MCS-86 ASSEMBLY LANGUAGE REFERENCE MANUAL

Manual Order Number: 9800640A

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.

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
Intellec	Multibus	μScope

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

PREFACE

Two main purposes guided the creation of this book: the first was to teach this language in easy steps of concepts and usage. In each section everything should be understandable using information that is defined close-by or is immediately findable via an explicit local reference to other sections in the text. Although the first steps always seem too easy, a gradual approach usually helps you progress farther with less effort.

The other purpose was to offer easy access to reference information. This purpose is addressed via the figures, tables, glossary-index, and tab-type flag headings on the top outer corners of reference pages.

Most chapters and sections begin with a brief explanation of their content, why and when you need to use what is presented, plus prerequisite or related information and where to find it.

This manual proceeds from general overview topics to a gradual development of the specific features of this assembly language. Later in the book come the complete and concise presentations of commands, permissible constructs, and other considerations.

If you are an advanced assembly programmer, you may not require this careful and gradual introduction to underlying concept and structure. You may choose to leap ahead, learning as you go the similarities and differences of this language from your prior knowledge/experience. However, at least one glance through might prove beneficial as to new concepts, features or requirements.

For those with less experience, the manual attempts to prepare you in advance for novel or complex concepts by supplying the motivation or rationale behind them. This can help you to understand, for example, why some otherwise appealing shortcuts are dangerous or disallowed.

Thus this manual gives an overview, first of certain programming considerations and then of the 8086 architecture in terms of addressing, register sets, and memory layout. It proceeds then to look briefly at this assembler's commands, directives, and automatic features, some of which are unusual in any assembler, particularly for a microprocessor.

With these topics as background and introduction, the manual begins to teach the details of the language in a conversational style, with many examples.

If your experience leads you to prefer to browse at random, you might benefit most by reading the assembler features, skipping the tutorial sections, and studying the later detailed discussions of the instructions and expressions. The frequent embedded references to discussions or explanations elsewhere in the manual will likely lead to further browsing, gradually filling in the full story.

CONTENTS

CHAPTER 1
INTRODUCTION TO 8086 ASSEMBLY LANGUAGE

Assembly Language and Processors

Introduction

This book is about the 8086 Assembly Language. The instructions and directives in this language use readily remembered abbreviations (e.g. MOV, ADD, EQU) for programming operations and assembler control. A block of such instructions and directives, intended for processing as a unit by the assembler, is called a source module. The assembler translates a source module into relocatable object code.

Assembly language source modules must be in a machine-readable form when passed to the assembler. The Intellec development system includes a text editor that will help you maintain source programs as diskette files. You can then pass the resulting source program file to the assembler. (The text editor is described in the *ISIS-II System User's Guide.*)

Most lines of source coding in an assembly language source program translate directly into one machine instruction for a particular processor. The assembly language programmer should be familiar with both the assembly language and the processor for which he or she is programming. The 8086 architecture and registers are described in this chapter. The instructions are summarized in Chapter 6.

What is an Assembler?

An assembler is a software tool — a program designed to simplify the task of writing computer programs. If you have ever written a computer program directly in a machine-recognizable form such as binary or hexadecimal code, you will appreciate the advantages of programming in a symbolic assembly language. There is less to remember. It is easier to verify the program's validity and to correct it.

An assembly language is a step up from coding instructions directly in machine language. How large an improvement it is depends on how much the assembler does for you and how smart it is. This means how many correct decisions it can make about what machine code to generate, based on inferences from the code you write and any additional information you supply.

Thus a good assembler requires a minimum of source input lines written in a language easily handled by humans, and generates the machine-instructions you would otherwise laboriously code by hand if you had memorized the entire instruction set of the machine (plus, in the case of certain esoteric machines, wiring diagrams).

What the 8086 Assembly Language Provides

Mnemonic Instructions

The language includes about 100 symbolic instructions, grouped into six classes. From source input in this language, the assembler can generate over 3,800 distinct machine-instructions. Data or addresses (user-defined variables and labels) would

add to this total, requiring even more of your effort to manipulate and validate. However, handling data and addresses by readily remembered names instead of numbers is automatic in the 8086 assembler, which translates the assembly language program into machine code.

Assembly language operation codes (opcodes) are easily remembered, e.g., MOV for move instructions, JMP for jump. Using names for address labels and variables, you can make them meaningful to the problem you are solving.

For example, if your program must manipulate a date as data, you can assign the symbolic name DATE to its address e.g.

```
DATE    DB    '780704'
```

If your program contains a set of instructions used as a timing loop, (a set of instructions executed repeatedly until a specific amount of time has passed), you can name the first instruction of the group TIMER, e.g.

```
TIMER:    MOV    AX,255
```

DATE is called a VARIABLE, because it is a name for a memory location whose contents are used as data. TIMER is a LABEL because it names a memory location whose contents are used as an instruction.

Typing

The 8086 assembly language is "strongly typed". This means it performs extensive checks on your variables and labels, like DATE and TIMER. The assembler uses the attributes which are derived implicitly when a variable or label is first declared (defined). The assembler makes sure that each use of a symbol in later instructions conforms to the usage defined for that symbol when it was declared. For example, DATE has the type "byte" because DB was used to define/declare it. The typing mechanism and means for overriding it are more fully explained in Chapters 2 and 5.

What these checks provide, of course, is an extra safeguard against unintended or meaningless code arising from errors of omission or inconsistency. These errors can sometimes slip into the middle of a complex, high-pressure project and be difficult to discover until deep into the debugging process.

Enhanced Data Handling

Compared to earlier assembly languages, this one has substantially improved flexibility in data definition and manipulation, which can significantly simplify coding. Its capabilities allow very sophisticated goals to be achieved with a straightforward use of the language.

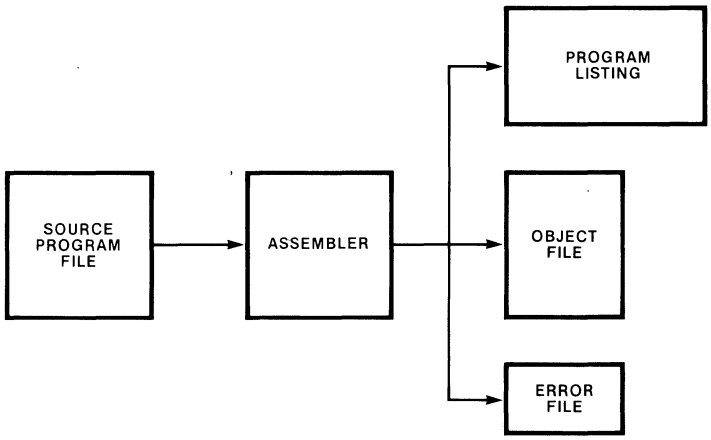
Powerful string manipulation instructions permit direct transfers to or from memory or the accumulator. They can be prefixed with a repeat operator for repetitive execution with a count-down and a condition test. These operations automatically increment or decrement the relevant indexes to memory, depending on the direction flag, (DF). They also automatically decrement the count register, (CX), after each repetition. This implicitly controls the number of iterations by terminating the operation when CX = 0.

The assembler fully supports the 8086 addressing modes by providing for complex expressions involving multiple indexes and field offsets. A powerful EQU facility allows the use of simple synonyms for complicated expressions which may recur throughout a module.

What the Assembler Does

The Assembler performs the clerical task of translating your symbolic source code into object code which can be executed by the 8086 microprocessor (after the relocation and linkage facilities assign absolute addresses). The major functions of the assembler include assigning a value to each name you code, and later substituting this value for every use of that name. Assembler input is your source file. The output consists of three possible files:

1. the object-file containing your program which has been translated into object code;
2. the list-file printout of your source code, the assembler-generated object code, error messages, and the symbol table, and
3. a file containing only error messages and the source lines in which the errors occurred.



Object Code

The object code is the form of the program ultimately executed (after intermodule references are handled by LINK86 and absolute addresses are assigned by LOC86 or QRL86). For most microcomputer applications, you probably will eventually load the object program into some form of Read-Only Memory. This assembler produces output modules in relocatable format. The ability to use the above-named Relocation and Linkage facilities (R&L) frees you from worrying about the eventual mix of read only and random access memory in the application system; individual portions of the program can be located as needed when the application design is final.

Also, the R&L linking facility allows a large program to be broken into a number of separately assembled modules. Such modules are both easier to code and to test, and can later be linked to function as a unit. A more thorough description of these advantages appears later in this chapter.

Program Listing

The program listing provides a permanent record of the source program, the object code, and the assembly process. The program listing also shows the assembler's

diagnostic messages issued for programming errors. For example, if you specify a 16-bit value for an instruction that can use only an 8-bit value, the assembler tells you that the value exceeds the permissible range. (See accompanying sample listing.)

```
LOC OBJ          LINE SOURCE

                    1 ; THIS SDK86 PROGRAM ECHOS CHARACTERS FROM A KEYBOARD TO A CRT,
                    2 ; AND GENERATES A FREQUENCY DISTRIBUTION OF CHARACTER OCCURRENCES.
                    3
—                   4 RAMSEG     SEGMENT AT 30H              ;PLACE RAM SEGMENT AT 300H
0000 (128           5 FREQUENCY  DB      128 DUP(0)          ;INITIALIZE OCCURRENCE COUNT ARRAY TO ZERO
     00
     )
0080 (10            6            DW      10 DUP(?)           ;RESERVE AN AREA FOR THE STACK
     ????
     )
0094                7 STKTOP     LABEL   WORD                ;INITIAL STACK POINTER POSITION
—                   8 RAMSEG     ENDS
                    9
                   10
—                  11 ROMSEG     SEGMENT AT 20H              ;PLACE ROM SEGMENT AT 200H
                   12            ASSUME  CS:ROMSEG,DS:RAMSEG,SS:RAMSEG,ES:NOTHING
                   13
  FFF0             14 USARTDATA  EQU     0FFF0H              ;8251A DATA PORT ON SDK86
  FFF2             15 USARTSTAT  EQU     0FFF2H              ;8251A STATUS PORT
0000 3000          16 SETSEG     DW      RAMSEG              ;SEGMENT ADDRESS OF BEGINNING OF RAMSEG
                   17
0002 2E8E1E0000    18 START:     MOV     DS,CS:SETSEG        ;SET UP DATA SEGMENT AS IN ASSUME
0007 2E8E160000    19            MOV     SS,SETSEG           ;SET UP STACK SEGMENT
000C 8B269400      20            MOV     SP,OFFSET STKTOP    ;SET INITIAL STACK POINTER VALUE
                   21
0010 E81900        22 LOOP1:     CALL    CI                  ;READ CHARACTER TO AL
0013 8AE0          23            MOV     AH,AL
0015 E80500        24            CALL    CO                  ;WRITE CHARACTER FROM AH
0018 E81E00        25            CALL    COUNTIT             ;COUNT OCCURRENCE OF CHARACTER IN AL
001B EBF3          26            JMP     LOOP1
                   27
001D BAF2FF        28 CO:        MOV     DX,USARTSTAT        ;IF USART NOT READY FOR CHARACTER
0020 EC            29            IN      AL,DX               ;INPUT A BYTE INTO AL WITH PORT # IN DX
0021 2401          30            AND     AL,1
0023 74F8          31            JZ      CO                  ;THEN WAIT
0025 BAF0FF        32            MOV     DX,USARTDATA        ;ELSE OUTPUT CHARACTER
0028 8AC4          33            MOV     AL,AH
002A EE            34            OUT     DX,AL
002B C3            35            RET
                   36
002C BAF2FF        37 CI:        MOV     DX,USARTSTAT        ;IF CHARACTER NOT READY
002F EC            38            IN      AL,DX
0030 2402          39            AND     AL,2
0032 74F8          40            JZ      CI                  ;THEN WAIT
0034 BAF0FF        41            MOV     DX,USARTDATA        ;ELSE BRING IN CHARACTER
0037 EC            42            IN      AL,DX
0038 C3            43            RET
                   44
0039               45 COUNTIT    PROC    NEAR                ;EXPECTS CHARACTER IN AL
0039 32E4          46            XOR     AH,AH               ;ZERO AH
003B 8BF0          47            MOV     SI,AX               ;16 BIT INDEX INTO FREQUENCY TABLE IN SI
003D FE04          48            INC     FREQUENCY[SI]       ;INCREMENT ARRAY INDEXED BY SI
003F C3            49            RET
                   50 COUNTIT    ENDP
                   51
—                  52 ROMSEG     ENDS
                   53
 0002              54            END     START
```

Line #	Explanation	See also chapter
1 & 2	Comment line	2
4	Declaration for data segment in RAM	4
5	Array declaration of 100 zeroed bytes	3
6	Declaration of 10 uninitialized words	3
7	Declaration of label for stack area	4
8	End of declaration of RAM segment	4
11	Declaration for code segment in ROM	4
12	Declaration to assembler of run-time segment register values	4
14 & 15	Symbolic equivalences for hardware I/O port number	1, 4
16	Address constant for use in 18 and 19	3
18	Initialize DS register with address constant for ROM	1, 5
19	Initialize SS register with address constant from ROM	1
20	Move immediate address constant to SP register	1
22, 24, 25	Subroutine calls	1, 6, Ap D
28	Move immediate device address to DX (see line 15)	6
29	Input byte to AL using I/O address in DX	1
35	Subroutine return (to line 25)	1, 6, Ap D
45	Procedure declaration of type NEAR (optional)	1, 4
48	Increment with indexed address mode	1, 5
50	End of procedure declaration	4
52	End of segment declaration	4
54	End of Program	4

NOTES:
i) Segments can be located using "at" option as shown on lines 4 and 11
ii) Segment register and SP must be initialized (lines 19, 20)
iii) Explicit use of segment override prefix (line 18) Cs: was not required, but shown for emphasis
iv) Procedure declaration not required but shown as example

Error File

The assembler detects a variety of errors, both syntactic (form) and semantic (meaning). The error messages are listed in the Operator's Manual. The discussions in the present manual present the correct use of this assembly language and mention certain common errors that may occur.

Do You Need the Assembler?

The assembler is but one of several tools available for developing microprocessor programs. Typically, choosing the most suitable tool is based on cost constraints versus the required level of performance and support from the software tools you choose. Your company and you must determine cost constraints; the required level of performance and support depends on a number of variables:

- The number of programs to be written:

  The greater the number of programs to be written, the more you need development support. When your application has access to the power of a microprocessor, you can provide clients with custom features through program changes. Furthermore, you can add features with programming. Thus your product can offer unique or more complete services, if the requisite development support is available.

- The time allowed for programming:

  As the time allowed for programming decreases, the need for programming support tools increases.

- The level of support for existing programs:

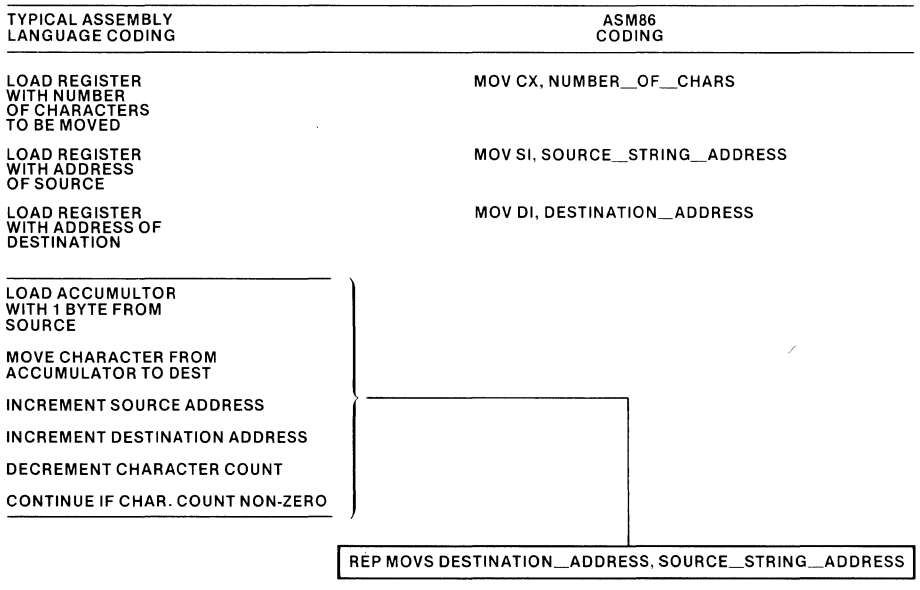
  Sometimes programming errors are not discovered until the program has been in field use for quite a while. Your need for programming support increases if you agree to correct such errors for your clients. The number of different supported programs in use can greatly multiply this requirement. Also, program support is frequently subject to stringent time constraints.
- The complexity of the problem(s) to be solved.

If most of the support-needed factors listed above apply to you, you may also want to explore the advantages of a higher-level language such as PLM86. PLM86 is Intel's compiler language for program development. Such languages are directed more toward problem-solving than most assemblers, although ASM86 is closer to that than most.

PLM86 may allow you to write programs more quickly than the assembly language. On the other hand, the assembler offers greater flexibility and control in direct manipulation of the 8086 and its operation. In many cases there are many advantages to a mixture of 80-90% PLM and 10-20% assembly language. Using PLM for most of the code can cut development time, while the assembly language programs can be chosen to provide critical control and performance factors.

As an example of the compression of coding, the accompanying chart illustrates the differences between most assembly languages and ASM86. The more typical code would take about 18 bytes. The ASM86 code takes 8 bytes.

Assume that a program must move five characters from one location in memory to another. The assembly language instructions are represented in a flowchart. The PLM86 code is: CALL MOVE (NUMBEROFCHARS, SOURCE$STRING$ADDRESS, DESTINATION$ADDRESS).



Programming, Debugging, and Designing

At many stages of problem definition it is easy to leap into programming, producing modules or entire subsystems relatively soon after their functions are first described.

Problems arise, however, when these modules don't work or are incomplete. They will require expansion, modification, or integration with other modules. Some functions may merge. Communication of parameters may change.

Reprogramming may be required again and again, to fit things in, to rework sections, and to retest everything.

Thus arises the idea of good design, to cope in advance with later change.

Change is costly in time and effort. The later it occurs, the greater the cost. How may we minimize change? For unavoidable change, how may we minimize the redesign, rework, and relearning resulting from changes or errors in specification? in design? in implementation?

The answers lie in 4 main areas:

1. extremely clear and specific goals for the program, written down and explicitly agreed upon by the designers, implementers, and users

2. isolation into separate modules of every non-trivial function of the system or program, including the isolation of difficult design decisions

3. full and clearly understandable documentation for every module, including liberal comments in the code

4. clearly written standards for implementation of modules, including conventions for naming and passing parameters.

Study of this assembly language may not help you to choose your goals well, to formulate them clearly, and to get everyone to agree on them. It can, however, aid you in making your work modular, parameterized, and easier to document. This then assists delegation of tasks, teamwork, ease of modification, and the sharing of modules among tasks or teams without the usual cutting and stitching to make them fit.

How This Assembler Helps

1. One of this assembler's first contributions to successful projects is the use of typed symbols. A variable is typed 1 or 2 or 4 bytes by the basic unit of its declaration, being byte or word or doubleword, respectively. A label is typed NEAR (usually within the same segment) unless you state FAR in its declaration (types are discussed more in Chapters 2 through 5).

   There are several reasons this helps. It allows one mnemonic (e.g. MOV, JMP) to be used for one kind of function applied to several different kinds of data or operands, even though there may be many hardware instructions to choose from. The assembler can choose the correct hardware instruction based on the type of the operands you supply in the source line. This reduces the amount and complexity of what you must remember. It allows you to focus more on solving your problem of creating the program, and less on conforming to too many restrictions.

One example of how typed symbols help keep things simple is the mnemonic MOV. It is used in the same form, MOV, for over 2 dozen different hardware instruction cases. These cases cover the use of bytes, words, registers, memory, and immediate data expressions, either as source or destination operands.

Examples:

```
MOV AL, BYTE_EXPRESSION   ; low byte of accumulator as destination implies byte
                          ; source

MOV AX, WORD_EXPRESSION   ; full-word accumulator implies word source

MOV BYTE_EXPRESSION, AL   ; low byte of accumulator as source implies byte
                          ; destination

MOV BX, AX                ; full word register to full word register

MOV MEM_BYTE, AL          ; AL as source requires the destination to have type
                          ; "byte"

MOV CL, MEM_BYTE_IMMEDIATE_VALUE      ; register CL as destination
                                      ; requires a byte source
```

The use and implications of each of the above instructions and expressions are further explained in the chapters that follow.

2. The assembler is built to assemble programs as collections of user-defined segments, up to 64K bytes in each segment. This is a natural aid to modularizing your code, grouping together related functions and/or data.

   Examples:

```
INITIALIZATION_ROUTINE      SEGMENT
                            o
                            o; statements to initialize
                            o; tables from input and
                            o; verify consistency, etc.
                            o
                            o
INITIALIZATION_ROUTINE ENDS

GET_5_TYPE_1_RECORDS        SEGMENT
                            o
                            o; input and edit
                            o
GET_5_TYPE_1_RECORDS        ENDS

PROCESS_FIRST_TYPE_1_REC    SEGMENT
                            o
                            o; begin analysis,
                            o; updating, reporting
PROCESS_FIRST_TYPE_1_REC    ENDS
```

3. It supports procedures and code macro definition. These features make it easier to break a problem down into separately programmable functions which are easier to develop, test, and modify. Once they are created as generalized routines, other team members and other projects can share their use. You can create libraries of programs representing solved problems, problems that needn't again use up your time and manpower resources.

4. The assembler is very flexible in allowing a large variety of address expressions, which are used to refer to locations in memory. It has a very powerful name- and synonym-capability using the EQU directive. This gives you the option to use expressions and names that are meaningful to you in the context of your application. It is an aid both to clearer thinking and greater readability of the program. This in turn is helpful in writing, testing, and modifying that program.

