MOV   AX, FOO[4]       ; Load AX with 3rd word from FOO
                            91     ;
                            92     ;-------------------------------------------------------------------------
                            93     ;                       Summing the STRUCTURE ARRAY Fields
                            94     ;-------------------------------------------------------------------------
                            95     ;
                            96     ; This next section adds in all the .FLDC fields from the structure array ELAN.
                            97     ; First we set SI=0 to index the first field to be added.
                            98     ; Then we use the index [SI] to add each such field into AX,
                            99     ; and increment the index [SI] each time by the size of structure PANACHE,
                           100     ; which is the TYPE of the array ELAN.
                           101     ; Loop control for the addition is provided by CX,
                           102     ; which is loaded with the length (100, the number of elements) of ELAN.
                           103     ;
0003 BE0000                104                     MOV   SI, 0                    ; Use SI for array index
0006 B96400                105                     MOV   CX, LENGTH ELAN          ; Loop control = number elements
0009 0384CD00              106             SUM:    ADD   AX,ELAN[SI].FLDC         ; Add a FLDC value to AX
000D 83C606                107                     ADD   SI, TYPE ELAN            ; Update index to next FLDC
0010 E2F7                  108                     LOOP  SUM                      ; Loop to SUM if CX<>0
                           109     ;
                           110     ;-------------------------------------------------------------------------
                           111     ;                     Summing the RECORD ARRAY Fields
                           112     ;-------------------------------------------------------------------------
                           113     ;
                           114     ; First, the beginning of the array will be addressed by [SI] (SI=0).
                           115     ; Each DIBSB field will be isolated and right-justified (aligned) in DX
                           116     ; as follows:   the operator MASK masks out the unwanted bits,
                           117     ; and the field-name DIBSB gives the shift-count needed to align the field.
                           118     ;
0012 BE0000                119                      MOV SI, 0                     ; Initialize SI to index first record
0015 BF6400                120                      MOV DI, LENGTH ESPRIT         ; Loop control for adding
0018 8B942203              121          GETRECORD:  MOV DX, ESPRIT[SI]            ; Load current record into DX
001C 81E2800F              122                      AND DX, MASK DIBSB            ; Set irrelevant bits to 0
0020 B107                  123                      MOV CL, DIBSB                 ; Field-name is shift count
0022 D3EA                  124                      SHR DX, CL                    ; Right-justify field in DX
0024 2BC2                  125                      SUB AX, DX                    ; Subtract this field from sum
0026 83C602                126                      ADD SI, TYPE ESPRIT           ; Bump SI to index next record
0029 4F                    127                      DEC DI                        ; Loop control
002A 75EC                  128                      JNZ GETRECORD                 ; Get another if DI<>0
                           129     ;
                           130     ;-------------------------------------------------------------------------
                           131     ;
                           132     ; Push the grand total on the stack, and call DIDDLE.
                           133     ;
002C 50                    134                      PUSH AX
002D 9A0000----      E     135                      CALL DIDDLE                   ; Need EXTRN for DIDDLE
                           136     ;
                           137     ;-------------------------------------------------------------------------
                           138     ;
                           139     ; When DIDDLE returns, we're done, and return to the calling routine.
                           140     ;
0032 CB                    141                      RET
                           142        COMPUTE      ENDP                          ; End of procedure COMPUTE
----                       143     ILLUSTRATION    ENDS                          ; End of segment ILLUSTRATION
                           144                     END                           ; End of assembly
```

```
MCS-86 MACRO ASSEMBLER    SAMPLE                                    PAGE    3


XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME            TYPE     VALUE  ATTRIBUTES, XREFS