   Example:

   ```
   CUR_PROJ   EQU   PAYROLL_REC [BX] [SI]
   ```

   The expression on the right could represent part of one employee's payroll record, out of the many such data records indexable in memory by registers BX and SI. This kind of usage for base and index registers is called indexing or subscripting. It is a commonly-used form of address-expression.

5. A number of operators are provided for use in expressions. They aid good programming practices, the generality of the code, and the ease of modifying that code. They enable you to refer to (or change) attributes of variables or labels without coding those attributes explicitly.

   Examples:

   If you define an array of 50 words initialized to 0 via

   ```
   RATE_TABLE   DW   50   DUP   (0)
   ```

   then your instructions may later say

   ```
         MOV   CX,    LENGTH RATE_TABLE
         MOV   SI,    SIZE RATE_TABLE
   FILL: SUB   SI,    TYPE RATE_TABLE
         MOV   AX,    INPUT_ RATE [SI]
         MOV   RATE_TABLE [SI], AX
         LOOP  FILL
   NEXT:
   ```

   thereby putting 50 into CX (the number of entries), 100 into SI (the number of bytes in the word array), and decrementing SI for each iteration by 2 (the number of bytes in each element). LOOP automatically decrements CX by 1 until 0 is reached, halting the iterations and going on to the next sequential instruction, at NEXT.

Introduction to Relocation

Relocation means the ability to reassign addresses at the time the program is loaded into memory, changing them from the relative addresses assigned during assembly.

This feature allows you to subdivide a complex program into a number of smaller, simpler programs. After development and debugging of the component modules, you can link them together, locate them as you choose and enter final testing with much of the work behind you. (For information on this process of linking and locating, see the *8086 Cross Development Utilities Operator's Manual*). Component modules can often be allocated among members of a development team, and grow into a library of solved problems for later use.

The relocation feature also has a major advantage at assembly-time: often, large programs with many symbols cannot be assembled because of limited work space for the symbol table. Such a program can be divided into a number of modules that can be assembled separately and then linked together to form a single object program.

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory. This offers developers of microcomputer systems advantages in two areas, memory management and modular program development.

Modular Program Development

As an example, two programmers might take different approaches to solve the following problem. Both programmers want to calculate the degree of spark advance that provides the best fuel economy with the lowest emissions. Programmer A codes a single program that senses all inputs and calculates the correct spark advance. Programmer B uses a modular approach and codes separate programs for each input plus one program to calculate spark advance.

Although programmer A avoids the need to learn to use the relocation feature (a one-time cost), the modular approach used by programmer B has a number of advantages:

- Simplified Program Development

  It is generally easier to code, test, and debug several simple programs than one complex program.

- Sharing the Programming Task

  If programmer B finds that he or she is falling behind schedule, one or more subprograms can be assigned to another programmer. Programmer A will probably have to complete the program alone, because of the single program concept.

- Ease of Testing

  Programmer B can test and debug most modules as soon as they are assembled; programmer A must test the program as a whole. B has an extra advantage if the sensors are being developed at the same time as the program: if one of the sensors is behind schedule, programmer B can continue developing and testing programs for the sensors that are ready. Because programmer A cannot test the program until all the sensors are developed, the testing schedule is dependent on events beyond his or her control.

- Programming Changes

  It is reasonable to expect some changes during product development. If a change to one of the sensors requires a programming change, programmer A must search through the entire program to find and alter the coding for that sensor. The entire program must then be retested to be certain those changes do not affect any of the other sensors. By contrast, programmer B need be concerned only with the module for that one sensor.

  Similarly, when a bug in some function is reported, programmer B is likely to find it much sooner, because functions have been localized into modules. Programmer A may need to trace through the whole program to see the connections leading to the error.

Modularity reduces development time and cost overall. It continues to be a major advantage throughout the life of the program. Flaws are more readily found and corrected. Enhancements are accomplished more quickly.

One disadvantage is the greater need for intermodule and interpersonal communication. Another is the need to manage a larger data base of current modules as they go through changes and tests.

Overview of Hardware and Architecture

The rest of this chapter is devoted to the following topics:

- 8086 hardware, partial review (see *8086 User's Manual* for complete detail)
- General Registers
- Memory
- Pointer and Index Registers
- Program counter (Instruction Pointer)
- Segment Registers
- Condition flags
- Stack and stack pointer
- Input/output ports

Hardware

The 8086 is a complete microprocessor for use in general-purpose computer systems of widely varying levels of complexity. At its upper limit, it can be part of a multiple-processor system, each of whose processors is capable of accessing up to 1 megabyte of memory.

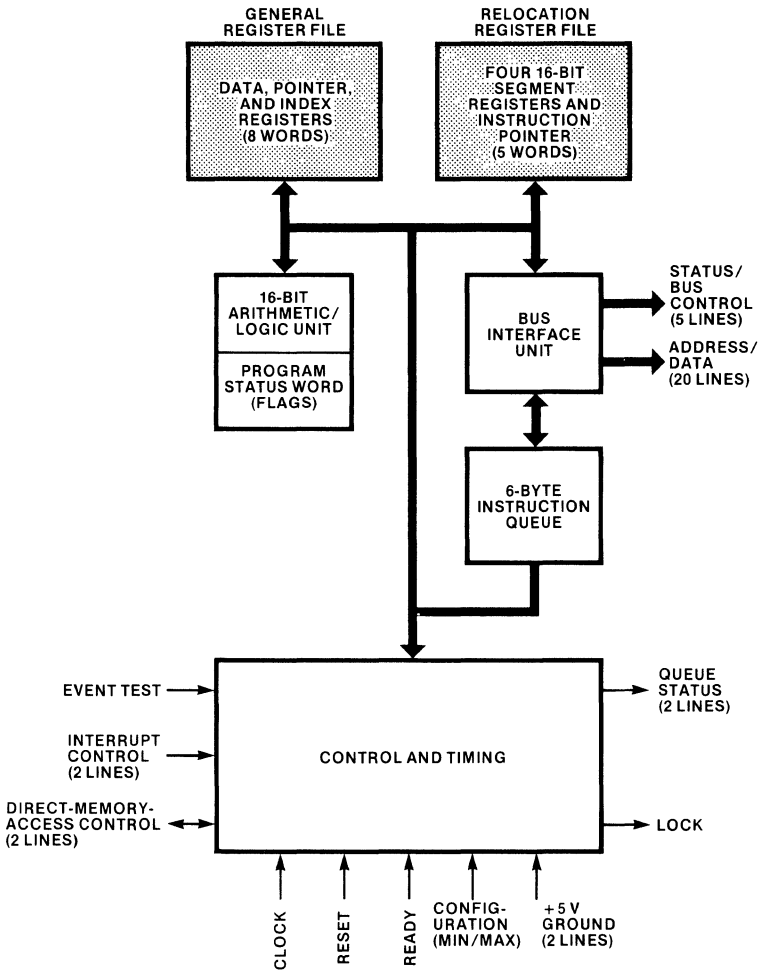
Memory transfers are handled in 8-bit bytes or in 16-bit words. Bit, byte, word, and block (string) operations are accommodated in the instruction set. The 8086 performs signed arithmetic and interruptible string operations, and can make use of dynamically relocatable procedures, reentrant programming, and multiprocessing.

Access to memory and peripherals is accomplished through a 20-bit time- multiplexed address and data bus. Internal configuration switching can adapt the processor to the level of system complexity you desire.

The bus structure of the MCS-86 systems is compatible with MCS-80 and MCS-85 peripherals. This allows you to utilize pre-existing devices and hardware designs. Existing 8080 system software also is adaptable for use in the MCS-86 systems.

The 8086 uses a queue of prefetched instructions. This aids throughput, but the calculation of expected execution times must take into account rebuilding the queue in some circumstances. A jump or call typically forces building a new queue. Conditional jumps (and loops) naturally only do this sometimes: when the jump is taken, 16 clock cycles are used, otherwise 4.

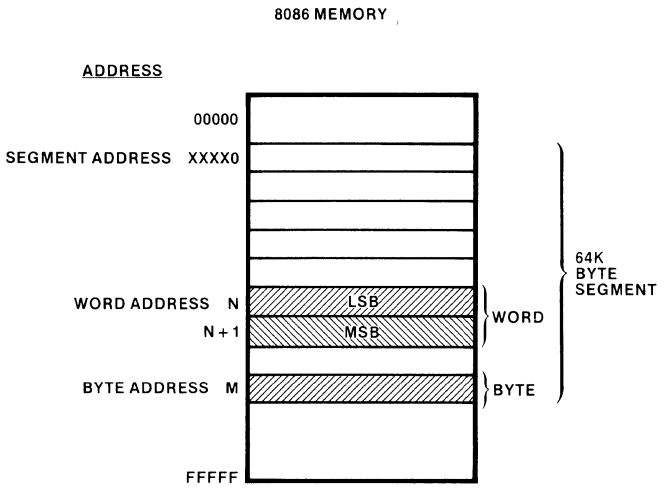
See the *8086 User's Manual* (9800722) for more information on all hardware and timing data.


GENERAL REGISTER FILE
DATA, POINTER, AND INDEX REGISTERS (8 WORDS)
RELOCATION REGISTER FILE
FOUR 16-BIT SEGMENT REGISTERS AND INSTRUCTION POINTER (5 WORDS)
16-BIT ARITHMETIC/ LOGIC UNIT
PROGRAM STATUS WORD (FLAGS)
BUS INTERFACE UNIT
STATUS/ BUS CONTROL (5 LINES)
ADDRESS/ DATA (20 LINES)
6-BYTE INSTRUCTION QUEUE
EVENT TEST
INTERRUPT CONTROL (2 LINES)
DIRECT-MEMORY-ACCESS CONTROL (2 LINES)
CONTROL AND TIMING
QUEUE STATUS (2 LINES)
LOCK
CLOCK
RESET
READY
CONFIG-URATION (MIN/MAX)
+5 V GROUND (2 LINES)

Memory Organization

The 8086 uses a 20-line address bus to locate a byte or word in memory, and can therefore access 2 to the 20th bytes (1 megabyte = 1,048,576 bytes). Each location so addressed is a byte, 8 bits.

Words (16-bits) consist of 2 consecutive bytes, and can begin at even or odd addresses. Words in memory are stored with the least significant byte in the lower- addressed location and the most significant byte in the higher-addressed location. When a word begins at an even address, access requires only 1 memory cycle. If it begins at an odd address, access requires 2 memory cycles, with no other penalty. Each hardware memory cycle is 4 clock cycles.

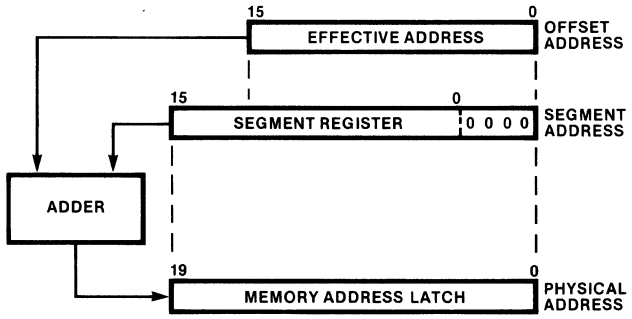


- 8086 OPERATES ON 8-BIT BYTES and 16-BIT WORDS.
- A BYTE OR WORD CAN ONLY BE ACCESSED BY THE CPU IF IT RESIDES IN ONE OF FOUR CURRENT 64K SEGMENTS AS ADDRESSED BY THE SEGMENT RELOCATION REGISTERS.
- EVERY BYTE WITHIN A SEGMENT IS ADDRESSABLE USING A 16-BIT ADDRESS.
- WORDS OCCUPY TWO ADJACENT BYTES WITH THE MSB OF A WORD IN THE HIGHER ADDRESSED MEMORY LOCATION. WORDS ARE ADDRESSED BY LSB (LOWER) MEMORY ADDRESS.

Memory Addressing

Memory addresses must be 20-bits long to access unique bytes in the megabyte memory. To achieve this using 16-bit words, the 8086 memory space is viewed as consisting of 64K-byte segments. Within each 64K segment a 16-bit address is sufficient to access any byte. This address is called the offset.

To specify which 64K segment contains the desired location, a second 16-bit address is used, called the segment address, or simply, segment. This is used as a base- address, moved by the user program into one of the four segment registers designed for this purpose. It is useful to think of every address as such a pair of numbers, the segment and offset.



The 8086 memory can be thought of as an arbitrary number of segments. A byte or word within a segment is addressed with a 16-bit offset address. The 8086 automatically adds the offset address to the shifted 16-bit segment address, creating a 20-bit physical address.

The hardware automatically forms a unique 20-bit address from these two words in the following way:

The segment address is shifted left 4 bits in a special 20-bit register, leaving its lowest 4 bits zero. The other address word, the offset, is then added to that 20-bit number, giving a unique 20-bit address as the result.

Example:

A location with segment word of 123AH and with an offset word of 341BH would be addressable by the 20-bit number 157BBH, formed by adding the two words as described above.

```
123AH becomes  123A0H, to which is added 341BH,
             + 341BH
             -------
               157BBH
```

This design thus expects segments to begin at an address ending in 4 zeroes. Such addresses are called paragraph boundaries, and the 4 high-order hex digits are called the paragraph number. A segment address, therefore, is always a paragraph number in the assembly language.

One consequence of the above design is that every byte is accessible from many base-addresses using different offsets. For example, the sample address above (157BBH) can be reached with an offset of 0BH and a segment of 157BH, or an offset of 8BH and a segment of 1573H. The 64K-byte segments can overlap, depending on their beginning point.

Complete addresses are formed using the above technique, with the segment word always placed in one of the 4 segment registers, called CS for code segment register, DS for data segment register, SS for stack segment register, and ES for extra segment register.

These 4 segment-registers, discussed later in this chapter, are used as base addresses for all references into memory, potentially a megabyte. Thus, you can address up to one quarter of the megabyte, i.e., four 64K-byte physical segments, at any one time. A new segment becomes addressable whenever the contents of one of these segment registers is changed by your program.

Registers

The 8086 processor contains three sets of four 16-bit registers and a set of nine one-bit flags. The sets of registers are the GENERAL registers, the POINTER and INDEX registers, and the SEGMENT registers. There is a 16-bit Instruction Pointer (IP) which is not directly accessible; rather it is manipulated with control transfer instructions. (See figure below.)

Instruction Pointer. The Instruction Pointer keeps track of the next instruction byte to be fetched from memory, which may be Read-Only-Memory or Random-Access-Memory. Each time it fetches an instruction from memory, the processor increments the IP by as many bytes as necessary to point to the next instruction. Therefore, the IP always indicates the next instruction byte to be fetched. This process continues as long as program instructions are executed sequentially.

To alter the flow of program execution as with a jump instruction or a call to a procedure, the processor overwrites the current contents of the IP with the address of the new instruction. In the case of a call, the processor saves the old contents of the IP on the hardware stack to enable the return from the procedure. The next instruction fetch occurs from the new address.

(If the new address is not in the 64K bytes above the current contents of the code segment register (CS), then the contents of CS must also be replaced by using an intersegment jump. This is generated by the assembler when you transfer to a label or procedure which you declared to be of type FAR. This point is discussed further in several places: ADDRESSING, the ASSUME statement, and Chapter 6.)

Instruction Pointer Wraparound

All addresses are the result of a positive offset address taken from a segment register. It is generally not possible to access an address lower than the contents of a given segment register using the segment register.

All offset arithmetic is 16-bit, done modulo 64K. One consequence is that if you add 1 to the highest possible offset, you get an offset of 0:

```
 0FFFFH
+    1
------
 0000H
```

The carry out of the high-order bit is unused. Thus, one byte beyond the highest address in a segment, you find yourself at the lowest possible address in that segment. This is sometimes termed WRAPAROUND.

This fact is used in generating the displacement in certain self-relative jumps. For example, a jump instruction located near the high end of a code segment, say at offset 0EFFDH, may need to transfer control to an instruction near the beginning of that code segment, say at offset 5. The source line is written exactly the same as if the target location were closer, say at only 257 bytes (= 0101H) bytes after the jump.

If TARGET_LABEL_NAME were 257 bytes after the jump, the instruction:

```
JMP  TARGET_LABEL_NAME
```

would assemble as 0E90101H. When it is executed, the IP will already point to the next sequential instruction at 0F000H, i.e., 3 bytes past 0EFFDH where the JMP begins.

The 0101H is added to the IP, so that the instruction at offset 0F101H is executed next.

However, if TARGET_LABEL_NAME is located at offset 5, the assembler must generate a "distance" which, when added to the Instruction Pointer, will cause the IP to point correctly to the target location. It does this by adding the complement of the IP to the offset of the target. This effectively generates the correct offset to the base address in CS. This process is only necessary for JMPs within the same segment, which are self-relative, whereas intrasegment CALLS replace the IP rather than add to it. (Intersegment calls or jumps, to a different segment, always replace both CS and the IP.)

Example:

If CS holds the paragraph number 3456H, and the source line JMP TARGET_LABEL_NAME occurs at relative location (offset) 0EFFDH, and TARGET_LABEL_NAME is at offset 5, then the assembler will generate a self-relative distance of 1005H, making the instruction read JMP 1005H. The effect on the Instruction Pointer is then:

```
                    Instruction Pointer = 0F000H;
complement is 0FFFH + 1 =
                                          1000H

                          target's offset = 0005H;
                 self-relative distance used = 1005H
```

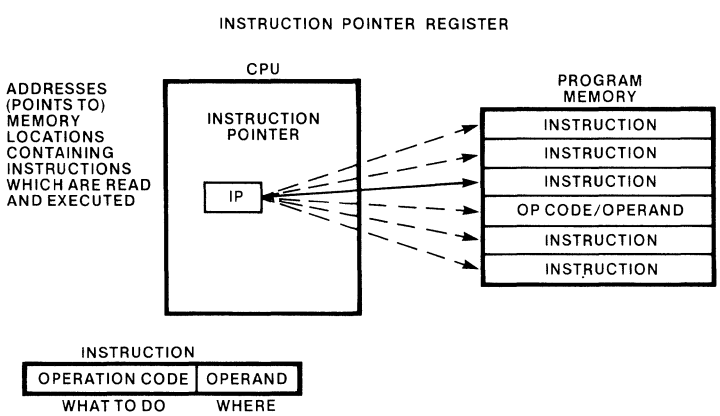
because the carry-bit out of the high order sum is ignored.

Thus the true offset of the target label, relative to the segment's base address in CS, is generated correctly, since when 1005H is added to the IP during the JMP, 0F000H, the target's true offset from the segment's base address is correctly generated.

Your source line:

```
JMP  TARGET_LABEL_NAME
```

is all you have to code. There is no complication of how you write programs. The assembler automatically generates the correct displacement to reach your indicated target, even when the "apparent distance" is very large, using the above mechanism.



Segment Registers. The (CS, DS, SS, ES) register set, called the Segment Registers, are used in ALL memory address computations (but not for I/O, as discussed at the end of this chapter). The segment mnemonics are:

CS:	Code
DS:	Data
SS:	Stack
ES:	Extra

The contents of a segment register is a word called the paragraph number or the segment-base-address. It represents a unique address in the one million bytes the 8086 can access. The special handling given to the contents of a segment register was explained above under Memory Addressing.

The paragraph number in the CS register defines the current CODE SEGMENT as the 64K bytes of addresses higher than that paragraph number. All instruction fetches are taken to be relative to CS, using the instruction pointer (IP) as an offset.

The paragraph number in the DS register defines the current DATA SEGMENT as the 64K bytes of addresses higher than that paragraph number. Most data reference hardware instructions use the DS register by default. One advantage to DS is that slightly shorter code will be generated.

Most references to data can be forced to be relative to one of the other three segment registers by preceding the data reference with a one-byte segment override prefix. In ASM86, the management of these prefixes is done automatically for you via the ASSUME directive (see Chapter 4).

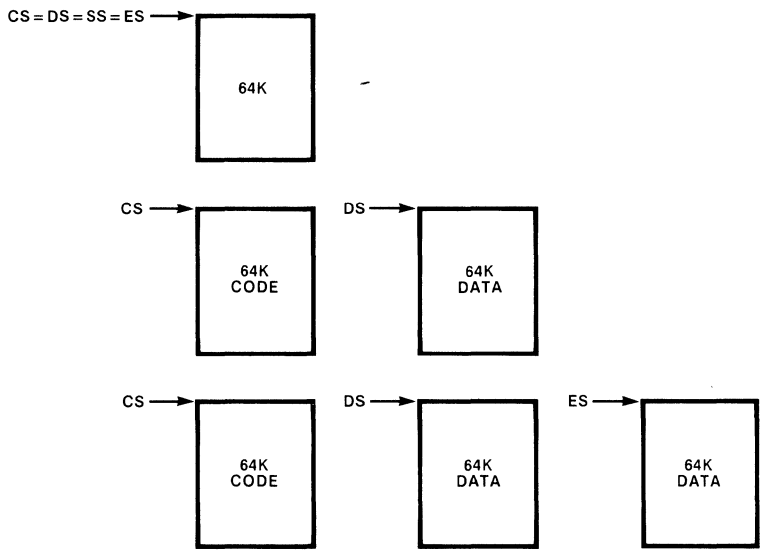
(There are three exceptions to the hardware's assumption of DS. When either SP or BP is used in an address-expression, the hardware assumes that register contains an offset to the stack segment (SS register). In certain string instructions, DI is an offset to the extra segments (ES register). BP can actually be used with any segment, using an appropriate override byte, but SP does unalterably refer to SS. DI, in those certain instructions, does unalterably refer to ES.)