??SEG . . . .   SEGMENT         SIZE=0000H PARA PUBLIC
COMPUTE . . .   L FAR    0000H  ILLUSTRATION  84# 142
DATA2 . . . .   SEGMENT         SIZE=03EAH PARA    3# 47 76
DIBSA . . . .   R FIELD  0CH    VERVE WIDTH=4  35#
DIBSB . . . .   R FIELD  07H    VERVE WIDTH=5  35# 122 123
DIBSC . . . .   R FIELD  00H    VERVE WIDTH=7  35#
DIDDLE. . . .   L FAR    0000H  EXTRN  73# 135
ELAN. . . . .   V    6   00CAH  DATA2  28# 105 106 107
ESPRIT. . . .   V WORD   0322H  DATA2  44# 120 121 126
FLDA. . . . .   V BYTE   0000H  S FIELD  15#
FLDB. . . . .   V WORD   0001H  S FIELD  16#
FLDC. . . . .   V WORD   0003H  S FIELD  17# 106
FLDD. . . . .   V BYTE   0005H  S FIELD  18#
FOO . . . . .   V WORD   0002H  DATA2  6# 90
GETRECORD . .   L NEAR   0018H  ILLUSTRATION  121# 128
ILLUSTRATION.   SEGMENT         SIZE=0033H PARA    74# 75 143
INITIALIZE. .   L FAR    0000H  STACK_INIT  63# 68
PANACHE . . .   STRUC           SIZE=0006H  #FIELDS=4  14 19# 28
STACK_BTM . .   V WORD   07D0H  STACK3  56# 66
STACK_INIT. .   SEGMENT         SIZE=0009H PARA    61# 62 69
STACK3. . . .   SEGMENT         SIZE=07D0H PARA    54# 57 64
SUM . . . . .   L NEAR   0009H  ILLUSTRATION  106# 108
TEMP. . . . .   V WORD   0000H  DATA2  5#
VERVE . . . .   RECORD          SIZE=2 WIDTH=16  35# 44

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX L
MACRO PROCESSOR LANGUAGE

This appendix is intended as a supplementary reference for the macro language and as a guide to more advanced use of the macro processor. It is assumed that the reader is already familiar with the introductory material on macros presented in Chapter 7.

Terminology and Conventions

A percent sign will be used as the Metacharacter throughout this appendix although the user may temporarily change the metacharacter by using the METACHAR function.

The term "logical blank" refers to a blank, horizontal tab, carriage return, or linefeed character.

Throughout the appendix the term "parameter" refers to what are sometimes known as "dummy parameters" or "formal parameters" while the term "argument" is reserved for what are sometimes known as "actual parameters". The terms "Normal" and "Literal", names for the two fundamental modes used by the macro processor in reading characters, will be capitalized in order to distinguish these words from their ordinary usage.

In the syntax diagrams, non-terminal syntactic types are represented by lower case words, sometimes containing the break character, "_". If a single production contains more than one instance of a syntactic type each instance may be followed by a unique integer so that the prose description may unambiguously refer to each occurrence. Opening and closing quotes (" and ") are used to refer to literally-coded character sequences.

Basic Elements of the Macro Language

Identifiers

With the exception of some built-in functions, all macro processor functions begin with an identifier, which names the function. Parameters also are represented by identifiers. A macro processor identifier has the following syntax.

id = alphabetic | id id_continuation

The alphabetic characters include upper and lower case letters. An id_continuation character is an alphabetic character, a decimal digit, or the break character ("_").

Examples:

```
WHILE Add_2 MORE_TO_DO Much_More
```

An identifier must not be split across the boundary of a macro and may not contain Literal characters.

For example,

```
"%%(FOO)"
```

is illegal. The first metacharacter is followed by the letters "FOO", but they do not constitute an identifier since they are Literal characters.

```
"%ADD%SUFFIX"
```

where SUFFIX is defined as "UP" is a call to ADD followed by a call to SUFFIX, rather than a call to ADDUP, because identifiers may not cross macro boundaries.

A null-string bracket or escape function ("%()" or "%0") will also end an identifier, and since these functions have no textual value themselves, may be used as separators.

Example:

```
" %TOM%0SMITH "
```

concatenates the value of the macro, TOM, to the string, "SMITH".

This could also be done by writing, "%TOM%(SMITH)". Upper and lower case letters are equivalent in their use in identifiers. ("CAT", "cat", and "cAt" are equivalent.)

Text and Delimiters

"Text" is an undistinguished string of characters. It may or may not contain items of significance to the macro processor. In general the MPL processor simply copies characters from its input to its output stream. This copying process continues until an instance of the metacharacter is encountered, whereupon the macro processor begins analyzing the text that follows.

Each macro function has a calling pattern that must match the text in an actual macro function call. The pattern consists of text strings, which are the arguments to the function, and a number of delimiter strings.

For example,

```
"JOIN( FIRST, SECOND )"
```

might be a pattern for a macro, JOIN, which takes two arguments. The first argument will correspond to the parameter, FIRST, and the second to the parameter, SECOND. The delimiters of this pattern are "(", ",", and ")".

A text string corresponding to a parameter in the pattern must be balanced with respect to parentheses (see below). A delimiter which follows a parameter in the pattern will be used to mark the end of the argument in an actual call to the macro.

An argument text string is recognized by finding the specific delimiter that the pattern indicates will end the string. A text string for a given argument consists of the characters between the delimiter (or macro identifier) that precedes the text and the delimiter which follows the text.

In the case of built-in functions, there are sometimes additional requirements on the syntax of an argument. For example, the text of an argument might be required to conform to the syntax for a numeric expression.

Balanced Text

Arguments must be balanced with respect to left and right parentheses in the usual manner of requiring that the string contain the same number of left and right parentheses and that at no point during a left to right scan may there have been more right parentheses than left parentheses. (An "unbalanced" parenthesis may be quoted with the escape function to make the text meet this requirement.)

Expressions

Balanced text strings appearing in certain places in built-in macro processor functions are interpreted as numeric expressions:

1. As arguments that control the execution of "IF", "WHILE", "REPEAT", and "SUBSTR" functions.
2. As the argument to the evaluate function, "EVAL".

Operators (in order of precedence from high to low):

```
Parenthesized Expressions
HIGH  LOW
*  /  MOD SHL SHR
+     -
EQ    LT  LE  GT  GE  NE
NOT
AND
OR    XOR
```

All arithmetic is performed in an internal format of 17-bit two's complement integers.

The Macro Processor Scanning Algorithm

Literal or Normal Mode of Expansion

At any given time, the macro processor is reading text in one of two fundamental modes. When processing of the primary input file begins, the mode is Normal. Normal mode means that macro calls will be expanded, i.e. the metacharacter in the input will cause the following macro function to be executed.

In the simplest possible terms, Literal mode means that characters are read Literally, i.e. the text is not examined for function calls. The text read in this mode is similar to the text inside a quoted character string familiar to most users of high level languages; that is, the text is considered to be merely a sequence of characters having no semantic weight. There are important exceptions to this very simple view of the Literal mode. If the characters are being read from a user-defined macro with parameters, the parameter references will be replaced with the corresponding argument values regardless of the mode. The Escape function and the Comment function will also be recognized in either mode.

The mode can change when a macro is called. For user-defined macros, the presence or absence of the call-literally character following the metacharacter sets the mode for the reading of the macro's value. The arguments to a user-defined macro are evaluated in the Normal mode, but when the processor begins reading the macro's value, the mode changes to that indicated by the call. When the processor finishes reading the macro's definition, the mode reverts to what it was before the macro's processing began.

To illustrate, suppose the parameterless macros, CAT and TOM are defined as follows.

CAT is: "abcd %TOM efgh", and TOM is: "xyz"

Now consider the text fragment,

"... %CAT, %*CAT ..."

Assume the string is being read in the Normal mode. The first call to CAT is recognized and called Normally. Since CAT is called Normally, the definition of CAT is examined for macro calls as it is read. Thus the characters "%TOM" in the definition for CAT are recognized as a macro call and so TOM is expanded Normally. The definition for TOM is read, but it contains no macro calls. After the definition for TOM is processed, the mode reverts back to its value in reading CAT (Normal). After the definition of CAT is processed the mode reverts back to its original value (Normal). At this point, immediately before processing the comma following the first call to CAT, the value of the text fragment processed thus far is:

"... abcd xyz efgh"

Now the processor continues reading Normally, finally encountering the second call to CAT, this time a Literal call. The mode changes to Literal as the definition of CAT is read. This time the characters from the definition are read Literally. When the end of the definition of CAT is reached the mode reverts to its original value (Normal) and processing continues. The value of the entire fragment is,

"... abcd xyz efgh, abcd %TOM efgh ...".

The use of the call-literally character on calls to builtin macro functions is discussed in the description of each function. The important thing to keep in mind when analyzing how a piece of text is going to be expanded is the Normal or Literal Mode of the environment in which it is read.