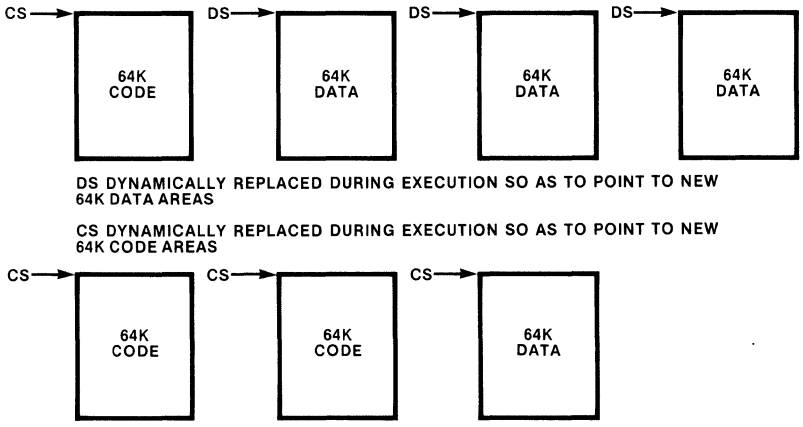
The current Stack Segment is defined as the 64K bytes of addresses higher than the contents of the SS register. Data references involving SP (and typically BP) are taken relative to SS. This includes all push and pop operations, including those caused by CALL operations, interrupts, and RETurn operations. Data references involving BP (but not SP) can be forced to be relative to one of the other segment registers by using the special one-byte base prefix discussed below under Overrides.

The current Extra Segment is defined as the 64K bytes of addresses higher than the contents of the ES register. The extra segment is usually created as an additional data segment. String instructions which use DI apply its contents as an offset to the base-address in ES.

Programs which do not load or manipulate the segment registers and do not contain FAR labels or FAR procedures (see Chapter 2), are said to be dynamically relocatable. Such a program may be interrupted, moved to a new location, and restarted with new segment register values.

The uses of segment registers depends on the size of program code, data, as is illustrated in the figures below.


CS = DS = SS = ES
64K
CS
64K
CODE
DS
64K
DATA
CS
64K
CODE
DS
64K
DATA
ES
64K
DATA


CS
64K
CODE
DS
64K
DATA
DS
64K
DATA
DS
64K
DATA
DS DYNAMICALLY REPLACED DURING EXECUTION SO AS TO POINT TO NEW 64K DATA AREAS
CS DYNAMICALLY REPLACED DURING EXECUTION SO AS TO POINT TO NEW 64K CODE AREAS
CS
64K
CODE
CS
64K
CODE
CS
64K
DATA

In order for the automatic addressing described above (under Memory Addressing) to occur properly, you must load the segment registers with the paragraph numbers, i.e., the segment base-addresses, that you want used for each section of code. You do this by using the name of your segment, e.g.,

```
MOV AX,SEGNAM4
MOV ES,AX

MOV AX,DATA_SEC_3
MOV DS,AX
```

The names SEGNAM4 and DATA_SEC_3 represent these segment base- addresses and are defined by your use of SEGMENT directive (discussed fully in Chapter 4). The first such directive that the assembler saw bearing that name, i.e.,

```
SEGNAM4      SEGMENT
                o
                o
                o
SEGNAM4      ENDS
DATA_SEC_3   SEGMENT
                o
                o
                o
DATA_SEC_3   ENDS
```

ultimately defined the segment's starting point.

NOTE

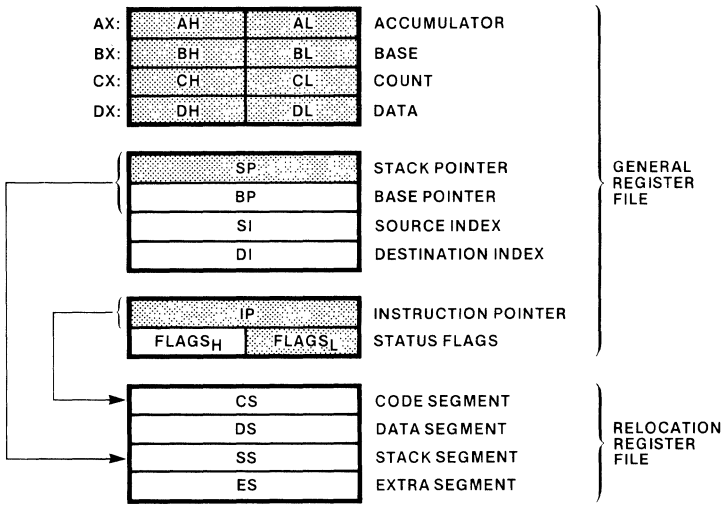
If you were to put consecutive paragraph numbers into some of the segment registers, e.g.,

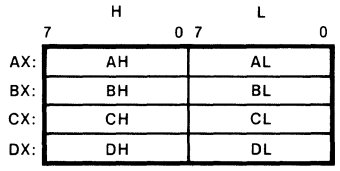
```
MOV AX, 1234H
MOV ES, AX

MOV AX, 1235H
MOV DS, AX

MOV AX, 1236H
MOV SS, AX
```

Then clearly many of the same addresses would be accessible using ES, DS, or SS. For example, a variable at the address 12378H might be reached using ES plus the offset 38H, or using DS plus the offset 28H, or using SS plus the offset 18H. The segments would in this sense overlap.

On the other hand, if the addresses you used were 1234H, 3234H, and 5234H, then none of the addresses accessible using ES or DS or SS would be the same. The segments would be disjoint, since the address 32340H is further away from 12340H than the maximum possible offset of 64K −1 bytes = 0FFFFH. There are 16 disjoint segments making up the megabyte, and 65536 overlapping segments.

In the discussions that follow, here and throughout the manual, the words "segment" and "base-address" and "paragraph-number" will effectively be synonyms whenever the text is referring to the beginning address of a segment.





General Registers. The (AX, BX, CX, DX) register set is called the General Register, or HL group. The general registers can participate in the arithmetic and logic operations of the 8086 without constraint. Some of the other 8086 operations (such as the string operations) dedicate certain of the registers to specific uses. These uses are indicated by the following mnemonic phrases:

AX: Accumulator

BX: Base

CX: Count

DX: Data

The general registers have a property that distinguishes them from the other registers, namely that their upper and lower halves are separately addressable. Thus, the general registers can be thought of as two sets of four 8-bit registers. These are called H and L, for high-byte and low-byte, i.e. AH, BH, CH, DH, and AL, BL, CL, DL.

The accumulator has an additional property: you get more compact programs by using it as the target of your data transfer, arithmetic, and logic instructions than when you use the other general-registers. Also, assembly language instructions whose destination is the accumulator can be abbreviated.

The remainder of the registers in the 8086 processor must be accessed as if containing 16-bit words, whether or not both their high-order and their low-order bytes are utilized.

Pointers and Indexes. The (SP, BP, SI, DI) register set is called the Pointer and Index Register (P and I) Group. The registers in this group are similar in that they generally contain offset or base addresses used for calculating addresses within some segment. Like the general registers, the pointer and index registers can participate in all the 16-bit arithmetic and logical operations of the 8086.

They are also similar in that they can enter into address computations, with one difference, however, which results in dividing this set into two groups, the P, or Pointer Group (SP, BP) and the I, or Index Group (SI, DI).

The difference is that the offset addresses in the Pointers are assumed by the hardware instructions to be relative to (use the base-address of) the current stack segment, and the offset addresses in the Indexes are assumed by the hardware to be relative to (use the base-address of) the current data segment. (Certain string operations listed in Chapter 6 are exceptions, using DI relative to the "extra segment" instead of the data segment).

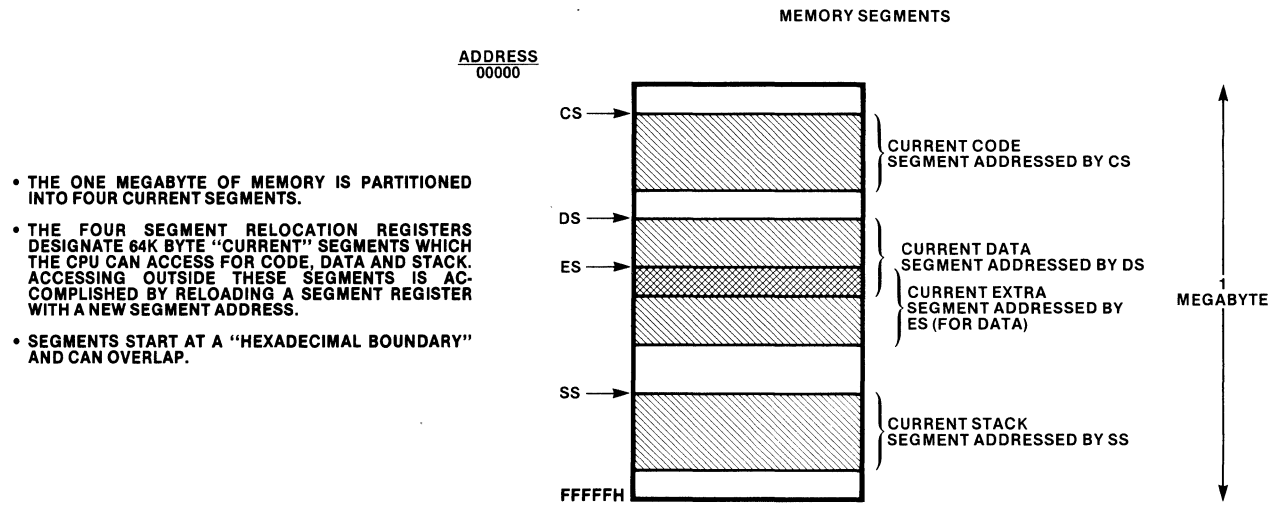
The mnemonics associated with these registers are:

SP: Stack Pointer

BP: Base Pointer

SI: Source Index

DI: Destination Index

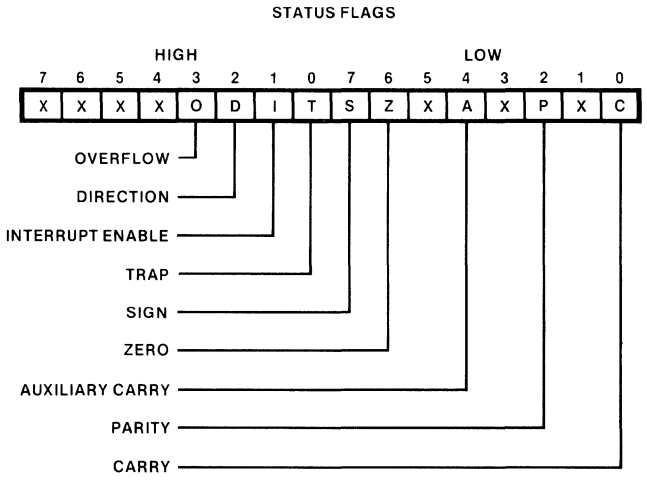


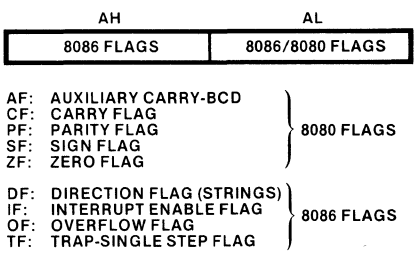
Flags Overview. The (AF, CF, DF, IF, OF, PF, SF, TF, ZF) register set is called the Flag Register or F group. The flags in this group are all one bit in size, and are used to record processor status information and to control processor operation. The details of their interpretation are given in the next section. The flag register mnemonics are:

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

The AF, CF, PF, and ZF flags are equivalent to 8080 flags, generally reflecting the status of the latest arithmetic or logical operation. The 8086 adds to this group as follows: The OF flag reflects the signed arithmetic overflow condition. The DF flag controls the direction of the string manipulation instructions (auto-incrementing or auto-decrementing). The IF flag enables or disables external interrupts. The TF flag puts the processor into a single-step mode for program debugging. Interrupt and trap mode are discussed in greater detail in the 8086 User's Manual (9800722).

The flag registers are illustrated in the figures below in the format in which they are stored by push-flag operation (PUSHF). The bit positions marked X are undefined. The figures show the equivalence of 8080 flags to those of 8086. An additional difference from the MCS-80 family is that the content of the accumulator is not transferred to and from the stack by flag operations. Fuller detail on flag usage appears in Appendix C.





Addressing Modes

The 8086 instruction set provides several different ways to address operands. In 2-operand instructions, the source (rightmost) operand may generally be an immediate-constant i.e., a value contained in the instruction itself. When an immediate-constant is the source, then the destination (left) operand may be either a register or a location in memory. Otherwise one of the 2 operands MUST be a register. The other may be either a register or a location in memory.

The examples below illustrate these points. The first three statements define symbols used in the 2-operand instructions below them.

```
MFG_DEPT_ID              EQU  5
    EXMP                 SEGMENT
ON_HAND                  DB   0
                         DB   2
                         DB   4
ITEM_COUNT               DB   0
                         DB   17
                         DB   19
REPORTING_DEPARTMENT     DB   0
            EXMP  ENDS

    MOV  AL,4  ; destination, register AL, receives immediate-value 4

    MOV  ITEM_COUNT, 14  ; destination, memory location ITEM_COUNT,
                ; gets immediate-value 14

    MOV  BL,ITEM_COUNT  ; register BL is filled
                ; with the contents of memory location ITEM_COUNT

    ADD  BL,ON_HAND  ; contents of memory location ON_HAND
                ; are added onto the contents of register BL

    MOV  REPORTING_DEPT,MFG_DEPT_ID  ; memory location
                ; REPORTING_DEPT is filled by immediate-value 5
```

Operands in memory may be addressed directly, e.g., by a simple name as above, or indirectly using registers and/or subscripts. Direct reference involves a simple 16-bit offset, automatically added by the hardware to the address in a segment register.

Indirect reference involves either one or two of the four registers allowed within square brackets [indicating an indirect reference]: BX, BP, SI, or DI. (If a variable is named too it must precede the square-brackets expression.) BX and BP are called Base registers. SI and DI are called Index registers. An indirect reference may involve a single base register alone, a single index register alone, or one base and one index register. An indirect reference may also include an 8 or 16 bit displacement.

The assembly language address expressions which result in these 4 kinds of memory access are described in Chapter 5 on Expressions. Only those address-expressions which result in the feasible addressing modes (see Table below) are valid.

Operands residing in memory may be thus addressed in four ways:

- Direct 16-bit offset address
  Example: `MOV  REPORTING_DEPARTMENT, AL`
- Indirect through a base register, optionally summed with an 8- or 16-bit displacement
  Example:
  ```
  MOV  ON_HAND [BX+2], AL
  MOV  BL, ON_HAND [BP]
  ```

- Indirect through an index register, optionally summed with an 8- or 16-bit displacement
  Example: MOV CL, ITEM_COUNT [SI+1]
  MOV ON_HAND [DI+1], CL
- Indirect through the sum of a base register and an index register, optionally summed with an 8- or 16-bit displacement.
  Example: MOV AH, ITEM_COUNT [BX+1] [SI+1]
  MOV ON_HAND [BX+1] [DI+1],AH

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) always 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.

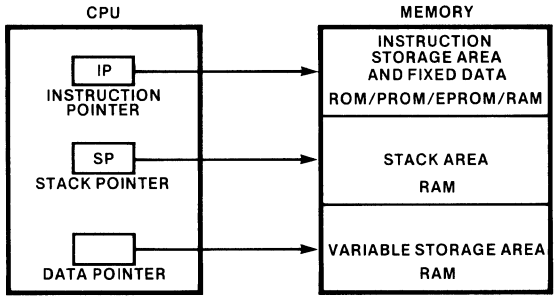
The r/m field either can be the location of the operand (if in a register) or can specify how the 8086 will locate the operand in memory, in combination with the mod field as shown below.

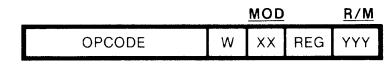
These fields are set automataically by the assembler in generating your code. They are discussed in greater detail in Chapter 7 on Code macros. The effective address (EA) of the memory operand is computed according to the the mode and r/m fields:

```
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.

TYPICAL MEMORY USAGE





32 COMBINATIONS

XX

MOD	SELECTED MODE
11	REGISTER MODE
10	D16 DISPLACEMENT
01	D8 DISPLACEMENT
00	NO DISPLACEMENT

REGISTER TO REGISTER MODE USES 8 OF 32 COMBINATIONS OF MODE - R/M, AND W BIT SELECTS BYTE OR WORD

YYY

(XX = 11)

R/M	MEMORY MODE	REGISTER MODE	
		BYTE	WORD
111	(BX)	BH	DI
110	(BP)	DH	SI
101	(DI)	CH	BP
100	(SI)	AH	SP
011	(BP) + (DI)	BL	BX
010	(BP) + (SI)	DL	DX
001	(BX) + (DI)	CL	CX
000	(BX) + (SI)	AL	AX
		W=0	W=1

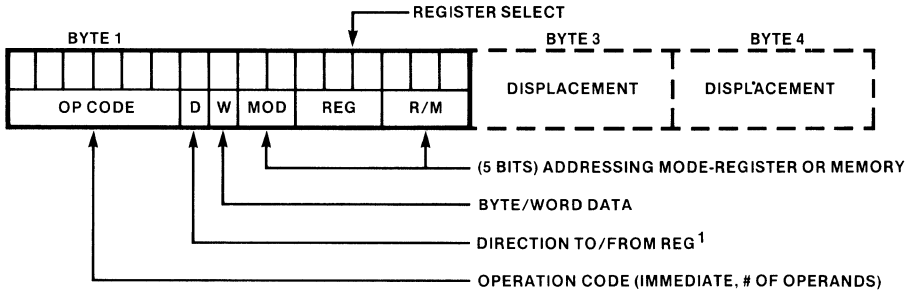
ADDRESS MODE ENCODINGS FOR MOD AND R/M

R/M	MOD				
	00	01	10	11	
				W = 0	W = 1
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16	AL	AX
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16	CL	CX
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16	DL	DX
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16	BL	BX
100	(SI)	(SI) + D8	(SI) + D16	AH	SP
101	(DI)	(DI) + D8	(DI) + D16	CH	BP
110	DIRECT ADDRESS	(BP) + D8	(BP) + D16	DH	SI
111	(BX)	(BX) + D8	(BX) + D16	BH	DI

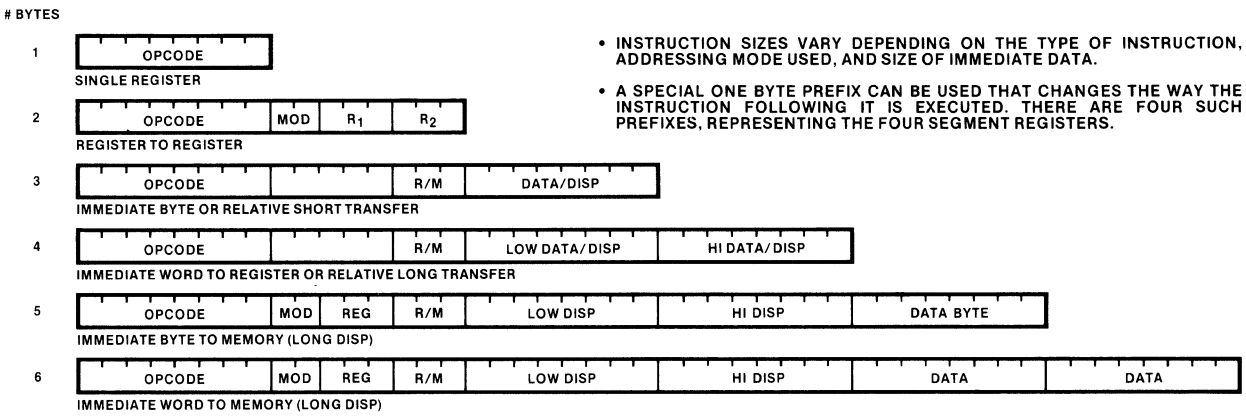
ADDRESS COMPUTATION REVIEW

DISPLACEMENT	0, D8, D16
+	
BASE REGISTER	BX, BP
+	
INDEX REGISTER	SI, DI
+	
RELOCATION REGISTER	CS, DS, ES, SS
MEMORY ADDRESS	

TYPICAL 8086 INSTRUCTION FORMAT



[1] 0 = FROM REG
1 = TO REG



- INSTRUCTION SIZES VARY DEPENDING ON THE TYPE OF INSTRUCTION, ADDRESSING MODE USED, AND SIZE OF IMMEDIATE DATA.
- A SPECIAL ONE BYTE PREFIX CAN BE USED THAT CHANGES THE WAY THE INSTRUCTION FOLLOWING IT IS EXECUTED. THERE ARE FOUR SUCH PREFIXES, REPRESENTING THE FOUR SEGMENT REGISTERS.

Segment-Register Defaults in the Hardware

General registers 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. When an index register alone is used, the operand by default resides in the current data segment. As mentioned above, you can override the above defaults through use of a segment prefix byte described in Chapter 5 and later in this chapter. Three defaults that cannot be overriden are:

- execution is only performed on instructions accessed by the Instruction Pointer acting as an offset to the CS register
- SP acts as an offset to the SS register only
- Those certain string instructions which use the ES register cannot use a different segment register instead.

Segment Override Prefixes

Since every instruction that deals with memory uses a segment register, the assembler must decide which one is appropriate for each reference. The address- expression in the source line determines this, as discussed in Chapter 5.

The physical address of most other memory operands is by default computed using the DS segment register. These default segment register selections may be overridden by your preceding the referencing instruction with a segment override prefix, or allowing the assembler to do it for you, as discussed in Chapter 4.

The segment register selected by the reg field below is used by the 8086 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 may not return appropriately from an interrupt if multiple prefixes are present.

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

- Overrides implied segment register in next instruction's data reference.