The Call Pattern

In general, each macro function has a distinctive name which follows the metacharacter (and possibly the call-literally character). This name is usually an identifier, although a few built-in functions have other symbols for names. For identifier named functions, the macro processor allows the identifier to be the result of another macro call.

For example, suppose the macro, NAME, has the value "BIGMAC" and that the macro BIGMAC has the calling pattern, "BIGMAC X & Y;". Then the call,

"... %%NAME catsup & mustard; ..."

is a call to the macro BIGMAC with the first argument having the value, " catsup " and the second argument having the value, " mustard".

Associated with this name is, possibly, a pattern of delimiters and parameters which must be matched if the macro call is to be syntactically correct. The pattern for each builtin macro function is described in the section of this appendix dealing with that function. The pattern for a user-defined macro is defined at the time the macro is defined.

At the time of a macro call, the matching of text to the pattern occurs by using the delimiters one at a time, left to right. When a delimiter is located, the next delimiter of the pattern becomes the new goal. The delimiters in the call are separated by either argument text (if there was a corresponding parameter in the macro's definition pattern), or by any number of logical blanks (in the case of adjacent delimiters in the pattern). The argument text corresponding to a parameter in the definition pattern becomes the value of the parameter for the duration of the macro's expansion. Null arguments are permitted.

See the section "Macro Definition and Invocation" for more information on delimiters and their relationship to argument strings.

Evaluation of Arguments—Parameter Substitution

MPL uses "call-by-immediate-value" as the ordinary scheme for argument evaluation. This means that as the text is being scanned for the delimiter which marks the end of an argument, any macro calls will be evaluated as they are encountered. In order to be considered as a possible delimiter, characters must all be on the same level of macro nesting as the metacharacter which began the call. In other words, the arguments to a macro can be any mixture of plaintext and macro calls, but the delimiters of a call must be plaintext.

For example, suppose STRG is defined as "dogs,cats" and MAC1 is a macro with the calling pattern, "MAC1(P1, P2)". Then in the call,

"... %MAC1(%STRG, mouse) ..."

the first argument will be " dogs,cats" and the second argument will be " mouse". The comma in the middle of the first argument is not taken as the delimiter because it is on a different level from the metacharacter which began the call to MAC1.

When all arguments of a macro have been evaluated, the expansion of the body begins, with characters being read either Normally or Literally as discussed under "Literal or Normal Mode of Expansion". One should keep in mind that parameter substitution is a high priority function, i.e. arguments will be substituted for parameters even if the macro has been called Literally.

The Evaluate Function

The syntax for the Evaluate function is:

evaluate_function = "EVAL" "(" expr ")"

The single argument is a text string which will be evaluated as a numeric expression, with the result returned as a text string.

Examples:

%EVAL(7) evaluates to "07H"

%EVAL((7+3)*2) evaluates to "14H"

If NUM has the value "0101B" then %EVAL(%NUM - 5) evaluates to "00H"

Numeric Functions: LEN, and String Compare Functions

These functions take text string arguments and return some numeric information in the form of hexidecimal integers.

length_function = "LEN" "(" balanced_text ")"

string_compare_function = op_code "(" balanced_text "," balanced_text ")"

op_code = EQS | GTS | LTS | NES | GES | LES

The length numeric function returns an integer equal to the number of characters in the text string. The string comparison functions all return the character representation for minus one if the relation between the strings holds, or zero otherwise. These relations are for string compares. These functions should not be confused with the arithmetic compare operators that might appear in expressions. The ASCII code for each character is considered a binary number and represents the relative value of the character. "Dictionary" ordering is used: Strings differing first in their Nth character are ranked according to the Nth character. A string which is a prefix of another string is ranked lower than the longer string.

The Bracket Function

The bracket function is used to introduce literal strings into the text and to prevent the interpretation of functions contained therein, (except the high priority functions: comment, escape, and parameter substitution). A call-literally character is not allowed; the function is always called Literally.

bracket_function = "(" balanced_text ")"

The value of the function is the value of the text between the matching parentheses, evaluated Literally. The text must be balanced with respect to left and right parentheses. (An unbalanced left or right parenthesis may be quoted with the escape function.) Text inside the bracket function that would ordinarily be recognized as a function call is not recognized; thus, when an argument in a macro call is put inside a bracket function, the evaluation of the argument is delayed—it will be substituted as it appears in the call (but without the enclosing bracket function).

The null string may be represented as %().

Examples:

```
%(This is a string.)          %( %EVAL( %NUM ) )
evaluates to:                 evaluates to:
"This is a string."           " %EVAL( %NUM ) "
```

The Escape Function

The escape function provides an easy way to quote a few characters to prevent them from having their ordinary interpretation. Typical uses are to insert an "unbalanced" parenthesis into a balanced text string, or to quote the metacharacter. The syntax is:

```
escape_function = /* A single digit, 0 through 9, followed by
                      that many characters. */
```

The call-literally character may not be present in the call. The escape function is a high priority function, that is one of the functions (the others are the comment functions and parameter substitution) which are recognized in both Normal and Literal mode.

Examples:

"... %2%% ..." evaluates to "... %% ..."
"... %(ab%1)cd) ..." evaluates to "... ab)cd ..."

Macro Definition and Invocation

The macro definition function associates an identifier with a functional string. The macro may or may not have an associated pattern consisting of parameters and/or delimiters. Also optionally present is a list of local symbols. The syntax for a macro definition is:

```
macro_def_function =
    "DEFINE" "(" macro_id define_pattern ")" [ "LOCAL" id_list ]
    "(" balanced_text ")"
```

The define_pattern is a balanced string which is further analyzed by the macro processor as follows:

```
define_pattern = [ parm_id ] [ delimiter_specifier ]

delimiter_specifier =    /*String not containing non-Literal
                           id_continuation, logical blank, or
                           "@" characters. */

                       | "@" delimiter_id
```

The syntax for a macro invocation is as follows:

```
macro_call = macro_id [ call_pattern ]
call_pattern = /* Pattern of text and delimiters
                  corresponding to the definition
                  pattern. */
```

As seen above, the macro_id optionally may be defined to have a pattern, which consists of parameters and delimiters. The presence of this define pattern specifies how the arguments in the macro call will be recognized. Three kinds of delimiters may be specified in a define pattern. Literal and Identifier delimiters appear explicitly in the define pattern, while Implied Blank delimiters are implicit where a parameter in the define pattern is not followed by an explicit delimiter. Literal delimiters are the most common and typically include commas, parentheses, other punctuation marks, etc. Id delimiters are delimiters that look like and are recognized like identifiers. The presence of an Implied Blank delimiter means that the preceding argument is terminated by the first logical blank encountered. We will examine these various forms of delimiter in greater detail later in this description.

Recognition of a macro name (which uniquely identifies a macro) is followed by the matching of the call pattern to the define pattern. The two patterns must match for the call to be well-formed. It must be remembered that arguments are balanced strings, thus parentheses can be used to prevent an enclosed substring from being matched with a delimiter. The strings in the call pattern corresponding to the parameters in the define pattern become the values of those parameters.

Reuse of the name for another definition at a later time will replace a previous definition. Built-in macro processor functions (as opposed to user-defined macros) may not be redefined. A macro may not be redefined during the evaluation of its own body. A parameter may not be redefined within the body of its macro.

Parameters appearing in the body of a macro definition (as parameter substitution functions) are preceded by the metacharacter. When the body is being expanded after a call, the parameter substitution function calls will be replaced by the value of the corresponding arguments.

The evaluation of the balanced_text that defines the body of the macro being defined is evaluated in the mode specified by the presence or absence of the call literally character on the call to DEFINE. If the DEFINE function is called Normally, the balanced text is evaluated in the Normal mode before it is stored as the macro's value. If the define function is called Literally, the balanced_text is evaluated Literally before it is stored.

Literal Delimiters

A Literal delimiter which contains id_continuation characters, "@", or logical blanks must be quoted by a bracket function, escape function, or by being produced by a Literal call. Other literal delimiters need not be quoted in the define pattern.