```
                INST
                INST
                INST
SEGMENT         INST
OVERRIDE        R
                INST
                INST
```

- Interrupts are not accepted between prefix and next instruction.
- Code fetch always made to CS (using IP).
- Stack fetch always made to SS (using SP).

Example:

```
MOV AL,ES:XYZ
MOV ES:PDQ, AL
```

Controlling Segment Override Prefixes. When a segment override is necessary, it must be specified individually in each data reference. Or, segment override prefixes can be declared once with an ASSUME statement, and all references will automatically generate the correct prefix. The ASSUME directive is described fully in Chapter 4.

The ASSUME declaration associates a segment register with a segment name. All references to items in the named segment cause segment override prefixes to be generated if necessary.

Example:

```
MORSTUFF  SEGMENT
XYZ       DB      1
PDQ       DW      0
FOO       DD      2
MORSTUFF  ENDS

                  ASSUME ES:MORSTUFF
                  MOV AL,XYZ

is equivalent to: MOV AL, ES:XYZ
```

Segment override prefixes can be symbolic segment names, if the segment names appear in a prior ASSUME statement.

Example:

```
                  ASSUME ES:SEG1
                  MOV AL, SEG1:XYZ
                  .
                  .
                  .
is equivalent to:
                  MOV AL,ES:XYZ
```

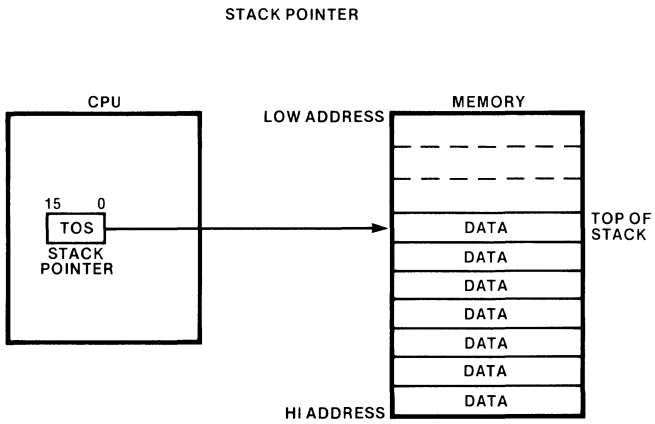
See the discussion of the ASSUME directive in Chapter 4. and of the override operator in Chapter 5.

Stack and Stack Pointer. A stack is an area of memory used for storing values temporarily. It is available via its own segment register, SS. Thus the same stack can be used by different code segments, which must use unique values in CS and often also in DS and ES.

The use of a stack segment depends on SP, the stack pointer. You must set SS to hold that segment's base-address, and set SP to the highest offset in the stack segment, e.g.,

```
MOV  AX,  STACK_SEG_NAME1
MOV  SS,  AX
MOV  SP,  LAST_WORD_SS1
```

This is because the stack expands by decrementing the stack pointer. As items are added to the stack, via PUSH or CALL, the stack expands into memory locations with lower addresses toward the stack's base address. As items are removed from the stack, via POP or RETurn, the stack pointer is incremented back toward its highest value, furthest from the base-address. The most recent item on the stack is known as the "top of the stack" (TOS).



"Stack" is a most descriptive term because you can always put something on top of the stack, like a dish stacker. You can PUSH a new dish (word) on top, or POP the last one on top off into a destination you specify. In terms of programming, a procedure can call a procedure, and so on. The only limitations to the number of items that can be added to the stack are the amount of RAM available and the 64K limit on segments.

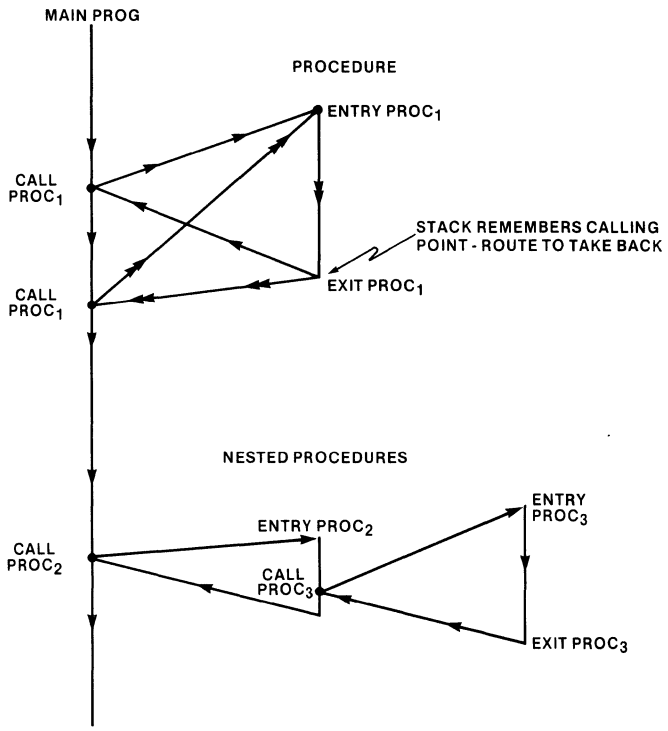
To appreciate the purpose and effectiveness of a stack, it is useful to explore this concept of a procedure.

Assume that your program requires a special routine several times. You can recode this routine each time it is needed, but this can use a great deal of memory. Or, you can code a procedure:

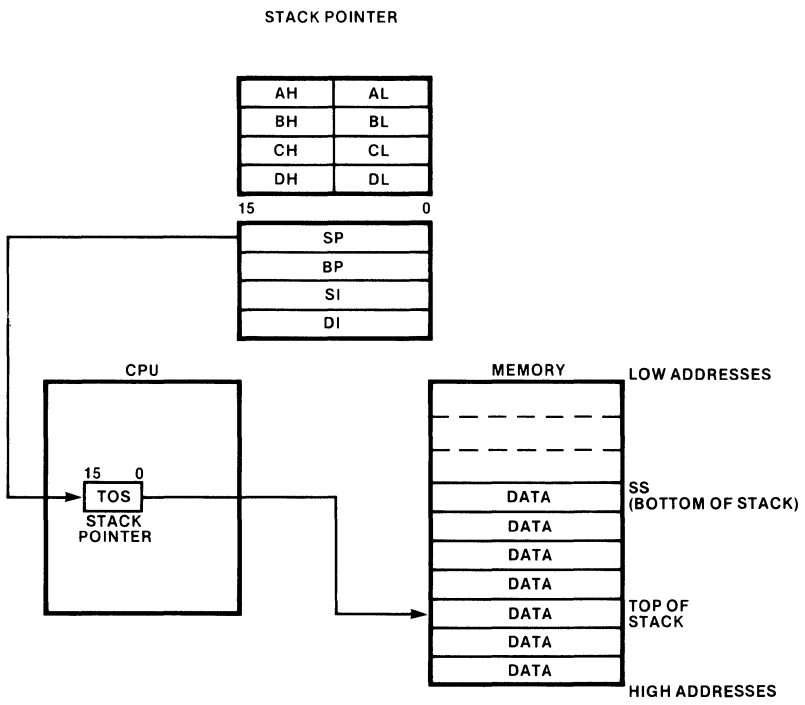
Inline Coding	Use of Procedure
o	o
o	o
o	o
inline-copy-of-PROC1	CALL PROC1
o	o
o	o
o	o
inline-copy-of-PROC1	CALL PROC1
o	o
o	o

The 8086 provides instructions you can use to CALL and RETurn from a procedure. When the CALL instruction is executed, the address of the next instruction (the contents of the instruction pointer) is automatically PUSHed onto the stack. The contents of the instruction pointer are replaced by the address of the desired procedure. At the end of the procedure, a RETurn instruction POPs that previously-stored address off the stack and puts it back into the instruction pointer. Program execution then continues as though the procedure had been coded inline. The mechanism that makes this possible is a stack.

PROCEDURE PROGRAM FLOW



- PROCEDURES REDUCE THE AMOUNT OF PROGRAM MEMORY SPACE USED, BY ALLOWING COMMON SEQUENCES TO BE EXECUTED FROM DIFFERENT POINTS IN THE PROGRAM.



The stack is also used for temporary storage of parameters for use in such a procedure. Before the CALL, the calling routine can first PUSH the data onto the stack. The called routine can then simply access the stack directly. Though this has certain advantages, it leaves the data sitting on the stack, violating the normal expectation that the RETurn will leave the stack as it was before the calling program used it. In this situation the called routine can end with the statement:

```
RET 8
```

which causes the stack pointer to be incremented by that number, effectively skipping over ("popping off") those words (4 in this case) for any subsequent stack operations. Since stack operations always involve WORDS, the number, if any, placed on a RETurn instruction must be two times the number of items to be skipped. Three parameters would mean a RET 6.

Stack Operations. As mentioned above, stack operations transfer sixteen bits of data between memory and a destination register or destination memory word. The two basic operations are PUSH, which adds data to the stack, and POP, which removes data from the stack.

A CALL instruction PUSHes the contents of the instruction pointer (which contains the address of the next instruction) onto the stack and then transfers control to the desired procedure by placing its address in the instruction pointer. A RETurn instruction POPs the word off the top of the stack and places it in the instruction pointer. This requires the programmer to keep track of what is in the stack. For example, if you call a procedure and the procedure PUSHes data onto the stack, the procedure must remove that data before executing a return instruction. Otherwise, the return instruction POPs data from the stack and places it in the instruction pointer. The results are unpredictable, of course, and probably not what you want.

Saving Program Status. It is likely that a procedure requires the use of one or more of the working registers. However, it is equally likely that the main program has data stored in the registers, which it will need when control is returned to it. As a general rule, a procedure should save the contents of a register before using it and then restore the contents of that register before returning control to the main program. The procedure can do this by PUSHing the contents of the registers onto the stack and then POPping the data back into those registers before executing a return. It is important to restore (POP) them in the opposite order from their saving, e.g.,

```
PUSH  BX
PUSH  CX
.
.
.
.
POP   CX
POP   CS
```

A pair of procedures could be written to do this, named, say, SAVE and RESTORE. Then all other procedures could simply call SAVE at the beginning and RESTORE at the end. See also Appendix D.

Input/Output. Input/output is done using addressable ports, either 1 byte or 1 word in size. There are 65536 such port addresses, reflecting the fact that I/O space is addressed using a 16-bit address. Segment registers are not used.

The input/output ports provide communication with the outside world of peripheral devices. The IN and OUT instructions initiate data transfers.

The IN instruction latches the number of the desired port onto the address bus. As soon as a byte (or word) of data is returned to the data bus latch, it is transferred into the accumulator, AL (or AX).

The OUT instruction latches the number of the desired port onto the address bus and latches the data in AL (or AX) onto the data bus.

Notice that the IN and OUT instructions simply initiate a data transfer. It is the responsibility of the peripheral device to detect that it has been addressed. Notice also that it is possible to dedicate any number of ports to the same peripheral device. You might use a number of ports as control signals, for example.

Because input and output are almost totally application dependent, a discussion of design techniques is beyond the scope of this manual. For additional hardware information, refer to the *8086 Microcomputer Systems User's Manual*(9800722).

The instructions IN and OUT each have 2 forms. You may specify DX, as in OUT DX, AX or OUT DX, AL using the contents of the DX register as the address of the port. Using DX you have 65536 ports.

Alternatively, you may specify an immediate byte operand as the address of the port in which case you have 256 possible ports.

In either case the accumulator is the source for output or the destination for input. When AX is specified, a word is input or output. When AL is specified, a byte is input or output.

I/O Device Selection.

- IN/OUT port numbers can be designated with 8 bit literals in the instruction (0-255).
- IN/OUT port numbers can be contained in a word register, (0-64K), DX.
- IN/OUT ports can transmit bytes (8 bits) or words (16 bits).
- Byte I/O ports can communicate on the low (D0-D7) data bus lines or the high (D8-D15) data bus lines.
- Even addressed I/O ports transfer data on low (D0-D7) data bus lines.
- Odd addressed I/O ports transfer data on high (D8-D15) data bus lines.

WARNING

Care must be exercised that each register within an 8 bit peripheral chip is addressed by all even or odd addresses.

Interrupt Procedures—(See also Appendix D)

The 8086 language supports two types of interrupts, external and internal. An external interrupt is initiated by some peripheral asserting an interrupt request to the 8086 in the hardware (refer to the *MCS-86 User's Manual* for details). An internal interrupt is one initiated by the software the 8086 is executing. An interrupt represents a transfer of program execution control. The type of transfer used in the 8086 is called a vectored interrupt. An interrupt vector represents an address of a procedure which services the interrupt.

In the 8086, all interrupts (both external and internal) perform a transfer by pushing the flag registers onto the stack (as in PUSHF), and then performing an indirect call (of the intersegment variety) through an element of an interrupt vector located at absolute memory locations 0 through 3FFH. Each vector is a four byte element with the first two bytes containing the offset of a procedure (or label) and the second two bytes containing the paragraph number of the segment containing the procedure (or label). There are 256 possible interrupt vectors. Within the 8086 assembly language, each vector is given a number from 0 through 255. Intel Corporation reserves the use of interrupts 0 through 31 (locations 0 through 7FH) for Intel hardware and software products. Users who wish to maintain compatibility with present and future Intel products should not use these locations except as defined by Intel. Interrupts 0 through 4 (0 - 13H) currently have the dedicated hardware functions as defined below.

Interrupt #	Location	Function
0	0-03H	divide by zero
1	04H-07H	single step
2	08H-0BH	non-maskable interrupt
3	0CH-0FH	one byte interrupt instruction (INT 3)
4	10H-13H	interrupt on overflow

There are three interrupt transfer operations provided:

— INT pushes the flag registers, clears the TF and IF flags, and transfers control with an indirect call through any of the 256 vector elements, i.e., INT 24 will do an indirect call through interrupt vector 24 (location 96). A one byte form of this instruction is available for interrupt type 3, INT 3.

— INTO pushes the flag registers, clears the TF and IF flags, and transfers control through vector element 4 if the OF flag is set (interrupt on overflow). If the OF flag is cleared, then no operation takes place.

— IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers.

The following example illustrates the means of setting up the interrupt vectors and the procedures which service the interrupts. An absolute segment is defined at location 0 which contains the interrupt vectors (see the chapter on data initialization and directives for details concerning the following constructs.)

```
INT_VECTORS  SEGMENT  AT  0

             ORG  0CH
             DD   TYPE_3_PROC     ;interrupt type 3
             ORG  14H
             DD   TYPE_5_PROC     ;interrupt type 5
                  *
INT_VECTORS  ENDS
```

In the above example the DD TYPE_3_PROC will store the address (the offset and paragraph number) of the procedure called TYPE_3_PROC.

The interrupt procedures themselves will then appear in another user-defined segment elsewhere in the program.

```
INT_PROCS  SEGMENT

    TYPE_3_PROC  PROC  FAR
                 *
                 *       ;the code for the procedure
                 *
                 IRET
    TYPE_3_PROC  ENDP

    TYPE_5_PROC  PROC  FAR
                 *
                 *
                 *
                 IRET
    TYPE_5_PROC  ENDP

INT_PROCS  ENDS
```

Somewhere in the program you can then specify the following code.

```
        *
        *
        INT  3  ;will cause the execution of TYPE_3_PROC
        *
        INT  5  ;will cause the execution of TYPE_5_PROC
        *
        *
```

For external interrupts, the peripheral device will request an interrupt from the 8086. When the 8086 grants the interrupt, the device will supply a byte value on the data bus which represents the type or number of the interrupt i.e., 0 through 255. The 8086 will read this value and then execute the interrupt through the vector. In the above example the procedure TYPE_5_PROC can be executed either through the instruction INT 5 in the software or by a device requesting an interrupt from the 8086 and then putting the value 05H on the data bus. (See also Appendix D.)

CHAPTER 2
BASIC CONSTITUENTS OF AN 8086 ASSEMBLY LANGUAGE PROGRAM

This chapter discusses the elements that constitute a program in the 8086 assembly language. The topics include:

Introduction

There is a legitimate distinction between the 8086 Assembly Language and the ASM86. The latter is a program that recognizes and translates the language into object (machine) code. This manual, however, uses the two terms more or less interchangeably.

Programs in the 8086 Assembly Language are written free-form. That is, the input (source) lines are column-independent and blanks may be freely inserted between the elements of the program. The only exception is a continuation line, which must have an ampersand (&) immediately following the terminator of the previous line.

ASM86 Character Set

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

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

along with these special characters

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

and the non-printing characters

space tab carriage-return line-feed

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

Upper- and lower-case letters are not distinguished from each other (except in string constants—see below). For example, xyz and XYz are interchangeable. In this manual, all ASM86 code is in upper-case letters to help distinguish it visually from explanatory text.

Blanks are not distinguished from each other (except in string constants). Any blank is considered to be the same as any other blank. Moreover, any unbroken sequence of blanks is considered to be the same as a single blank.

Special characters and combinations of special characters have particular meanings in an ASM86 program, as described in the remainder of this manual.

Syntactic Elements of ASM86

Tokens and Separators

A token is the smallest meaningful unit of a ASM86 source program, much as words are the smallest meaningful units of a book in English. Separators are used to separate two adjacent tokens so that they are not mistakenly thought to be one longer token. The most commonly used separator is the blank ( ). Blanks are not distinguished from each other (except in string constants). Any unbroken sequence of blanks may be used wherever a single blank is allowed. Horizontal tabs are also used as separators and are interpreted by the assembler identically to blanks except that they may appear as multiple blanks in the list file (see operator's manual). Any illegal character, or character used in an illegal context, is also treated as a separator.

Blanks may thus be inserted freely around any token, without changing the meaning of the ASM86 statement. Thus the statements:

```
MOV   ITEM, [BX+3]
MOV           ITEM,[BX+      3]
```

are identical insofar as the meaning to the assembler is concerned.

Delimiters

Delimiters are special characters that serve to mark the end of a token and also have a special meaning unto themselves (as opposed to separators, which merely mark the end of a token). In the sample statements above, the comma, the plus-sign, and the square brackets all serve as delimiters. When a delimiter is present, separators need not be used; however, using separators often makes your programs easier to read and, therefore, easier to understand in ASM86.

The table below describes the special usages of the delimiters and separators in ASM86:

Character(s)	English Name	Use
20H 09H	blank or blanks horizontal tab	Separate or terminate tokens; enhance program readability
,	comma	Separate an operand from a preceding one when more than one is present
'...'	pair of single quotes	Delimit a character string
(...)	pair of matched parentheses	Delimit an expression or subexpression, often to enhance program readability or to alter operator precedence
0DH (CR) 0AH (LF) CR-LF	carriage-return line-feed carriage-return/ line-feed pair	Statement terminator (unless immediately followed by a '&')
;	semicolon	Comment field delimiter
:	colon	Delimiter for symbols used as labels, segment overrides, macro specifiers, extrn types, assume fields, record field definitions
.	period or dot	Selector for a field from a record; only allowed within codemacros
&	ampersand	Continuation line indicator, when immediately following a statement terminator
<...>	pair of matched angle brackets	Indicates the enclosed values are to be used to initialize a record within a codemacro
$	dollar sign	Shorthand notation for "The present value of the location counter"
[...]	pair of matched square brackets	Encloses an index or pointer (subscript) expression
=	equal sign	Separates field width specification from (optional) default initial value
-	minus sign	Between 2 operands, indicates subtraction; alone to the left of an operand, indicates negative value
+	plus sign	Between 2 operands, indicates addition; alone to the left of an operand, indicates positive value
*	star, asterisk, times sign	Indicates multiplication
/	slash, division sign	Indicates division operation
?	question mark	Indicates an unspecified value to be used to initialize storage; may also be used with other characters to form identifiers
@	commercial at-sign	Used to form identifiers
_	underscore	Used to form identifiers

While many of the usages and terms used above may be new to you, they will be explained in subsequent chapters of this manual.

Numeric Constants

A constant is a value known at assembly-time, which does not change during execution. A constant may either be a whole-number (integer) or a character string. Whole-number constants may be expressed as a binary (base 2), octal (base 8), decimal (base 10), or hexadecimal (base 16) number.

A binary number consists of a sequence of the digits (0, 1) terminated by a "B". Examples:

```
 0B
 1B
 01101010101B
 1111111111111111B
-0001000b
```

An octal number consists of a sequence of the digits (0,1,2,3,4,5,6,7) terminated by an "O" or a "Q". Examples:

```
 1234567o
-3Q
 377O
 0q
```

A decimal number consists of a sequence of the digits (0,1,2,3,4,5,6,7,8,9) terminated by a "D" (or a blank). Examples:

```
 10000
-6d
 65535
```

A hexadecimal number consists of a sequence of hexadecimal digits and letters (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F) terminated by an "H". Furthermore, the constant must begin with a digit and not a letter in order to enable ASM86 distinguish it from an identifier (see below). It is sufficient to place a "0" before the leading character of a hexadecimal constant if it is a letter. Examples:

```
 1H
-600H
 0FFH
-0AAAh
```

All numbers must be representable in 17-bits (including one bit for the sign of the number), i.e.,

```
-1111111111111111B <= binary constant       <= 1111111111111111B
        -177777Q <= octal constant          <= 177777Q
           -65535 <= decimal constant       <= 65535
         -0FFFFH <= hexadecimal constant<= 0FFFFH
```

Character Strings

Character strings are denoted by printable ASCII characters enclosed within apostrophes ('). Blanks and horizontal tabs are also allowed within strings but carriage-returns and line-feeds are not. The assembler represents the character string as a sequence of bytes containing the ASCII code for each character within the string. (The ASCII encoding consists of 7 bits.) Examples of character strings are:

```
'ABCDEFG'
'This is a string'
'Let''s include a '' ' '' character'
'123456!''#$%&()_-'
```

Notice that the third string includes two consecutive apostrophes within the string to represent the presence of a single apostrophe.

Strings of length 1 translate to single-byte values; strings of length 2 translate to word values. For example:

```
'A' is equivalent to 41H
'Ag' is equivalent to 4167H
'#' is equivalent to 23H
```

Character strings longer than two characters may only be used to initialize storage (see Chapter 3). One-character strings may be used any place a one byte immediate value may be used; two character strings may be used any place a one word immediate-value may be used.

Strings may not exceed the maximum length described in the MCS-86 Assembler Operator's Manual for ISIS-II Users (9800641).

Identifiers

Identifiers are sequences of characters which have a special, symbolic meaning to the assembler. All identifiers in ASM86 must obey the following rules:

1. The first character must be alphabetic (A,..,Z,a,...,z) or one of the special characters, @, _, or ?. (The question mark, however, may not be used alone as an identifier.)
2. Any subsequent characters can be either a character as mentioned in rule 1 above or a numeral (0, 1,..., 9).
3. Identifiers are unique within the first 31 characters; subsequent characters are ignored.

Thus the following are examples of valid identifiers:

```
A
WORD
FFFFH
Third_Street_and_Main
Should_We_Jump?
@variable_number_1234567890123456
@variable_number_1234567890123457
```

Notice that the last two identifiers will be treated as the same identifier by the assembler since they are identical in their first 31 characters.

Examples of invalid identifiers are:

```
First$Id ('$' is not allowed as part of an identifier)
0FFFFH (this is a number since it begins with a number)
'Memphis' (characters enclosed by apostrophes constitute a string, not an
          identifier)
```

Keywords

One of the examples of an identifier in the section above was WORD. This is a keyword. A keyword is an identifier that has a pre-defined meaning to the assembler. In general, keywords may only be used in their special context, unless they are "purged" first, and not all keywords may be purged.