Example 1:

```
%*DEFINE ( SAY(ANIMAL,COLOR) ) (THE %ANIMAL IS %COLOR.)
%SAY(HORSE,TAN)
```

produces,

```
THE HORSE IS TAN.
```

Example 2:

```
%*DEFINE ( REVERSE [ P1 %(.AND.) P2 ] ) (%P2 %P1)
%REVERSE [FIRST.AND.SECOND]
```

produces,

```
SECOND FIRST
```

Id Delimiters

Id delimiters are specified in the define pattern by using a delimiter_specifier having the form, "@ id". The following example should make the distinction between literal and identifier delimiters clear. Consider two delimiter_specifiers, "%(AND)" and "@AND " (the first a Literal delimiter and the second an Id delimiter), and the text string,

```
"... GRAVEL, SAND AND CINDERS ..."
```

Using the first delimiter specifier, the first "AND", following the letter "S", would be recognized as the end of the argument. However, using the second delimiter specifier, only the second "AND" would match, because the second delimiter is recognized like an identifier. Another example:

Definition:

```
%*DEFINE ( ADD P1 @TO P2 @STORE P3. )
(       MOV   %P1
        ADD   AL,%P2
        MOV   %P3,AL
)
```

Macro call:

```
%ADD TOTAL1 TO TOTAL2 STORE GRAND.
```

Generates:

```
MOV   TOTAL1
ADD   AL, TOTAL2
MOV   GRAND
```

Implied Blank Delimiters

If a parameter is not followed by an explicit Literal or Id delimiter then it is terminated by an Implied Blank delimiter. A logical blank is implied as the terminator of the argument corresponding to the preceding parameter. In this case any logical blank in the actual argument must be literalized to prevent its being recognized as the end of the argument. In scanning for an argument having this kind of delimiter, leading non-literal logical blanks will be discarded and the first following non-literal logical blank will terminate the argument.

Example:

```
%*DEFINE ( SAY ANIMAL COLOR ) (THE %ANIMAL IS %COLOR.)
```

The call,

```
%SAY HORSE TAN
```

will evaluate to,

```
THE HORSE IS TAN.
```

In designating delimiters for a macro one should keep in mind the text strings which are likely to appear as arguments. One might base the choice of delimiters for the define pattern on whether the arguments will be numeric, strings of identifiers, or may contain embedded blanks or punctuation marks.

LOCAL Macros and Symbols

The LOCAL option can be used to designate a list of identifiers (separated by blanks) that will be used within the scope of the macro for local macros. A reference to a LOCAL identifier of a macro occuring after the expansion of the text of the macro has begun and before the expansion of the macro is completed will be a reference to the definition of this local macro. Every time a macro having the LOCAL option is called, a new incarnation of the listed symbols is created. The local symbols thus have dynamic, inclusive scope.

At the time of the call to a macro having locals, the local symbols are initialized to a string whose value is the symbol name concatenated with a unique number. The number is generated by incrementing a counter each time a local declaration is made.

Definition:

```
%*DEFINE (MAC1 (FIRST,SECOND,THIRD)) LOCAL LABL
(%LABL:   MOV   BX,%FIRST
          MOV   AX,[BX]
          MOV   BX,%SECOND
          MOV   CX,[BX]
          MOV   BX,%THIRD
          MOV   DX,[BX] )
```

Macro call:

```
%MAC1(ITEM,NEXT,ANOTHER)
```

Generates: (Typically, depending on value for local, "LABL")

```
LABL3:    MOV   BX,ITEM
          MOV   AX,[BX]
          MOV   BX,NEXT
          MOV   CX,[BX]
          MOV   BX,ANOTHER
          MOV   DX,[BX]
```

The Control Functions: IF, REPEAT, and WHILE

These functions can be used to alter the flow of control in a sense analogous to that of their similarly named counterparts in procedural languages; however, they are different in that they may be used as value generating functions as well as control statments.

The three functions all have a "body" which is analogous to the defined value, or body, of a user-defined macro function. The syntax of these functions is:

if_function = "IF" "(" expr ")" "THEN" "(" body ")" ["ELSE" "(" body ")"] "FI"

repeat_function = "REPEAT" "(" expr ")" "(" body ")"

while_function = "WHILE" "(" expr ")" "(" body ")"

The expressions will evaluate to binary numbers. As in PL/M 80, two's complement representation is used so that negative expressions will map into a large positive number (e.g., "−1" maps into 0FFFFH). The bodies of these functions are balanced_text strings, and although they look exactly like arguments in the syntax diagrams, they are processed very much like the bodies of user-defined macro functions; the bodies are "called" based upon some aspect of the expression in the IF, REPEAT, or WHILE function. The effects for each control function are described below.

The IF Function

The first argument is evaluated Normally and interpreted as a numeric expression.

If the value of the expression is odd (=TRUE) then the body of the THEN phrase is evaluated and becomes the value of the function. The body of the ELSE clause is not evaluated.

If the value of the expression is even (=FALSE) and the ELSE clause is present, then the body of the ELSE phrase is evaluated and becomes the value of the function. The body of the THEN clause is not evaluated. Otherwise, the value is the null string.

In the cases in which the body is evaluated, evaluation is Normal or Literal as determined by the presence or absence of the call- literally character on the IF.

Examples:

```
%IF (%VAL GT 0) THEN( %DEFINE(SIGN)(1) ) ELSE( %DEFINE(SIGN)(0) ) FI
```

If the value of the numeric symbol VAL is positive, then the SIGN will be defined as "1"; otherwise, it will be defined as "0". In either case the value of the IF function is the null string.

```
%DEFINE(SIGN) (%IF (%VAL GT 0) THEN(1) ELSE(0) FI)
```

This example has exactly the same effect as the previous one.

The REPEAT Function

The REPEAT function causes its body to be expanded a predetermined number of times. The first argument is evaluated Normally and interpreted as a numeric expression. This expression, specifying the number of repetitions, is evaluated only once, before the expansion of the text to be repeated begins. The body is then evaluated the indicated number of times, Normally or Literally, as determined by the presence or absence of the call- literally character on the REPEAT and the resulting string becomes the value of the function. A repetition number of zero yields the null string as the value of the REPEAT function call.

Examples:

Rotate the accumulator of the 8080 right six times:

```
%REPEAT (6)
(       RRC
)
```

Generate a horizontal coordinate line to be used in plotting a curve on a line printer. The line is to be 101 characters long and is to be marked every 10 characters:

```
%REPEAT (10) (+%REPEAT (9) (.))+
```

evaluates to:

```
+.........+.........+.........+.... (etc.) ...+.........+
```

The WHILE Function

The WHILE function tests a condition to determine whether the body is to be evaluated. The first argument is evaluated Normally and interpreted as a numeric expression. If the expression is TRUE (=odd) then the body is evaluated, and after each evaluation, the condition is again tested. Reevaluation of the functional string continues until the condition fails (i.e. the value of the expression is an even number).

The body of the WHILE function is expanded Normally or Literally depending on how the function was called.

Example:

```
%WHILE (%I LT 10) ( ...
                    ... %IF (%FLAG) THEN(%DEFINE(I) (20)) FI
                    ...
                    ...)
```

The EXIT Function

The syntax for EXIT function is:

exit_function = "EXIT"

This function causes termination of processing of the body of the most recently called REPEAT, WHILE, or user-defined macro. The value of the text already evaluated becomes the value of the function. The value of the exit function, itself, is the null string.

Example:

```
%WHILE (%Cond) ( ...
                    ...
                    %IF (%FLAG) THEN (%EXIT) FI
                    ...
                    ... )
```

Console Input and Output

The Macro Processor Language provides functions to allow macro time interaction with the user.

The IN function allows the user to enter a string of characters from the console. This string becomes the value of the function. The IN function will read one line from the console (including the terminating carriage return line feed).

The OUT function allows a string to be output to the console output device. It has the null string as a value. Before it is written out, the string will be evaluated Normally or Literally as indicated by the mode of the call to OUT.

The syntax of these two functions is:

in_function = "IN"

out_function = "OUT" "(" balanced_text ")"

Examples:

```
%OUT (Enter the date:)
%DEFINE(DATE)(%IN)
```

(Note that DATE will include <CR> <LF>. Refer to MATCH in Chapter 7 for one way to strip off <CR> <LF>. You can use SUBSTR with LOCAL for another.)