```
AX
Segment   (remember, upper- and lower-case letters are equivalent)
END
MOV
```

For a complete list of keywords, see Appendix E.

Symbols and Their Attributes

A symbol is an identifier defined by you, the user, in order to represent certain locations in memory, or data, expressions, or code (or data) constructs. The assembler also predefines the register set and one segment for you. (You may redefine register symbols if you desire.) Symbol is used here in a very specific sense, and not generically as a name for all tokens.

Symbols may be divided into five categories:

1. registers
2. variables
3. labels
4. numbers
5. others (segments, groups, records, record fields, codemacros, formal parameters)

Each symbol carries with it certain attributes which allow the assembler to use that symbol to represent the desired information and distinguish aspects of its intended usage.

Registers

Each register belongs to a class as shown in Chapter 1. These classes determine whether the register is considered to be a byte or word (two byte) register. The 8086 flags are also considered one-bit registers. The following table partitions all the predefined 8086 registers:

CLASS	REGISTERS INCLUDED	SIZE	ALSO KNOWN AS
H	AH	1byte	Accumulator-High-Byte
	BH	"	Base-RegisterHigh-Byte
	CH	"	Count-Register-High-Byte
	DH	"	Data-Register-High-Byte
L	AL	"	Accumulator-Low-Byte
	BL	"	Base-Register-Low-Byte
	CL	"	Count-Register-Low-Byte
	DL	"	Data-Register-Low-Byte
X	AX	2 bytes	Accumulator (full word)
	BX	"	Base Register "
	CX	"	Count Register "
	DX	"	Data Register "
P	BP	"	Base Pointer
	SP	"	Stack Pointer

CLASS	REGISTERS INCLUDED	SIZE	ALSO KNOWN AS
I	SI	"	Source Index
	DI	"	Destination Index
S	CS	"	Code-Segment-Register
	SS	"	Stack-Segment-Register
	DS	"	Data-Segment-Register
	ES	"	Extra-Segment-Register
F	AF	1 bit	Auxiliary-Carry-Flag
	CF	"	Carry-Flag
	DF	"	Direction-Flag
	IF	"	Interrupt-Enable-Flag
	OF	"	Overflow-Flag
	PF	"	Parity-Flag
	SF	"	Sign-Flag
	TF	"	Trap Flag
	ZF	"	Zero Flag

General purpose registers (X, H, L classes) may be used as "source" or "destination" for most instructions (see below and Chapter 6). Some registers have special usages (e.g., BX is used in calculating addresses, CL is used as a count register for some instructions, etc.). Flags are not individually addressable but may be altered individually by the outcome of certain arithmetic, logical, or relational instructions; or collectively, by the outcome of certain data transfer instructions.

Variables

Variables are used to identify data which is residing at a particular location or locations in memory. All variables have three attributes:

1. Segment (which segment was being assembled when this variable was defined)
2. Offset (how many bytes there are between the beginning of the segment and the location of this symbol)
3. Type (how many bytes of data are manipulated when this variable is referenced)

As described in Chapter 1, segments start on any one of 64K segment boundaries. One of these values is used as the segment part of the variable's definition (although this number may not be available at assembly time). The offset of a symbol may be any number between 0 and 64K-1, inclusive. A variable must have one of the following types:

BYTE (1 byte long)
WORD (2 bytes long)
DWORD (4 bytes long)

Variables are generally defined by:

1. Apearing as the name for a storage initialization directive; for example:
   Variable_1 DB 0 (See Chapter 3)
2. Appearing as the name for a LABEL directive; for example:
   Variable_2 LABEL BYTE (See Chapter 4)
3. Appearing as the name for a EQU directive; for example:
   Variable_3 EQU Another_variable (See Chapter 4)

For a more detailed explanation of variables, see Chapter 3.

Labels

Labels represent locations in memory which contain instruction code and are intended to be referenced via jumps or calls. All labels have four attributes:

1. Segment (similar to the segment part of variables)
2. Offset (similar to the offset part of variables)
3. Distance (analogous to the type part of variables; indicates whether the label is reachable using a two byte offset or whether a segment-offset pair is needed (4 bytes))
4. CS-Assume (indicates what was ASSUMEd to be in the CS register when this label was defined)

Labels and variables are similar in meaning to the assembler, except that the former reference instruction code in memory, while the latter refer to data in memory. That is why their attributes are similar. The segment and offset parts of labels are defined similarly to those for variables. The distance part is one of two values:

NEAR: all references to this label use only a two byte, "self-relative" value, i.e., only the IP must be altered to reach this location and NOT the CS register's contents.

FAR: all references to this label require that both the IP and the CS registers be altered and that all jumps or calls to this label must specify new values for both.

Your choice of NEAR or FAR depends on whether this label is ever going to be referred to by jumps or calls whose CS: assumption is different from this label's CS: assumption. This usually means a jump or call from outside the code segment (or group) defining the label.

If all references use the same CS: assumption, then the label's distance attribute should be declared NEAR, otherwise FAR. If you say nothing, NEAR is assumed.

Choosing NEAR means you are telling the assembler that jumps to this label can always reach it with a self-relative offset of at most 16-bits, whereas FAR will always require such a reference to include a second word, giving the segment of this label as well as an offset.

If the assembler finds a reference which requires a "long" jump or call to a label you declared NEAR, it will flag an error.

Thus the CS-Assume attribute is crucial in the assembler's determining what sort of jumps and calls are required to transfer to the label. This concept is more fully explained under Label in Chapter 4.

Labels are generally defined by

1. Preceding an instruction, with a colon (:) between the label and instruction; for example:
```
Label_1:  ADD  AX,BX
```
2. Appearing as a name to a LABEL directive; for example:
```
Label_2  LABEL  FAR
```
3. Appearing as the name to an EQU directive; for example:
```
Label_3  EQU  THIS NEAR
```
4. Appearing as the name on a matching PROC/ENDP pair; for example:
```
Label_4  PROC
          .
          .
          .
Label_4  ENDP
```

For a more detailed explanation of labels, see Chapter 4.

Numbers

A symbol may be defined to represent a pure number rather than representing a register or a memory location (address). When this symbol is used, it is as if you had explicitly coded the number it represents, e.g.,

```
Number_5  EQU  5
MOV   AL, Number_5
```

is equivalent to writing:

```
MOV AL, 5
```

Number_5 has thus been defined as a symbol which does not represent a specific memory address but rather the value 5.

If a symbol is defined to be used as a number, it simply represents a 16-bit value (17 bits including sign bit during assembly). Operations on numbers are rather intuitive; a full explanation of operations on numbers is presented in Chapter 5. Remember that a one or two byte character string may also be used as the value of a number, e.g.,

```
Initials  EQU  'AA'
MOV   AX, Initials
```

is a perfectly legitimate usage of a number (4141H in this case).

For a further explanation of operators, expressions, and their relationship to numbers, see Chapter 5.

Other Symbols

Other symbols may be defined by appearing as the name to an assembler directive. The directives which require these symbols are:

SEGMENT/ENDS	(defines a segment name)
GROUP	(defines a group name)
RECORD	(defines both a record name and the names of any record fields contained within the record)
CODEMACRO	(defines both a CodeMacro and the name of any formal parameters used within the CodeMacro. Note that once a symbol has been defined via a CodeMacro, it is no longer considered an ordinary symbol but is considered an instruction mnemonic)
EQU	(in addition to defining variables and labels, EQU may be used to name expressions)

SEGMENT/ENDS, GROUP, RECORD, and EQU are all explained in detail in Chapter 4; CODEMACRO is explained in detail in Chapter 7.

A facility for referencing symbols not defined locally is provided in the EXTRN directive. This allows an arbitrary list of variables created in other modules to be referenced in the current module, and it requires the type (i.e., BYTE, WORD, DWORD, FAR, NEAR, or ABS—for numbers) to be specified with the name as well. For example:

```
EXTRN little:BYTE, medium:WORD, bit:DWORD, close:NEAR, distance:FAR, x:ABS
```

For a further explanation of the EXTRN directive, and its usage in linking program modules, see Chapter 4.

Statements

Just as tokens may be seen as the assembly language counterparts to the English concept of words, so may statements be viewed as analogous to sentences. A statement is a specification to the 8086 assembler as to what action to perform. In fact, one way of viewing a computer program is as a sequence of statements which, when taken as an aggregate, is intended to perform a particular function. Statements may be divided into two types:

Instructions: these are translated by the assembler into machine instruction code which "instruct" the 8086 to perform certain operations.

Directives: these are not translated into machine instruction code by the assembler but rather "direct" the assembler itself to perform certain clerical functions. (Note: the storage initialization directives DO cause information to be placed in the 8086's memory when the program is loaded; however, this is usually intended to be used as data, e.g., to provide initial values for variables, and is not intended to be "executed".)

Instruction mnemonics are either predefined by the assembler or defined by the user via CodeMacro directives (see Chapter 7 and Appendix A). The assembler, in fact, recognizes instructions ONLY if they have been so defined. Instruction mnemonics may be changed, added to, redefined, or "purged" at any time. (See Chapters 4 and 7.)

Directives, on the other hand, are permanent, built-in features of the assembler. They characterize ASM86 to a large degree. The assembler always takes the same, specific action when it encounters a directive keyword. Directives may be neither created nor destroyed.

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

There are several directives which MUST be encoded on more than one line. These are briefly described later in this section.

The format for encoding a statement is flexible in that (except for the restriction on the "&" used to denote continuation lines) tokens may appear anywhere on the source line. However, it is useful to think of the statement in terms of its constituent "fields" where certain kinds of symbols are constrained to appear. Furthermore, the general formats for instructions and directives are distinctive enough to require different descriptions of their fields.

The INSTRUCTION statement format is:

```
label    prefix    mnemonic(opcode)    operand(s)    comment
```

where the fields are defined as:

label: a symbol followed by a ":"; defines a label at the current value of the location counter in the current segment (see above and Chapter 4)
THIS FIELD IS ALWAYS OPTIONAL.

prefix — certain machine instructions may only be used as a prefix to other instructions (e.g., LOCK, REP).
THIS FIELD IS ALWAYS OPTIONAL

mnemonic — a symbol which has been previously defined via a CodeMacro directive, either by the assembler or by the user. This field is optional; however, if omitted, no operands may be present, although the other fields may appear. The set of all instruction mnemonics recognized by the assembler at any one time constitutes the assembler's "instruction set".

operand(s) — an instruction mnemonic may require other symbols to follow it to be the object of the actions called for by the machine instruction the mnemonic represents. These symbols are called "operands". Instructions provided as part of ASM86 require zero, one, or two operands. Users may define other instructions via CodeMacros that require more than two operands. All operands after the first must be preceded by a comma (,).

;comment — any semicolon (;) appearing outside of a character string begins a comment, which is ended by a line terminator. Comments are used to document and enhance the readability of programs. The liberal usage of comments is strongly urged.
THIS FIELD IS ALWAYS OPTIONAL

Examples:

```
LAB1:  MOV   AX,BX
        .
        .
LAB2:  MOV   CX, ALF[SI]
```

The DIRECTIVE statement format is:

```
name    directive    operand(s)    comment
```

where the fields are defined by:

name — the name field of a directive MUST NOT BE CONFUSED with the label field of a directive. This name is NEVER terminated with a ":". Some directives require that a name be present (viz., SEGMENT, ENDS, GROUP, RECORD, LABEL, EQU, PROC, ENDP); others PROHIBIT the use of name (PURGE, NAME, ASSUME, ORG, PUBLIC, EXTRN, END). Storage initialization directives (DB, DW, DD) allow names to be optionally present. The Codemacro directive is an exception to the above description of the directive statement and is partially described below.

directive — one of 20 keywords defined by the assembler to perform various "assembly-time" functions to assist the programmer in allocating storage, communicating between modules, and manipulating symbols.

operand(s) — analogous to the operands to an instruction mnemonic. Some directives allow an arbitrary list of operands (e.g., DB, DW, DD, PUBLIC, EXTRN, PURGE). Others allow special keywords designed to impart specific attributes to the entity being defined (e.g., SEGMENT, PROC). These operands may or may not be required, depending upon the directive.

;comment — exactly as defined for instruction statements.

Example:

```
RRR1  EQU  AX
```

CODEMACRO is an exception to the above formulae. The proper form for a CODEMACRO definition is:

```
CODEMACRO    name    operand(s)    ;comment
```

Notice that the name appears AFTER the CODEMACRO keyword.

Some directives require certain other directives to be present. These matched pairs are:

```
SEGMENT/ENDS
PROC/ENDP
CODEMACRO/ENDM
```

For both the SEGMENT/ENDS and PROC/ENDP pairs, the name that appears on the first directive must also appear on the second. For example:

```
Seg_1 SEGMENT
```

requires a matching

```
Seg_1 ENDS
```

ENDM directives may contain (but do not require) the name from the matching CODEMACRO.

MODULES

A module is the unit of assembly, i.e., when you assemble your source file, your object file defines a module. A program may span several modules (when you employ the technique of "modular programming"). These modules usually contain distinct logical functions which are combined using the relocation and linkage programs of the MCS-86 software family. (See the *MCS86 Software Utilities* manual.) Only one module is produced per assembly.

CHAPTER 3
VARIABLES AND INITIALIZATION

In designing a program or system, you must lay out the flow of control, i.e., the sequence of steps that the computer will follow in processing incoming commands or data. The goal is to accomplish this definition in a fashion that is convenient for several people: the designer, the programmer, and the person who may need to modify or add to the code in the future.

In the very nature of programming (at least for machines with some limit on memory), there are places and processes and data which are referred to multiple times. Rather than using numeric addresses, it is most convenient to establish names to stand for the addresses of these items:

a. LABELs, for references to chosen instructions (code), discussed in Chapter 4.

b. VARIABLES, for references to data.

c. NUMBERS, for references to immediate values.

d. EQUATED EXPRESSIONS, for references to indexed quantities, or more complex expressions, discussed in Chapters 4 and 5.

Throughout this manual, the word label almost always means code, i.e. relative to the CS (code segment) register only. The only exception occurs with a directive (in Chapter 4) named LABEL. VARIABLE always means data, not restricted as to segment register, although DS is the normal register used.

These two types of names are distinguishable in several ways: labels must have a distance attribute (see also Chapters 2 and 4) of NEAR or FAR, and can be defined using a colon after the name. Variables have a type attribute e.g., BYTE, never a distance attribute such as NEAR or FAR. Variables cannot be defined using a colon.

The convenience of names extends also to larger blocks of code in at least five ways:

1. in naming code-sequences as PROCEDUREs, defined once and then called into execution from many different locations
2. in naming code-sequences as CODE MACROs, defined in terms of a few varying parameters and reproduced in-line at each use of the name
3. in combining blocks of code
4. in checking for consistency in the use of names and registers, and
5. in localizing errors or omissions while testing the code for correctness.

SEGMENT-names are used to achieve the last three block-naming features.

Thus this chapter and related sections elsewhere in this book will discuss

1. how you establish such names
2. what default values automatically result as the meaning of such names in subsequent expressions
3. what options you have for altering or adding to these automatic consequences for names
4. how names are used.

Labels are discussed in Chapters 2, 4, and 5.

Code Macros are discussed in Chapter 7.

Procedures are discussed in Chapter 4.

Segments are discussed in Chapter 4.

Data naming, storage allocation, and initialization are discussed below.

Variable Declaration and Initialization

Since data can be defined in terms of bits, bytes, words, double-words, or other groupings, it is necessary to tell the assembler how much storage is required.

There are 3 kinds of storage allocation:

1. bytes — defined using DB
2. words — defined using DW
3. double-words — defined using DD

When you set up memory for data usage, you must specify its initial content. If you don't know or don't care what the initial value of the location is, then the question mark (?) should be used to indicate that no initialization is wanted (more about the question mark later).

The DB, DW, and DD Directives

The DB, DW, and DD directives serve two purposes: (1) to initialize memory and (2) to define variables. The acronyms stand for "define byte", "define word", and "define double-word".

Variable Definition

A variable can be defined with a DB, DW, or DD directive. The desired variable name appears to the left of the directive. As was mentioned in Chapter 2, variables have three address components: SEGMENT, OFFSET, and TYPE. The directive gives the type to the variable which appears to its left (i.e., BYTE for DB, WORD for DW, and DWORD for DD). The assembly time offset of the variable is equal to the number of bytes seen so far in the segment. The segment is the current segment.

In the case of a variable naming an array, the type is the number of bytes in a single element of the variable.

Example:

```
TABLE_DATA    SEGMENT
      TABLE   DW      12
              DW      34
       NUM1   DB      5
  TABLE_TWO   DW      67
              DW      89
              DW      1011
       NUM2   DB      12
      RATES   DW      1314
OTHER_RATES   DD      1718
 TABLE_DATA   ENDS
```

The segment attribute for all the above variables is TABLE_DATA. DW defines a word variable, 2 bytes, and DB defines a byte variable. The offset for TABLE_TWO is 5, the number of bytes between the beginning of the segment and that variable. The offset for RATES is 12.

The type of variables NUM1 and NUM2 is 1, meaning byte. The type of OTHER_RATES is 4, meaning doubleword. The type of all variables shown here in TABLE_DATA is 2, meaning word.

IDENTIFIER	SEGMENT	ATTRIBUTES	
		OFFSET	TYPE
TABLE	TABLE_DATA	0	2
NUM1	TABLE_DATA	4	1
TABLE_TWO	TABLE_DATA	5	2
NUM2	TABLE_DATA	11	1
RATES	TABLE_DATA	12	2
OTHER_RATES	TABLE_DATA	14	4

Memory Initialization

The DB, DW, and DD directives also serve to initialize storage. As seen above, an expression may appear to the right of the directive. This has the effect of initializing one "unit" of storage to the value of the expression. A "unit" of storage is BYTE for DB, WORD for DW, and DWORD for DD, that is 1 byte, 2 bytes, and 4 bytes respectively.

Expressions

Expressions may be used to initialize storage. Expressions are discussed in Chapter 5. It will suffice here to say that there are two kinds of expressions, numeric and address expressions. 5 is a numeric expression as is 4 * 50. A variable is an address expression as is a label. When an address expression is used to initialize storage, it may only appear in a DW or DD directive, never in a DB. "DW variable" will initialize a word of memory with the offset of the variable from its segment. "DD variable" will initialize two words of memory with the segment and offset of the variable.

Example:

```
FOO SEGMENT AT 55H

ZERO  DB  0             ; ONE BYTE OF 0
ONE   DW  ONE           ; ONE WORD OF 1  (0001H)
TWO   DD  TWO           ; LOW WORD OF 3  (0003H), HIGH WORD OF 55H (0055H)
FOUR  DW  FOUR + 5      ; ONE WORD OF 12 (000CH)
SIX   DW  ZERO - TWO    ; ONE WORD OF -3 (0FFFDH)
ATE   DB  5 * 6         ; ONE BYTE OF 30

FOO ENDS
```

The DD labelled TWO is very informative. Recall that the DD directive allocates 4 bytes. The first two were used as a word and filled with the offset of the variable TWO (offset = 3). The second word was filled with the number 55H. This is the paragraph number that the segment FOO is to begin on. Thus the two words in the DD represent the absolute address of the symbol TWO, which is the primary reason for the DD directive.

No Initialization; the question mark:

A single question mark is a keyword in the assembly language (see Chapter 2 about keywords). It may only be used in storage initializations, and means that it doesn't matter how the assembler initialized this location. What actually gets put into a location initialized by the question mark is indeterminate.

Example:

```
DB  ?
DW  ?
DD  ?
```

For these examples, 1, 2, and 4 bytes are reserved, but remain uninitialized.

The DUP Facility

The DUP facility allows for storage to be initialized by specifying an initial value (or set of values) and the number of times these values should be repeated. This allows for large storage areas to be initializedwith a small command. The form of the DUP is

```
expression  DUP  (item)
```

where expression is a numeric expression evaluating to an absolute number greater than zero. Item may be an expression (address or numeric), question mark, list of items, or more DUP repetitions. Item must be enclosed in parentheses.