The Substring Function

The syntax of the substring function is:

```
substr_function =
    "SUBSTR" "(" balanced_text "," expr1 "," expr2 ")"
```

The text string is evaluated Normally or Literally as indicated by the mode of the call to SUBSTR. Assume the characters of the text string are consecutively numbered, starting with one. If expression 1 is zero, or greater than the length of the text string, then the value of this function is the null string. Otherwise, the value of this function is the substring of the text string which begins at character number expression 1 of the text string and continues for expression 2 number of characters or to the end of the string (if the remaining length is less than expression 2).

Examples:

```
%SUBSTR (ABCDEFGH,3,4)          has the value "CDEF"
%SUBSTR (%(A,B,C,D,E,F,G),2,100)  has the value ",B,C,D,E,F,G"
```

The MATCH Function

The syntax of the MATCH function is:

```
match_function =
  "MATCH" "(" id1 delimiter_specifier id2 ")" "(" balanced_text ")"
```

The MATCH function uses a pattern that is similar to the define pattern of the DEFINE function. It contains two identifiers, both of which are given new values as a result of the MATCH function, and a delimiter_specifier. The delimiter_specifier has the same syntax as that of the DEFINE function. The balanced_text is evaluated Normally or Literally, as indicated by the call of MATCH, and then scanned for an occurrence of the delimiter. The algorithm used to find a match is exactly the same as that used to find the delimiter of an argument to a user-defined macro. If a match is found, then id1 will be defined as the value of the characters of the text which precede the matched string and id2 will be defined as the value of the characters of the text which follow the matched string. If a match is not found, then id1 will be defined as the value of the text string, and id2 will be defined as the null string. The value of the MATCH function is always the null string.

Examples:

Assume XYZ has the value "100,200,300,400,500". Then the call,

```
%MATCH(NEXT,XYZ) (%XYZ)
```

results in NEXT having the value "100" and XYZ having the value "200,300,400,500".

```
%DEFINE (LIST) (FLD1,3E20H,FLD3)

%WHILE (%LEN(%LIST) NE 0)
(       %MATCH(PARM,LIST) (%LIST)
        MOV     [BX],%PARM
        INC     BX
)
```

The above will generate the following code:

```
MOV     [BX],FLD1
INC     BX
MOV     [BX],3E20H
INC     BX
MOV     [BX],FLD3
INC     BX
```

Assume that SENTENCE has the value "The Cat is Fat." and that VERB has the value "is", then the call,

```
%MATCH(FIRST %VERB LAST) (%SENTENCE)
```

results in FIRST having the value "The Cat " and LAST having the value " Fat.".

The Comment Function

The comment function allows the programmer to comment his macro definition and/or source text without having the comments stored into the macro definitions or passed on to the host language processor. The call-literally character may not be present in the call to the comment function. The syntax is:

comment_function = "'" text "'" | linefeed

When a comment function is recognized, text is unconditionally skipped until either another apostrophe is recognized, or until a linefeed character is encountered. All text, including the terminating character, is discarded; i.e. the value of the function is always the null string. The comment is always recognized except inside an escape function. Notice that the comment function provides a way in which a programmer can spread out a macro definition on several lines for readability, and yet not include unwanted end of line characters in the called value of the macro.

Examples:

```
%' This comment fits within one line.'
%' This comment continues through the end of the line. <LF>
```

The Metachar Function

The metachar function allows the programmer to change the character that will be recognized by the macro processor as the metachar. The use of this function requires extreme care. The value of the metachar function is the null string. The syntax is:

metachar_function = "METACHAR" "(" balanced_text ")"

The first character of the balanced_text is taken to be the new value of the metachar. The following characters cannot be specified as metacharacters: a logical blank, left or right parenthesis, an identifier character, an asterisk, or control characters (i.e. ASCII value < 20H).

INDEX

Note: See also the instruction set index at the end of Chapter 5.

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that mee the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness o this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions fo improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types c documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

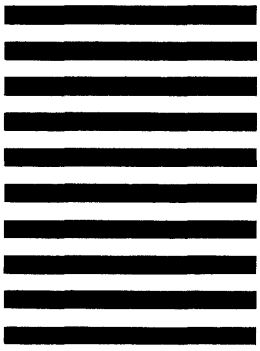
CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

E'D LIKE YOUR COMMENTS ...

is document is one of a series describing Intel products. Your comments on the back of this form will
lp us produce better manuals. Each reply will be carefully reviewed by the responsible person. All
nments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.