```
      DB  100 dup  (0)          ; 100 bytes of 0
      DW  10 dup (?)            ; 10 words of unknown value
FOO   DD  50 DUP (FOO)          ; 50 copies of the absolute address of
                                ;   FOO (i.e., offset AND segment)
      DB  10 DUP (10 DUP (0))   ; 10 repetitions of 10 repetitions of 0
      DW  35 DUP (FOO, 0, 1)    ; 35 repetitions of three words; the
                                ;  offset of FOO, 0, and 1.
```

Lists

As seen in the above example, a parenthesized list of items may be DUPed in storage initialization. The list (FOO, 0, 1) stands for a single entity to be repeated. Anything that can appear by itself can also appear as a list member. This includes lists and strings (explained below). Thus we have the following examples:

```
DB  5  DUP     (1, 2, 4 DUP (3), 2 DUP (1, 0))
```

This DB directive initializes 50 bytes, 5 copies of the bytes:

```
          1, 2, 3, 3, 3, 3, 1, 0, 1, 0
ALPHA  DW  2  DUP  (3  DUP (1, 2 dup (4, 8), 6), 0)
```

This DW directive initializes 38 words. Two copies of the words with values:

```
1, 4, 8, 4, 8, 6, 1, 4, 8, 4, 8, 6, 1, 4, 8, 4, 8, 6, 0
```

A single list, not using the DUP facility, may not use parentheses. The following lists achieve the same storage definition and initialization as the line of values above:

```
DW  1, 4, 8, 4, 8, 6
DW  1, 4, 8, 4, 8, 6
DW  1, 4, 8, 4, 8, 6, 0
```

Character Strings

A byte can also hold the ASCII representation of a character such as 'A', '9'. 'ABCDE' is a string of 5 characters run together ("concatenated").

If you wish to initialize storage with characters, you simply enclose them in single-quote marks. Strings for storage allocation/initialization which are longer than 2 characters are legal ONLY in the DB command, and illegal in the DW and DD commands.

For example, consider the sequence

```
PART1   DB   'THANKS'
PART2   DB   'LOT'
LINE1   DB   'THANKS A LOT'
BUFFER  DB   128 DUP(' ')                  ; INITIALIZE 128 BYTES TO BLANKS
LINES   DB   80 DUP (72 DUP ' ', 0DH, 0AH) ; INITIALIZE 80 LINES, WHERE EACH
                                           ; LINE HAS 72 BLANKS AND ENDS
                                           ; IN A CARRIAGE RETURN, LINE
                                           ; FEED.
```

Recall that each character takes a whole byte of memory. The above DB commands have thus implicitly reserved and initialized multiple bytes: PART1 got 6 bytes, PART2 got 3 bytes, and LINE1 got 12 bytes. DB is the only command in the language that can accept strings longer than 2 bytes. If you were later to type

```
MOV  PART1,' '
MOV  PART1+1,'B'
```

then the 6-byte string beginning at PART1 would say
' BANKS'

The memory location PART1, being only 1 byte, contains simply the blank you moved in, but it also serves as the beginning of this longer string if you were to need to refer to it in later instructions. You would get the same result by typing

```
MOV  WORD  PTR  PART1,'B '      ; 'B ' because the bytes will
                                ; be reversed in memory the
                                ; PTR operator is discussed
                                ; in Chapter 5
```

Examples:

```
INVENTORY_ACCESS        SEGMENT
     FILTER_1     EQU   4          ; FILTER_1 is now a
                                   ; name for the absolute number 4.

     SWITCH_1     DB    ?
     LEVELS_1     DB    4 DUP (?)
     ACCESS_1     DW    FILTER_1
     STORES_1     DW    FILTER_1 DUP (0,1)

     SWITCH_2     DB    ?
     LEVELS_2     DB    0,1,2,3
     ACCESS_2     DW    FILTER_1 + 2
     STORES_2     DW    (FILTER_1 + 2) DUP (1,2)
INVENTORY_ACCESS        ENDS
```

The first set of 4 commands, after the EQU above, allocates and initializes 23 bytes of storage. SWITCH_1 allocates one byte, not initialized. LEVELS_1 allocates 4 bytes, not initialized. ACCESS_1 allocates one word, initialized to the set value of FILTER_1, namely 4. STORES_1 allocates 8 words based on using the value of FILTER_1 as a DUP-control expression. These 8 words are initialized to 0, 1, 0, 1, 0, 1, 0, 1, respectively.

The second set of commands allocates and initializes 31 bytes of storage. SWITCH_2 allocates one uninitialized byte. LEVELS_2 initializes its byte to 0 and the successive 3 bytes to 1,2,3 respectively. ACCESS_2 allocates 1 word, initialized to 2 more than the value of FILTER_1, i.e., 4 + 2 or 6. STORES_2 allocates 12 words or 24 bytes based on the value of this expression used as a DUP-control. The word at STORES_2 is initialized to 1, and the succeeding 11 words are initialized to 2,1,2,1,2,1,2,1,2,1,2 respectively.

VARIABLE	TYPE	LENGTH	SIZE
SWITCH_1	1	1	1
LEVELS_1	1	4	4
ACCESS_1	2	1	2
STORES_1	2	8	16
SWITCH_2	1	1	1
LEVELS_2	1	4	4
ACCESS_2	2	1	2
STORES_2	2	12	24

Words and Double-Words

DW creates 16-bit values, and DD creates 32-bit values.

```
EARLY       EQU     3
MIDDLE      EQU     1041                ; = 0411H
FINAL       EQU     28672               ; = 7000H
BLAST       EQU EARLY * MIDDLE
HEAT1   DW  EARLY * MIDDLE              ; = 0C33H
HEAT2   DW  (FINAL + BLAST) * EARLY     ; = 95385 = 17499H (ERROR)
HEAT3   DW  (FINAL + BLAST)*EARLY/4     ; =95385 =17499H (ERROR)
HEAT4   DW  (EARLY/4)*FINAL + BLAST     ; NOT AN ERROR
```

The value computed to initialize HEAT1 will fit in a word (16-bits), being less than 2 to the 16th minus 1 = 1111 1111 1111 1111 B = 0FFFFH. HEAT2 and HEAT3 are errors because they exceed that value during evaluation. They will remain undefined. (If such a value resulted from a computation during execution, it would be automatically truncated from its computed value of 17499H to 16-bits, becoming the number 7499H.)

IT IS IMPORTANT TO NOTE that the Intel convention for storing words places the least-significant byte in the lower-numbered memory location and the most-significant byte in the next-higher memory location.

Similarly for double-words, the least-significant word is placed in the lower-numbered memory location, and the most-significant word goes into the next-higher word of memory.

Thus the initial value for HEAT1 will be stored as 33H 0CH, and HEAT2 will be completely undefined.

Furthermore, memory is typically presented left to right as increasing location addresses, OR top to bottom for increasing location addresses. Thus

```
DW    1234H
DW    5678H
```

becomes in memory: 34H 12H 78H 56H, or in vertical representation of memory bytes,

```
34H
12H
78H
56H
```

Thus the above statement: the initial value for HEAT1 will be 33H 0CH, stored as:

```
33H
0CH
```

Double-Words

In the case of DD, you are allocating 2 words in 1 command. One purpose for this could be reserving room for later storing (during execution) both the segment and offset values of a label or variable which is not in this segment.

Two words contain 4 bytes, and since each byte can store a 2-digit hexadecimal number, a double-word could hold an 8-digit hex number. However, the largest constant allowed by this assembler is a 17-bit (sign + 16) number, which is truncated into 1 word. This is consistent with the 8086 architecture and machine instructions, which permit manipulation of values no larger than 16 bits (except for multiplication and division—see Chapter 5). The largest negative number possible is 0FFFFH.

Thus if you write:

```
DD      1234H
```

the high order bits of the DD are assumed to be 0000H. The convention above is followed, first by placing the least-significant half of the double-word in the lower-addressed word, and then followed again by placing the least-significant byte of each word in the lower-addressed byte of each word. Thus when you write:

```
DD      1234H
```

it is assumed you mean 0000H 1234H, which is the same as

```
DW      1234H
DW      0000H
```

since both become become stored as

```
34H
12H
00H
00H
```

If you need to store 2 non-zero words, e.g., a constant or a segment/offset pair such as 8765H 9423H, then 2 DW commands are needed:

```
DW      8765H
DW      9423H
```

which is then stored as 65H 87H 23H 94H, or

```
65H
87H
23H
94H
```

in order of increasing addresses. (However, a name can be used, as shown early in this chapter, under initializing with expressions.)

NOTE

The DD command is often used to create space for addresses, or rather that pair of words called the offset and the segment, representing an 8086 address (see Chapter 1 on addressing). By 8086 convention, a pair of words representing an address always has the offset first and the segment last (e.g., as used by the LDS command, or PUSH of a "long" value). By convention, then, 8765H in the pair of words above would be considered the offset value of such an operation, and 9423H the segment value.

This reversal of bytes in memory is usually only important when reading dumps of memory, e.g., in debugging. In most other cases, such as MOVing such quantities, the hardware automatically compensates appropriately for this convention and you need not pay any attention to it. For example, the sequence:

```
LOC1:  MOV AX, 'NO'
LOC2:  MOV MEMWORD, AX
LOC3:  MOV BX, MEMWORD
LOC4:  MOV MORWORDS, BX
```

operates as follows:

At LOC1, AH is filled with 4E representing N, and AL gets 4F, representing O.

```
4E        4F
AH        AL
```

At LOC2, the low-byte of MEMWORD is filled with 4F, high-byte gets 4E:

```
4FH
4EH
```

or

```
4F        4E
lo        hi
MEMWORD
```

At LOC3, BH gets 4E and BL gets 4F.

```
4E        4F
BH        BL
```

At LOC4, the low-byte of MORWORDS gets 4F and its high byte gets 4E:

```
4FH
4EH

4F        4E
lo        hi
MORWORDS
```

There is one kind of situation where it is important to remain aware of the memory reversal: when you sometimes treat the 2 bytes in a word as individual bytes. For example, the sequence:

```
VECTORB  LABEL  BYTE      ; assigns a 2nd name to next
                          ; location, but with type "byte" rather than type
                          ; "word", permitting access by byte—(see Chapter 4)
VECTOR     DW    1234H
             o
             o
             o
   MOV  AL,VECTORB
```

will put 34H into AL, not 12H.

DW and DD Character String

2-byte strings (but none longer) can be used with DW or DD, but the convention of reversed order must be remembered. An example:

```
SIGNAL1      DW      'GO'
SIGNAL2      DW      'NO'
```

are each interpreted by the assembler as a 2-byte number, namely the ASCII value for the characters. ASCII for G is 47H, for O is 4FH, N is 4EH. Thus the DW commands above are equivalent to:

```
SIGNAL1      DW      474FH
SIGNAL2      DW      4E4FH
```

These will be stored with the least significant byte first in memory, at the lower-addressed location: 4FH 47H 4FH 4EH, or 'OGON'.

When used in initializing word or double-word variables, one-character strings follow the convention of being filled out by zeros, since they fill only 1 byte of the 2 byte (DW) or 4 byte (DD) field. For example:

```
SIGNAL3      DW      'K'
SIGNAL4      DD      'P'
```

are interpreted as being filled out with numeric zeros as above; the following pairs of commands are totally equivalent to the pair above:

```
SIGNAL3      DW      4BH
SIGNAL4      DD      50H

             OR

SIGNAL3      DW      004BH
SIGNAL4      DD      0050H
```

The convention is then followed, storing the least significant byte in the lower-addressed location of each name, so that 4BH becomes 4BH 00H and 50H becomes, for a double-word quantity, 50H 00H 00H 00H.

Some Attribute Operators (Length, Size, Type)

Recall the definition of LINE1:

```
LINE1  DB  'THANKS A LOT'
```

If you were constructing messages later on in your program, it could be important to know the length of the string pointed at by LINE1. The operator LENGTH provides this function, e.g. the instruction

```
MOV  AX, LENGTH LINE1
```

would move 12, or 000CH, into the accumulator. (In many earlier assemblers, the capability achieved in this one line would have required the use of 2 labels and a subtraction.)

Similarly, after defining variables using DB, DW, and DD as in the examples above, you will of course use them in instructions. In many cases it will be necessary to know how they were originally defined, in terms of the unit of definition and how many were defined there, in order to point to the correct locations for picking up data or transferring control. Lists, loops, and arrays of various kinds will require precise pointers to achieve your intentions.

Naturally, you could keep track of whether a name represented a byte or word quantity by using a master list, or later in the project by using the listings to look up the required information. However, the assembler tracks this automatically for you.

It uses the implicit type of your variables to select the correct machine instruction to generate, and if you code instructions inconsistent with the data definitions used, the assembler will flag this as an error. It also provides special operators for use in expressions which need type information.

These operators are especially useful when you are creating (or calling) generalized procedures which are designed to provide the same process for whatever parameters are sent.

It is also a better programming practice to use a name or an expression rather than an explicit number in many contexts. A name is both easier to understand when reading the listing, and easier to modify later if the need arises, since its single definition (or change) then applies to every usage in the program.

The assembler operators SIZE, LENGTH, and TYPE provide the above capabilities. TYPE tells how many bytes are in the basic unit defined, i.e., TYPE LINE1 is 1 because the basic unit is a byte. TYPE SIGNAL3 is 2 because the basic unit is a word, or 2 bytes. TYPE SIGNAL4 is 4, for the double-word unit.

SIZE tells how many bytes are defined by the entire line where the name is declared, whereas LENGTH tells how many of the basic units were used. In the following declaration,

```
PATH1      DW      1234H,5678H,0ABCDH
```

3 words are initialized, the first pointed to by the name PATH1.

TYPE PATH1 is 2, meaning the basic unit is a word. LENGTH PATH1 is 3, because 3 units were allocated and initialized. SIZE PATH1 is 6, since 6 bytes are required to store 3 words. For LINE1, defined in bytes, LENGTH LINE1 = SIZE LINE1.

The general formula is:

```
SIZE name = LENGTH name * TYPE name.
```

One use of SIZE is to check that an index or subscript does not exceed the extent of the table or array it is used with.

One use of TYPE is to increment or decrement a loop counter or index by the correct number of bytes to point to the next item in a list. In a list defined as bytes, i.e.,

```
LIST_EXMPL   DB   500 DUP (13,21,34)
```

the correct increment is 1. For words, it would be 2, for double-words, 4.

The DUP features enables one directive to declare and initialize multiple units of a given storage type, e.g., LIST_EXMPL above. Three initial values are given, to be duplicated 500 times, for a total of 1500 bytes. The name of the first byte is LIST_EXMPL .

If you think of the 1500 bytes as a list or an array, then it makes sense to think of accessing list elements using an index or subscript. SI (or DI) is the usual index used following the variable name, e.g.,

```
LIST_EXMPL[SI]
```

When SI is 0, the first byte is addressed.
When SI is 5, the sixth byte is addressed.

The additional examples below may help clarify the operators used above, and also further show the use of the DUP feature.

Examples:

1.
```
ZERO_ARRAY   DW   1000   Dup (0)
```

   This initializes a block of 1000 words to 0, or 2000 zeroes.

```
TYPE   ZERO_ARRAY = WORD = 2.
LENGTH   ZERO_ARRAY = 1000.
SIZE   ZERO_ARRAY = 2000.
```

2.
```
BUFFER   DB   256   DUP (' ')
```

   This causes BUFFER to be an array of 256 bytes, each containing a blank.

3.
```
FIB   DW   1,1, 100 DUP (?)
```

   This initializes "FIB" to be a word array with initial values 1, 1, ?, ?, ... with LENGTH FIB = 102 and SIZE FIB = 204. As bytes, the values are 1,0,1,0,?,?,?,?,?,?,...

4.
```
ALT DB 50 DUP (0,1)
```

   This initializes "ALT" to 50 repetitions of 0, 1; i.e. 0, 1, 0, 1, ... LENGTH ALT = 100.

5. The size operator is very useful for strings:

```
s1  DB  size s1 - 1, 'this string has 29 characters'
s2  DB  size (s2) - 1, '12345678901234567890123456789O', '32'
```

using the size operator allows for automatic initialization of the byte before the string to the number of bytes in the string. Size s1 is 30, size s1 - 1 is 29. Size s2 is 33, size s2 - 1 is 32.

Note: Recall that "LENGTH NAME" is the length of the block in terms of its constituent units. This is most useful for controlling loops, etc. It is sometimes useful, however, to know the length of a block in terms of some standard units, usually bytes. This is provided by the Size attribute.

Record Definition

Records may be used only in codemacros which are described in Chapter 7. They are described here because they act to allocate storage in the codemacro. A record is a map or template you define. You may then easily allocate and initialize storage later in this format. The template itself has no storage allocated to it. When you use its name as the operation field of an instruction, you cause storage to be allocated there, at that time, according to the definition in the template. This may include defaults to initialize each field individually when that field is not given a value in the actual invocation of the record.

The format of the RECORD directive is:

```
name    RECORD    field_1,field_2,...
```

where the fieldnames have the form

```
fieldname: length_expression [ = other_expression ]
```

Such a declaration defines "name" to be a RECORD, packed into a byte or word depending on the number of bits in the whole definition, i.e., the number of fields and their length.

Each field is defined by coding its name, a colon, and an expression giving its length in bits. The only optional parameter is the "= other_expression", which may be specified after the field-length, to provide default initialization values. If the initialization value provided is too large, an error is reported.

The maximum number of bits in a RECORD is 16, the minimum, 1 bit. The operator WIDTH of a record gives its width in bits, i.e., the sum of the values of each field-length expression. The SIZE of the RECORD is defined as the number of bytes needed to hold it, as follows:

```
SIZE EXAMPLE_REC =

    1 if WIDTH EXAMPLE_REC is from 1 to 8 bits
    2 if WIDTH EXAMPLE_REC is from 9 to 16 bits
```

Once defined as above, the record's "name" can be used for allocating and initializing storage. The manner of doing this is similar to using DB, DW, or DD, with a few extra options and consequences, as explored below.

Each "fieldname" can be used in expressions or instructions as the shift count needed to right-justify the field. "MASK fieldname" is defined as that mask necessary to access the field in its original position. (The dot usage of record-

fieldnames is not discussed here. This construction, NAME.RECFIELD, is allowable only in CodeMacros and is discussed under that heading in Chapter 7.) The examples may serve to make this clearer.

Examples:

```
HASH_ENT    RECORD    FREE:1, EMPTY:1, INDEX:14
```

The above RECORD declaration for HASH_ENT will result in creating symbol values as follows:

```
FREE  = 15      MASK FREE  = 8000H
EMPTY = 14      MASK EMPTY = 4000H
INDEX =  0      MASK INDEX = 3FFFH
```

The values on the left are the shift counts you could use to right-justify those fields if you were using them in subsequent instructions. The values on the right are the masks needed to extract or test those fields directly from the record at VAR_ONE using a logical AND or TEST instruction. Notice that WIDTH HASH_ENT = 16, SIZE HASH_ENT = 2.

To store the EMPTY field of VAR_ONE in the empty field of VAR_TWO, you could use the following instruction sequence:

```
MOV AX, VAR_ONE                 ; moves word at VAR_ONE into
                                ; accumulator
AND AX, MASK EMPTY              ; ANDs to accumulator
                                ; 0100000000000000
MOV BX, VAR_TWO
AND BX, NOT MASK EMPTY          ; cleans VAR_TWO's EMPTY field
 OR AX, BX                      ; preserves the new EMPTY field
                                ; and other prior contents
MOV VAR_TWO, AX                 ; moves acc. contents into word
                                ; at VAR_TWO
```

The field named EMPTY can be tested by using:

```
TEST VAR_ONE, MASK EMPTY
```

This sets the zero-flag (ZF) to according to that field of VAR_ONE, i.e., ZF becomes 1 if the field is zero.

The operators SIZE, WIDTH, MASK, plus the automatic definition of the fieldnames as the shift counts, provide powerful capabilities that may not be immediately evident. They permit record manipulation in loops without explicitly coding a variety of specific numbers, such as the number of bytes holding the record, or the position or width of individual fields, or of the entire record.

Instead of numbers, you code the names or "operator name", as in FREE or MASK EMPTY or SIZE HASH_ENT. This saves figuring out such sizes or shifts or masks for every field and record you use. It also creates code sequences which can apply to records of widely varying content and structure, since the sizes, shifts, or masks do not appear as fixed numbers but rather as operations which depend on the record definitions.

This makes it much easier to construct generalized sequences or procedures which can be used without modification in greatly dissimilar applications, i.e., those which process greatly dissimilar data in functionally similar ways.

CHAPTER 4
ASSEMBLER DIRECTIVES

This chapter describes the assembler directives used to control the 8086 assembler in its generation of object code.

Generally, directives have the same format as instructions. Assembler directives are grouped as follows:

- LOCATION COUNTER AND SEGMENTATION CONTROL
  SEGMENT/ENDS
  ORG
  GROUP
  ASSUME
  PROC/ENDP
  LABEL
- SYMBOL DEFINITION
  EQU
  PURGE
- PROGRAM LINKAGE
  NAME
  PUBLIC
  EXTRN
  END
- MEMORY RESERVATION AND DATA DEFINITION
  DB (discussed in Chapter 3)
  DW (discussed in Chapter 3)
  DD (discussed in Chapter 3)
  RECORD (discussed in Chapter 3)

Segment Definition: The Segment and Ends Directives

Every instruction and every variable is contained in a block of locations called a segment. You create a segment and a segment-name with the segment directive, i.e.,

```
name1 SEGMENT [align-type] [combine-type] [ 'classname' ]
```

After such a SEGMENT directive, all instructions or data (except embedded segments) are considered to be in the 'name1' segment, until a directive of the form

```
name1    ENDS
```

is encountered. This ends the definition of segment "name1" for the moment. The same name field is required on both directives. The parameters after SEGMENT must be in the order shown. (Embedded segments are entirely separate, that is, their instructions or data are not considered to be in the outer segment at all, but only in the local embedded segment. Examples below.)

As shown above, there are 3 optional parameters in this directive, specifying attributes of the segment:

1. an alignment type, called align-type above, with 5 choices;
2. a combinability type, called combine-type above, with 5 choices; and

3. a classname of up to 40 characters which can be an arbitrary name you choose, enclosed in single quotes. Segments with identical classnames will be located together in memory unless more stringent controls are specified to LOC86 (or QRL86).

If more than one parameter is specified, they must be in the above order.

Alignment Choices: PARA, BYTE, WORD, PAGE, INPAGE

Introductory Considerations

The choice of an alignment is essentially a directive to the locating facility. The assembler must use this directive to create the information needed later by the linking and locating facilities to align your segments in the manner you specify.

The 5 alignment types allow you to state the boundary where you want this segment located. This boundary, like all addresses, has 2 parts—a paragraph number and an offset. The paragraph number becomes the value of the segment name. (See Chapter 1 on Addressing.)

It is useful to think of every address as such a pair of numbers, i.e., 1234H, 0056H means the address 12396H, formed by first shifting the paragraph number left 4 bits or 1 hex digit (1234H becomes 12340H), and then adding the offset of 0056H:

```
12340H + 0056H = 12396H.
```

The displacement of a segment boundary above the paragraph number requires some explanation. To the assembler, a segment name automatically means the paragraph number where the defined segment begins, i.e., there is no offset or the offset is zero. In this manual, a segment name can sometimes also mean the extent of or the contents of the defined segment, going forward into higher memory addresses for any number of bytes from 0 up to a maximum of 64K-1 bytes. For example, we may speak of moving a segment into a segment register, meaning the paragraph number where the segment begins, or we may ask whether some variable is in the segment, meaning defined as part of that segment's contents.

In your assembly language source program, all addresses in a segment are relative to the segment's beginning, i.e., a paragraph number with no offset. Each segment's beginning operates as relative-zero, so that subsequent definitions of labels or variables within the segment occur at relative locations 0, 1, 2, ...

In your source code you must MOV the segment's beginning into a segment register, using its name as in:

```
MOV     AX, SEGNAM44
MOV     ES, AX
```

The 8086 hardware can then automatically form the correct complete addresses for the variables or labels you have used in your program code. It combines the paragraph number from the segment register with the offset of the variable or label, using the built-in shift-and-add described in Chapter 1 on Addressing.

However, as LOCATE is putting your modules into memory, substituting absolute addresses for relative addresses, segments of varying length will end at varying locations.

If the next segment to be LOCATEd is placed on a paragraph boundary, i.e., with no displacement, then all the relative addresses that were defined in the source code for that segment are correct absolute offsets from that paragraph. There may be a few bytes unused between the end of the last segment and the beginning of this one, a loss of from 0 to 15 bytes since paragraphs occur every 16 bytes. This situation is the normal default specification that you get if you say nothing, or which you may code explicitly by writing PARA as your align-type on the SEGMENT directive.

The assembler makes it possible for you to instruct LOCATE to pack your code more tightly into the memory space. Part of the job of the LOCATE program is to fit logical segments into memory in the most space-efficient manner consistent with your specified instructions to it.

The actual beginning of a segment could be made to be the very next byte after the end of the previously located segment. (This will be the case if your align-type is BYTE.) If this is done, the segment has one chance in 16 of falling exactly on a paragraph boundary, and generally will not.

THIS IS WHERE THE DISPLACEMENT OF A SEGMENT COMES IN. The segment's paragraph number will be the nearest paragraph boundary at or below its exact location. Since the beginning is now NOT on a paragraph boundary, that beginning place has a displacement (offset). This offset is the distance in bytes from the selected (nearest-lower) paragraph number. Since paragraphs come every 16 bytes, this offset will be a number between 0 and 15, inclusive.

The LOCATE program establishes the paragraph number and displacement for the segment's beginning. It then adds the segment's displacement to the offset of every location referenced within that segment. This achieves the desired result, that all addresses used within the segment are displacements from that segment's paragraph number. The value of the segment's name is always that paragraph number.

Specific Align-Types

The default alignment type is PARA. The segment will begin at a location whose address is divisible by 16 decimal, i.e., whose value in hexadecimal has a last digit of zero. This implies an automatic, initial offset of zero for segname.

PAGE means the boundary address ends in hex 00, e.g., 76500H. This implies that the paragraph number itself ends in zero and that the offset, as above for PARA, is zero.

BYTE would mean any offset is acceptable. WORD implies an even offset (lowest bit of paragraph number =0) so that the full beginning address of the segment falls on an even address. Accessing words on even boundaries requires only 1 memory cycle, whereas if the boundary is odd, 2 cycles are needed to access word quantities. If your word-variables within this segment are defined on even boundaries, i.e., hex addresses ending in 0, 2, 4, 6, 8, A, C, or E, then every access to such a variable will take only 1 cycle.

INPAGE means that this entire segment must be located between one page boundary and the next, e.g., between 56700H and 56800H. It must not be allowed to overlap a page boundary. Thus its size cannot exceed 256 bytes. This specification is usually relevant only to converting some types of programs designed for earlier INTEL microcomputers, using the 8080/8085 assembly language.

If you do specify an align-type, it should appear on the first definition of the segment. You can omit it on subsequent SEGMENT directives for this segment, but you may not contradict that first specification.

If you omit it on the first SEGMENT directive for this segname, you automatically invoke the default, PARA. Later SEGMENT directives for this segname may specify this default, but naming any other explicit align-type will cause an error.

Combinability Types: (NONE),PUBLIC, COMMON, AT expression, STACK, MEMORY

The combine-type parameter on the SEGMENT directive gives information about how this segment may be combined with others when linked and located into absolute addresses.

If no combine-type is given, then no combining is performed, and the segment is local to this module or program only. This is the default, and it has certain code optimization advantages as described in the section titled "Models of Computation", Chapter 8.

If you do specify a combine-type, it should appear on the first definition of the segment. You can omit it on subsequent SEGMENT directives for this segment, but you may not contradict that first specification. If you omit it on the first SEGMENT directive for this segname, you automatically invoke the default. Later SEGMENT directives for this segname may specify this default, but naming any other explicit combine-type will cause an error.

There are 5 choices for combine-type, described in the paragraphs below.

PUBLIC

If PUBLIC is specified, then this segment will be concatenated with others of the same name encountered during linkage with other modules, i.e. all such segments will ultimately be contiguous. Their order is not affected by this directive but rather by a Relocation and Linkage (R&L)-command or the R&L default, which uses their order in the files being linked.

COMMON

Specifying COMMON causes this segment to share the identical memory locations with all other segments of the same name from other modules. This means that different labels or variable names (from different modules) could be applied to the same address.

For example, say in one module of your program (MODULE1) you declare a segment:

```
GLOBAL_DATA     SEGMENT     COMMON

PARAM1    DB    34H
ASSOC1    DB    82H
PARAM2    DB    61H
ASSOC2    DB    75H

GLOBAL_DATA     ENDS
```

Then in another module, (MODULE2) assembled separately, you define GLOBAL_DATA as:

```
GLOBAL_DATA     SEGMENT     COMMON

ITEM1     DW    ?
ITEM2     DW    ?

GLOBAL_DATA     ENDS
```

Then exactly 4 bytes of memory will be allocated to GLOBAL_DATA, but how they are accessed will depend on which name is referred to. Each set of names is known only within its own module unless PUBLIC and EXTRN directives are used (explained later in this Chapter).

A MOV of PARAM1 or ASSOC2 in MODULE1 will simply get the appropriate byte, as you would expect, namely 34H or 75H respectively. However, a reference to ITEM2 in MODULE2 will pick up the full word 7561H (and a reference to WORD PTR ASSOC1 will pick up 6182H, as discussed in Chapters 3 and 5).

AT Expression

This phrase specifies that the segment is to be located at the paragraph number computed by the assembler as the value of the expression. The pair of numbers expressing the absolute address of the segment will be <paragraph number,0> i.e., no offset from that paragraph boundary. For example, if you wrote AT 1234H, the segment would be located at absolute address 12340H. If you needed it to begin at absolute 12345H, the first line after this segment directive must be ORG 5. (See ORG in next section.)

STACK

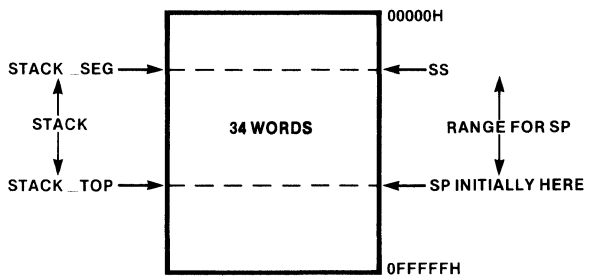
This combine-type is related to the 8080 STKLN directive. It will cause combining of the segment with others of the same name in other modules by overlay rather than concatenation. This means instead of one starting where the last one stopped, all begin at the same base address. Stack segments are overlayed against high memory. For example, suppose in one module you define:

```
STACK_SEG   SEGMENT       STACK
            DW 20 DUP (?)
STACK_TOP   LABEL WORD
STACK_SEG   ENDS
```

and in another you have:

```
STACK_SEG   SEGMENT       STACK
            DW  14 DUP (?)
TOP_STACK   LABEL WORD
STACK_SEG   ENDS
```

overlaying against high memory means that the locate program will combine these two pieces of STACK_SEG as follows:



Note that the length of the combined segment is 34 words, the sum of the lengths of the pieces; but that the top-of-stack variables "stack_top" and "top_stack", whose names are local to each module, are assigned the same offset (68D = 44H bytes) at the high end of memory. You can set the stack segment base and the stack pointer by the instructions (in module 1):

```
MOV     AX, STACK_SEG
MOV     SS, AX
MOV     SP, STACK_TOP
```

STACK is a special sort of segment, intended for use as temporary storage and retrieval using PUSH, POP, CALL, and RETurn, for passing parameters and return addresses of procedures and the like, working in a last-in-first-out (LIFO) fashion.

During execution, stacks grow downward as they increase, from higher memory addresses to lower addresses. Their usage is a bit like a stack of dishes: PUSH SI puts the word in SI on top of the stack, and POP DI pulls it off the top and puts it into DI. (See also Appendix D)

The storage reserved for combined stack segments is the sum of the individual segments, since the expected usage from each source could occur after the other had filled its stack and not yet emptied it.

MEMORY

MEMORY works similarly to COMMON, but the segment is located above all other segments in memory.

There should be only one segment with this combine-type per group of modules being linked together, since only the first one encountered by R&L will be given the MEMORY interpretation. Thus if MEMSEG is the first segment with the MEMORY combine-type encountered by R&L, it and any segment named MEMSEG in other modules will be located above all others in high memory. Other segments with MEMORY as their combine-type will be treated as if COMMON had been specified instead, and will not be located above all others. A warning will be issued by the LOCATE program.

NOTE

Although STACK and MEMORY are used as keywords above, they are still available as user-variable-names. Programs written in earlier Intel assembly languages used STACK to mean the first address of the stack, and MEMORY to mean the first byte of memory past the end of the program. Earlier assemblers passed along such keyword usage to the LOCATE program which filled in the final addresses. In ASM86, these keywords cannot be used for this identical function without additional coding.

Embedded Segments

If you use a segment name "name1" again after closing that segment with an ENDS directive, the code and data you write then will automatically follow the lines written earlier in segment "name1". One sequence that causes this is

```
ROUTINE_FIRST       SEGMENT
                      o
                      o             ; code here is in segment
                      o             ; ROUTINE_FIRST
                A:    o
ROUTINE_FIRST       ENDS

        OTHER_ROUTINES      SEGMENT
                              o
                              o     ; code here is in segment
                              o     ; OTHER_ROUTINES
                              o
        OTHER_ROUTINES      ENDS

ROUTINE_FIRST       SEGMENT
                      o
                      o             ; code here is in segment
                      o             ; ROUTINE_FIRST and
                      o             ; follows label "A"
ROUTINE_FIRST       ENDS
```

Another way this implicit concatenation happens is with a new segment directive, a sequence such as:

```
PROCESS_1           SEGMENT
                      o
                      o             ; code here is in segment
                      o             ; PROCESS_1
                      o
        PROC1DATA           SEGMENT
                              o
                              o     ; code here is in segment
                              o     ; PROC1DATA
                              o
        PROC1DATA           ENDS
                      o
                      o             ; code here is in segment
                      o             ; process_1 and directly
                      o             ; follows the last line
                      o             ; of the PROCESS_1
                      o             ; block interrupted by
                      o             ; the definition of
                      o             ; PROC1DATA
PROCESS_1           ENDS
```

An embedded segment must end before the outer segment ends. Thus if the

```
PROC1DATA    ENDS
```

directive came AFTER the

```
PROCESS_1    ENDS
```

directive, this overlapping of segments would be flagged as an error.

Code outside such SEGMENT/ENDS pairs will automatically be put in the default segment named "??SEG" defined by the assembler. (This segment is paragraph-relocatable and PUBLIC.

When used in other assembler instructions, e.g.,

```
MOV  AX,  Segnam
MOV  ES,  AX
```

this name will automatically have the value of the paragraph number where the segment begins. If a segment of identical name appears in another module that is later linked together with your program, the segments will be combined.

ORG Directive

The assembler's location counters perform a function during assembly similar to that of the instruction pointer during execution, namely, to tell the assembler the next memory location available to be assigned to instruction or data.

The first occurrence of the directive:

```
name1    SEGMENT
```

defines the beginning of segment name1. A new location counter is established and set to zero. This location counter is normally incremented automatically by 1 for each byte assigned. An ENDS directive for this segment freezes this location counter until the segment is re-opened, if ever, for continued assembly by a later "name1 SEGMENT" directive. At that point this location counter again begins counting the bytes assigned, beginning at the number last attained.

The currently active location counter can be altered by the ORG (origin) directive.

Code generated outside of all user-defined segments is placed in a special assembler-defined segment called ??SEG, which is PUBLIC. ORG statements outside of user-defined segments act upon the offset within that segment.

The ORG directive sets the location counter for the current segment being defined to the value specified by the operand expression.

```
Opcode      Operand

ORG         expression
```

Note: This directive may NOT have a label, e.g., SWITCH: ORG 14 is INVALID.

The location counter is set to the value of the operand expression, which may not be negative. The operands may be absolute numbers or relocatable numbers in the current segment. Assembly-time evaluation of ORG expressions always yields a modulo 64K address i.e., in the range 0 through 65,535. Any symbol in the expression must be previously defined. The next instruction or data item is assembled at the specified address.

In most modules, an ORG directive is unnecessary. If no ORG directive is included before the first instruction or data byte in a segment, assembly begins at location zero relative to the beginning of the segment.

Your program can include any number of ORG directives. Multiple ORG's need not specify addresses in ascending sequence, but if you fail to do so, you run the risk of instructing the assembler to create a second block of code for the same addresses as some previously assembled portion of the program. When loaded, one of the blocks will have overwritten the other.

See also the discussion of the $ sign in Chapter 5.

Example:

Assume that the current value of the location counter is 0FH (decimal 15) when the following ORG directive is encountered:

```
ORG     0FFH            ; ORG assembler to location
                        ; 0FFH (decimal 255)
```

The next instruction or data byte is assembled at location 0FFH.

Group Definition

This directive informs the assembler that the segments named in the operand list are intended to lie within the same 64K of memory. The group is given the specified name, and this name can be used in the same fashion as a segment name. One advantage to using groups is tighter code generation, since jumps within the group, for example, require only 16 bits even across segment boundaries. The assembler cannot check that all the segments named will fit into 64K, since some may be external or combined with others, but it causes such a check to be made by the Relocation and Linkage (R&L) facility. If they don't fit, you get the error then. This directive does not influence where segments are loaded by LOC86 (or QRL86). The classname parameter in the SEGMENT directive does that.

The form of this directive is:

```
name    GROUP    segnam1,segnam2, ...
```

where segnam1 etc. can be either the name field of a SEGMENT command or the expression "SEG variable-name", or "SEG label-name" which returns the segment in which that name is defined. This is particularly useful for forward-referenced or external names.

Example:

```
CODE_SET GROUP I_O_ROUTINES, INIT_PROC, SEG FIRST_RECS

DATA_SET GROUP INVOICE_REC, ACCT_REC, GEN_LDGR_DATA
```

When you load a segment register with a group name and put that information in a nearby ASSUME statement, then symbols from all segments in that group can be addressed using that one segment register. The offset used will be the distance from the group base-address. That usually will not be the same offset as the symbol's original offset in its segment, due to (possibly) other segments in between. This means the address you use for debugging will be from the LOCATE output, rather than the assembly output.

The relevant ASSUME statement could be simply:

```
ASSUME CS:CODE_SET, DS:DATA_SET
```

The order of the segments in the group command is not necessarily the order in which those segments will eventually be located in memory. Furthermore, a given segment name can appear in more than one GROUP command. There is no automatic method that guarantees an ordering which will satisfy all constraints imposed by various ASSUME and GROUP commands. You must exercise some care if you choose to reference several segments as a named group, from a single register. Special care is required to make multiple directives consistent.

If what you are after is not so much that the segments be addressable from a single segment register, but rather that they simply be near each other in memory (contiguous), then the class-name feature of the SEGMENT command achieves this goal more easily. However, it causes no check that the segments in the class lie within a 64K region of memory.

A further caution on groups is that they may NOT be forward-referenced (see Chapter 8 for discussion of Forward-References).

Assume Directive

Form: ASSUME segreg:segnam [,segreg:segnam,...]
or
ASSUME NOTHING

where segreg means one of: DS, ES, CS, SS
segnam means any segment name,
any previously defined group name,
the expression SEG variable_name or SEG label_name,
or the keyword NOTHING

Example:

```
ASSUME DS:DATAWORDS_SEGMENT_NAME,
&      ES:STRING_SEGMENT_NAME,
&      CS:NOTHING, CS:CODE_SEG_NAME
```

Note: This directive may NOT have a label, e.g.,

```
CASE1: ASSUME CS:S4 is INVALID.
```

It is essential that the assembler be informed of the execution-time environment in which the generated instructions will run. This environment consists of the expected contents of all four segment registers. The ASSUME directive tells the assembler what addresses will be there. The assembler uses this information to check that the variables and labels you refer to are addressable via the segment registers and to generate segment prefix bytes for variables whenever this is necessary.

The assembler-generated instructions depend on these expectations. Every memory address is actually a pair of 16-bit numbers, the offset and the segment-base-address (see Chapter 1). Nearly every instruction that refers to memory uses the offset only, expecting the segment-base-address to come from a segment register. If the run-time contents of that register are not as ASSUMEd at assembly-time, the correct run-time memory address is unlikely to be computed using the specified offset with that register. (That is, examples can be constructed wherein the correct address will be used, but the vast majority of cases result in a wrong address.) Therefore, an ASSUME directive is required before the use of segment registers, and before each point in the program where a run-time change to a segment register will occur.

Any references to memory or stack will use the DS, ES, or SS registers. These registers must therefore appear in an ASSUME prior to the code that will access memory. Similarly, instruction labels (including procedure names) implicitly use the CS register, which must therefore appear in an ASSUME statement prior to any code that refers to labels. If a segment register will not be used in a module or a segment, then ASSUME NOTHING should be specified for that register.

This directive is especially critical in the case of CS, because the offsets of locations containing instructions are always relative to CS. Instruction offsets are contained in the Instruction Pointer (IP), which determines the next instruction to be executed. In

most cases, it will only be valid if CS contains the segment-base-address ASSUMEd during assembly for that instruction. (Exceptions include self-relative jumps and register-only instructions.) If CS contains a different value, unexpected results are very likely to occur, untraceable without great effort.

For this reason, the assembler keeps track of the CS: assumed value for each label and instruction. It prohibits intrasegment/intragroup (NEAR) transfers to labels with a CS: assumed value different from the transfer instruction's CS: assumed value.

However, this prohibition naturally does not apply if the transfer replaces the contents of CS, as in an intersegment ("long" or FAR) jump or call. All subsequent uses of CS will then of course refer to the new value only. A new ASSUME directive is required at the target label, to inform the assembler of the new values which will have been placed in the segment registers during execution up to that point.

By informing the assembler as to the run-time contents of the segment registers, the ASSUME directive also allows you largely to avoid coding explicit segment prefix bytes.

In the absence of an ASSUME directive, all memory references to data must explicitly name the segment register to be used as the base address for accessing that data. The assembler requires this information in order to generate any prefix bytes necessary for the machine instructions it creates from your source code. For example, if SOURCE and DEST are defined in segment GLOBAL and FILL in segment PARAMS, then to put DEST-SOURCE+1 into FILL, you write:

```
GLOBAL SEGMENT
    SOURCE      DW  ?
    DEST        DW  ?
GLOBAL ENDS

PARAMS SEGMENT
    FILL        DW  ?
PARAMS ENDS

CODE SEGMENT
    ASSUME CS:CODE

MOV     AX, PARAMS
MOV     DS, AX

MOV     AX, GLOBAL
MOV     ES, AX

        ;following code assumes values were assigned to source and dest

MOV     AX, ES:DEST
SUB     AX, ES:SOURCE
INC     AX
MOV     DS:FILL, AX
.
.
.
CODE ENDS
```

As you can see, using explicit segment prefixes, ES points to GLOBAL and DS points to PARAMS.

The overrides ES: and DS: are required for every reference to variables in those segments. Data in the stack or code segments would similarly require SS: or CS: prefixes in your source lines.

The ASSUME directive allows you to tell the assembler once, rather than on each instruction, which segment registers are to be used for such references to variables. Thus the following sequence is equivalent to the above:

```
ASSUME CS:CODE, DS:PARAMS, ES:GLOBAL
MOV    AX, PARAMS
MOV    DS, AX

MOV    AX, GLOBAL
MOV    ES, AX

MOV    AX, DEST
SUB    AX, SOURCE
INC    AX
MOV    FILL, AX
```

As mentioned above, an ASSUME is needed in front of each block of instructions where a segment- register is changed, since the flow of control during execution can be quite different from the instruction sequence seen by the assembler at assembly time in any one module.

This means you will not have to code an explicit segment-override byte for variables whose segment base-address has been ASSUMEd into any segment register. Using ASSUME correctly, you usually do not have to think about segment override bytes. The assembler does it for you.

If the base-address needed for a reference to some variable has NOT been ASSUMEd into some segment-register, and there is no explicit override, then the assembler flags an error. The ASSUMEd content of each segment register controls what is regarded as known and allowed versus unknown and an error.

Each reference to a data item (variable) or label (code instruction) is checked against its initial-declaration-segment and the segments currently ASSUMEd in segment-registers. If the segment containing the referent is ASSUMEd into any segment-register, the assembler will generate the correct instruction to access that referent, including a segment prefix-byte if necessary. The assembler's choice of segment registers depends on the address-expression.

The address-expression implies a segment register by its use of variables and subscripts, i.e., pointer and/or index registers. This correspondence is discussed in Chapters 1, 5, and 7 where overrides and the MODRM byte are discussed.

If a variable appears in the address-expression, then there must be a segment register containing that variable's segment-base-address. This is the register used as a prefix if one is needed. If only registers are used in the address-expression, then the MODRM table tells which segment register is implied (DS for most subscripts, SS for those involving BP).

If that segment name, i.e., base-address, is ASSUMEd in a different segment-register from the default normally used by the relevant hardware instruction, then the assembler knows a segment-override prefix-byte is necessary to correctly access the given variable. If the needed base address is NOT assumed to be in any of the segment-registers, i.e., has not been named in an ASSUME directive, an error is reported (unless you coded an explicit prefix-byte as discussed below).

One way to understand this checking runs as follows:

a) which segment contains this referent?

b) has that segment (or a group containing it) been ASSUMEd into a segment-register, i.e. into SS, CS, DS, ES ?

c) if not ASSUMEd, report an error unless the data reference has an explicitly-coded segment prefix-byte.

d) if so ASSUMEd, is that segment-register the one normally used by the relevant hardware instruction?

e) if the ASSUMEd register is the normal default, generate the normal code.

f) if not, is the default overridable or absolutely required?

g) if overridable, generate a prefix-byte first, so as to use the correct segment-register, and then generate the normal code.

h) otherwise, report an error.

ASSUME NOTHING

The ASSUME NOTHING form of this directive removes all former assumptions as to which segment-base-addresses were in which segment-registers. This turns off the implicit generation of segment-overrides. The net result is to require you to code an explicit segment prefix-byte for every operand.

If you do not provide the prefix-byte, the assembler will give an error since it is unable to verify that the variable is addressable from any segment-register. If the assembler were to generate an instruction which used the segment-registers inappropriately for your expected arrangement of data, you would get unintended, confusing, and usually bad results.

For example, most hardware instructions for references to variables impicitly expect to use the (contents of the) DS register as a base:

```
MOV   AX,DATAWORD

MOV   CX,ARRAY[SI]

MOV   DX,MATRIX[BX+7][SI]
```

The 8086 hardware would expect these all to use DS unless you code a segment prefix-byte. The assembler cannot safely expect all your data to be currently addressable even from the DS register. In the absence of an explicit prefix-byte, the assembler must check and supply it from the ASSUME. In the absence of an ASSUME, and in the case of ASSUME NOTHING, you must code even the DS as an explicit segment prefix-byte.

Therefore, under the ASSUME NOTHING condition, the correct code for the instructions above is:

```
ASSUME NOTHING

MOV   AX,DS:DATAWORD

MOV   CX,DS:ARRAY[SI]

MOV   DX, DS: MATRIX[BX+7][SI]
```

Naturally, you would have earlier set BX and SI to the correct values such that they pointed to the desired elements in the data segment when used here.

Another example: if you know that DATAWORD is in a segment whose base address is in SS, and the others are in a segment whose base address is in ES, then an ASSUME directive could inform the assembler appropriately. Otherwise, in the ASSUME NOTHING case, the instructions should appear as follows:

```
MOV   AX,SS:DATAWORD

MOV   CX,ES:ARRAY[SI]

MOV   DX,ES:MATRIX[BX+7] [SI]
```

The string instructions provide another instructive illustration. In the case of MOVS (see Chapter 6), the source operand is normally in the DS segment, and the destination operand must be in the ES segment, i.e., the ES cannot be overridden. Thus the instruction sequence

```
ASSUME DS:SOURCE_STRING_SEGMENT, ES:DEST_STRING_SEGMENT

MOV DI, DEST_STRING_INDEX
MOV SI, SOURCE_STRING_INDEX
STD     ; this sets the direction-flag
MOVS   DEST_STRING, SOURCE_STRING
```

moves the byte (or word) pointed at by SOURCE_STRING indexed by SI, in DS, to the byte (or word) pointed at by DEST_STRING indexed by DI, in ES, and then increments DI and SI by 1 or 2.

(The indexing is implicit in MOVS. SI and DI are not explicitly named. In fact, MOVS operates solely by using SI and DI, and doesn't need even the string names, but the assembler requires them in order to check type and addressability via the ASSUME.)

If the source string were actually in the stack segment, then a segment-override prefix-byte would be necessary, giving the instruction:

```
MOVS      DEST_STRING, SS:SOURCE_STRING
```

In the ASSUME NOTHING case, omitting that SS prefix-byte will cause an error message from the assembler.

Since the ES default cannot be changed or overridden, only an ES: segment-override is ever appropriate on the DEST_STRING referent. It is in fact required if the variable's segment is not ASSUMEd in ES.

If SS were the correct segment containing the source-string and the prior ASSUME had said:

```
ASSUME SS:SOURCE_STRING_SEG
```

then the SS segment-override would have been generated automatically by the assembler.

Explicit Segment Prefix-Bytes

Only 1 segment prefix-byte is permitted in an instruction.

If you code an explicit segment prefix-byte, e.g.

```
MOV      AL, ES:DATABYTE
```

then the assembler does what you tell it to, rather than checking if it makes sense. This means it generates your specified instruction including the override byte you explicitly coded, rather than checking whether the data referent is truly accessible from that segment register.

Example:

The hardware expects BP to point within the stack segment, i.e., expects to use the contents of BP as an offset to the contents of SS in order to form an address. It is permissible to use BP as a pointer into the data segment instead, but an override byte (DS:) is required, to change which segment register is used to form the address.

The following code would cause the assembler to generate the needed override:

```
ASSUME DS:SEG ARRAY
   o        ; seg is described in Chapter 5.
   o
MOV  AX, ARRAY[BP]
   o
   o
```

but you might want the code to clearly reflect this usage upon even a casual reading, by coding the override explicitly on all such instructions:

```
MOV     AX, DS: ARRAY [BP]
```

Label Definition

Labels in the most general sense are names for any specific location in memory, i.e., for locations that contain instructions or that contain data. Throughout this manual a distinction is maintained between "variables", meaning data, and "labels", meaning instructions. NEAR and FAR are the two types of labels; the other choices below are variable types.

The LABEL directive can create a name for any location, regardless of its contents or intended use, and assign a type to that name. The type determines the legitimate uses of the name:

If the type is NEAR or FAR, then the name is a label per this manual's use of the term. It can be used in jumps or calls but not in MOVs or other data manipulating instructions. It may not be subscripted.

If the type is BYTE, WORD, DWORD, or some OTHER_VAR, then the name is a variable. It is valid in MOVs etc., but never directly in jumps or calls. (An indirect jump or call can use a variable (of type 2 or 4). See Chapter 5.)

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 "type" has 5 choices: BYTE, WORD, DWORD, NEAR, FAR

As discussed in Chapter 3, variables (i.e., data names) are created when a name is placed on a storage allocation/initialization. For example,

```
PUFF  LABEL  BYTE
      DB  21
```

is equivalent to

```
PUFF  DB  21
```

Either choice names the current assembly address as PUFF, types it as a byte-variable, and initializes it to 21.

Naming instructions on the same line requires a colon, e.g.

```
TRANS:    MOV    AX,CX
```

which is equivalent to

```
TRANS  LABEL  NEAR
       MOV  AX,CX
```

A label definition (i.e., for instruction code only, not variables) is allowed only when the segment currently being assembled is ASSUMEd to be within reach of the CS register. This means you must provide an ASSUME CS:NAME statement, where "NAME" is either the name of this segment itself or the name of a group which contains the segment. If a group name is ASSUMEd, the label's offset in all references will be from the base of the group, and the label's paragraph number will be that of the group. To write "name:" is the same as "name LABEL NEAR", due to the colon. (See also the NOTE below re: NEAR.)

However, this colon construction is not legal on the same line as a storage initialization, i.e., ITEM: DW 0 is illegal. Without the colon this would simply define ITEM as a word variable initialized to 0. If you need ITEM to be a label and not a variable, you may put the definition on the prior line by itself, e.g.,

```
ITEM:
     DW  0
```

or

```
ITEM  LABEL  NEAR
      DW     0
```

The value of "name" on the LABEL directive will include the current segment and offset, i.e., the pair of numbers defining this specific address. It will also carry the attribute specified by "type", here NEAR.

This directive is usually used to attach a second name to a location so that it can be referenced in different ways without using the PTR operator. For example, if you needed to treat the same area of memory as both a byte and word array, the definition of the array could appear as

```
VECTORB    LABEL    BYTE
VECTOR     DW       1000 DUP  (0)
```

Future references to it as a byte array would say, e.g.,

```
ADD AL, VECTORB   and for word usage would say
ADD AX, VECTOR
```

This could also be achieved using the PTR operator if it were not too frequent to be taxing, writing:

```
ADD AL, BYTE  PTR  VECTOR     ; see Ch.5 re PTR
```

which alters the type attribute for this reference only.

An example pertinent to code as opposed to data: you might wish to change the distance attribute for a label. Take the case of a label referenced extensively within this segment, but also referenced from outside this segment (and outside the group, if any). The fact that any references from outside occur forces you to declare the FAR attribute. You can, however, define a synonym within this segment with the distance attribute NEAR, thereby saving a word for each local reference since only the offset value would be needed, and not the segment value. The sequence would be

```
PROCESS_ITEMS     LABEL      FAR
  LOCAL_NAME:     MOV AX, FIRST_ITEM   ; any instruction
                  o
                  o
                  o
```

This objective could also be achieved using EQU, since at any point in an assembly, the following statements are identical:

```
name      LABEL     x
name      EQU       THIS x
```

The operator THIS transmits to "name" the current segment/offset pair plus the attribute specified by "x", which must be a "type" as above.

Thus in the example for code above you could have written:

```
PROCESS_ITEMS     EQU       THIS  FAR
  LOCAL_NAME:     MOV    AX, FIRST_ITEM   ; any instruction
```

The main utility of the operator THIS lies in expression arithmetic on the current location counter, e.g.:

```
STAR EQU (THIS FAR) + 1  ; see Ch.5 re THIS
```

NOTE

In assembling your code, the assembler matches the type (of the operands you supply) against the type specified in the codemacro definitions which make up the legal operations in this language. A NEAR label will match the specification Cw (meaning a 16-bit label expression) ONLY within code assembled under the SAME CS: assumption as when the label was defined. If this CS: assumption is not the same, the NEAR label will not match any codemacro specification. See Chapter 2.

Procedure Definition

A procedure is a section of ASM86 code which can also be then activated from other parts of the program as if it had been encountered sequentially. A CALL statement activates the procedure, causing the procedure code to be executed out of normal sequence. Program control is transferred from the point of activation to the beginning of the procedure code (or a label within it). The code is executed from that point, and upon encountering a RETurn instruction in exit from the procedure code,

program control is passed back to the next instruction beyond the point of activation. RET must be explicitly coded where desired. It is not an automataic function of the procedure's end, and there may be more than one point of return within a procedure.

The use of procedures has the following advantages:

1. It forms the basis of modular programming,
2. It facilitates making and using program libraries,
3. It eases programming and documentation, and
4. It reduces the amount of object code generated by a program.

The following paragraphs tell how to declare procedures, and how to activate procedures.

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

This pair of directives attaches a label to the entry point of a code sequence and declares whether the procedure is NEAR or FAR. If you omit the type, NEAR is used. The same name field is required on both directives of the pair, and there must be only matching pairs in the assembly.

Using the PROC declarative is almost the same as if LABEL were used instead. However, this pair is necessary for the use of CALL and RETURN. When you CALL a procedure, before control is transferred to it, the address of the next sequential instruction is stored on the stack. This enables the RETURN statement at the end of the procedure to return control to that next instruction after the call.

If the procedure is NEAR, the RETURN simply POPs the word at the top of the stack into the Instruction Pointer (IP). If the procedure's type is FAR, it POPs that first word into IP and then POPs the next word into the CS, restoring the original segment address.

RETurn can only work correctly if the return address stored by the CALL is at the top of the stack. If the stack is used by this procedure (or any procedure it calls) for temporary storage of parameters, such values must be POPped off the top of the stack before RETurn is executed.

If multiple entry points into a procedure are required, this can be accomplished by using LABEL directive. If the PROC is declared FAR, then multiple entry points can only be achieved via the LABEL directive. Alternate entry points to a procedure MUST have the same type (NEAR or FAR) or the returns will not work as needed. RETurn instructions appearing outside of PROC-ENDP pairs are assumed to be NEAR.

As with nearly all names in the symbol table, a procedure name can be purged. This removes the earlier definition of the name from the symbol table (only), and makes it possible to use the name for a new purpose.

NOTE

JMPing out of a procedure leaves the return address (1 or 2 words) on the stack. This must be taken into account if the stack is later used again.

RET need not be the physically last instruction in the source code of a PROC, but if execution of the procedure allows control to simply fall through to the ENDP, there is no automatic or implicit RET. A transfer to a RET is needed. There may be multiple RETurn statements in a procedure.

Procedures and segments may be nested within one another, meaning one procedure or segment may completely contain another. They may NOT overlap, i.e., the inner procedure or segment must end before the outer one ends. For example, this sequence is valid:

```
STARTER     SEGMENT
FIRST       PROC    NEAR
FIR_1:        o
              o
FIR_3:        o
            NEXT    PROC  NEAR
     NEX_1:             o
            MID     SEGMENT
                       o
                       o
            MID     ENDS
     NEX_44:   RET
            NEXT    ENDP
FIR_77:       o
              o
              o
            RET
FIRST       ENDP
STARTER     ENDS
```

The following sequence is invalid:

```
STARTER     SEGMENT
FIRST       PROC    NEAR
              o
              o
              o
            NEXT    PROC  NEAR
                       o
                       o
                       o
                    RET
FIRST       ENDP
STARTER     ENDS
              o
              o
              o
            RET
NEXT        ENDP
```

Procedures are executable where they appear as well as when they are CALLed. After a CALL FIRST instruction is executed, the instruction FIR_1 will be executed. If we assume no jumps or calls occur in FIRST, then FIR_3 will be executed, followed by NEX_1. If the instruction at NEX_44 were not a RETurn, then the

instruction next executed would be FIR_77. One reason for emphasizing this is that PLM86 works differently: procedures in that language are not executed unless CALLed, and embedded procedures are skipped over unless named explicitly in a CALL.

EQU Directive

The assembler automatically assigns values to symbols that appear as instruction labels or variable names. This value is the current setting of the location counter when the line is assembled. (See ORG.)

You may define other symbols and assign them values by using the EQU directive. Symbols defined using EQU cannot be redefined during assembly.

The name required in the label field of an EQU directive must not be terminated by a colon.

Symbols defined by EQU have meaning throughout the remainder of the program unless PURGEd.

EQU assigns the value of 'expression' to the name specified.

```
            Opcode    Operand

name        EQU       equ_op
```

EQU defines "name" as a synonym either for another name in the symbol table or for a constant value.

Equ_op may be a number, an expression, a register, or a macro name, e.g.:

```
• NUMBER              THREE   EQU  3
• ADDRESS EXPRESSION  XYZ     EQU  ALPHA  [SI]+3
• REGISTER            COUNT   EQU  CX
• MACRO NAME          RADD    EQU  ADDR
```

The required name field may not be terminated with a colon. This name cannot be redefined by a subsequent EQU or another directive (unless it is PURGEd first). The EQU expression cannot contain any external symbol. (External symbols are explained under EXTRN later in this chapter.)

Assembly-time evaluation of EQU expressions always generates a modulo 64K address, i.e., a value in the range −64KH to + 64K−1.

EXAMPLE: The following EQU directive enters the name ONES into the symbol table and assigns the binary value 11111111 to it:

```
ONES    EQU    0FFH
```

The value assigned by the EQU directive can be recalled in subsequent source lines by referring to its assigned name in subsequent expressions:

```
MOV     AL, 25 AND ONES
```

It is also possible to use substitute names for more complex expressions or address-expressions.

```
        A       EQU ARRAY [BX] [SI]
        B       EQU (ARM*7+44) AND MASK ONE
SUMMA           EQU ARRAY_SUMMER
```

After these EQU definitions, A and B may be used freely as synonyms for the expressions shown, possibly saving time and preventing coding errors in complicated usage. This feature can also improve readability of programs by creating names which have meaning to the application. For example, in a highway control project, instead of A, you might use the name TRAFFIC. In a defense command and control project, instead of B, you might code SAM_SELECT. SUMMA might be an acceptable synonym for a procedure that sums arrays.

PURGE Directive

The PURGE directive allows names to be deleted from the symbol table. Once a name is PURGEd it can be redefined and used in ways completely different from the earlier usage, without assembler conflict or confusion. All occurrences of a name after PURGing and redefinition will use the latest redefinition. If the name was PURGEd and not redefined, a later use will be flagged as an undefined symbol error.

```
PURGE name-1,name-2,...,name-n
```

This directive may not have a name, e.g., ALTER PURGE ADD is INVALID.

Program Linkage Directives

Modular programming and the relocation feature enable you to assemble and test a number of separate modules that are to be joined together and executed as a single program. Eventually, it becomes necessary for these separate modules to communicate information among themselves. Establishing such communication is the function of the program linkage directives.

A module may share its data addresses and instruction addresses with other modules. Only items having an entry in the symbol table can be shared with other modules; therefore, the item must be assigned a name or label when it is defined in the module. Segments with a combine-type of COMMON share the same locations. Other items to be made available to other modules must be declared in a PUBLIC directive.

Items needed from other modules must be declared in an EXTRN directive. Your module could directly access data or instructions defined in another module if it knew the actual address of the item, but this is unlikely when both modules use relocation. By using a name, you allow the assembler to arrange that the address be supplied by the Relocation and Linkage (R&L) programs. You thus gain access to data or instructions declared PUBLIC in other modules.

However, the assembler normally flags as an error any reference to a name or label that has not been defined in your program. To avoid this, you must provide the assembler with a list of items used in your module, but defined in some other module. The EXTRN directive does this.

Public Directive

The PUBLIC directive makes each of the symbols listed in the operand field available for access by other modules.

```
Opcode          Operand

PUBLIC          name-list
```

Note: This directive may NOT have name, e.g., PUB1 PUBLIC GETLIST is INVALID.

Example:

```
PUBLIC      SIN,COS,TAN,SQRT
```

Each item in the operand name-list must be a name assigned elsewhere in this program to a number, variable, or label (including PROCs). When multiple names appear in the list, they must be separated by commas. Each name may be declared PUBLIC only once in a program module. It may not be external too.

PUBLIC directives may appear anywhere within a program module.

When assembly is otherwise complete, if an item in the operand name-list has no corresponding entry in the symbol table, it is undefined and is flagged as an error.

EXTRN Directive

The EXTRN directive provides the assembler with a list of symbols referenced in this module but defined in another module. For these symbols, the assembler establishes a linkage to outside this module and does not flag the undefined references as errors.

```
Opcode      Operands

EXTRN       extref1,extref2,...
```

where each extref has the form

```
name: type
```

Each item in the list identifies a symbol that may be referenced in this module but is defined in another module. The type stated in EXTRN must be the same as that definition. When multiple items appear on the list, they must be separated by commas.

The "type" is required, and may be BYTE, WORD, DWORD, NEAR, FAR, or ABS. ABS means a pure number and not a variable or label.

Note: This directive may NOT have a name, e.g., MOJL1 EXTRN V:BYTE is INVALID.

Inside a user-declared segment, an external label or variable is assigned to that segment; outside of user segments (in the segment named ??SEG), it is assigned NOT to ??SEG, but to no segment. If you want to refer to its segment, you must say "SEG name" (see Chapter 5), or use an explicit override where you know which segment register will be correct at run-time.

If a symbol in the operand list is also defined in this module, the effect is the same as defining the same symbol more than once in a program. The assembler flags this error.

If a symbol is used without a definition in this module and without appearing in an EXTRN directive, then the assembler flags it as undefined.

Although EXTRN directives may appear anywhere within a program module, it is usually better to place them near the beginning, to avoid forward-reference problems. (See Chapter 8.)

A symbol may be declared external only once in a program module. There is no requirement that it be used in this module. However, it may not be declared both external and public.

Example:

```
EXTRN    ENTRY:BYTE,ADDRTN:FAR,BEGIN:WORD,NUMBER:ABS
```

NAME Directive

The NAME directive assigns a name to the object module generated by this assembly.

```
Opcode          Operands

NAME            module-name
```

The name directive requires the presence of a module-name in the operand field. This name must conform to the rules for defining symbols. It may be identical to symbols with other uses.

Module names are necessary so that you can refer to a module and specify the proper sequence of modules when a number of modules are to be bound together.

The NAME directive may appear in the program at most once. It may NOT have a name-field, e.g., MODNAM NAME RATES is INVALID.

If the NAME directive is coded erroneously or is missing from the program, the assembler supplies a default NAME directive. The module-name is taken from the root of the input file name: i.e., invoking the assembler by typing ASM86 MYFILE.MD1 will create an object-module, MYFILE, in a disk file named MYFILE.OBJ. This will cause an error if you later attempt to bind together several object program modules with this name. This could occur if different programs whose names differed only in extension were all assembled using the default name e.g.:

```
MYFILE.MD2
MYFILE.AB6
MYFILE.QRX
```

all become

```
MYFILE.OBJ
```

if the default is invoked each time.

Example:

```
NAME      MAIN
```

This name may be the same as symbols with other usages, e.g.:

```
NAME  AX
```

is valid.

Assembler Termination END Directive

The END directive identifies the end of the source program and terminates each pass of the assembler.

Opcode	Operand
END	optional-label

Only one END statement may appear in a source program, and it must be the last source statement. If it is not the last, all the remaining statements cannot contribute to the assembly, e.g., to match Segment or Procedure labels, or to provide definitions for forware references. This will cause many statements to appear as errors.

If the optional label is present, its value is used as the starting address for program execution. If no expression is given, there is no starting address, and this module is assumed not to be the main module.

Whenever a number of separate program modules are to be joined together, only one may specify a program starting address. The module with a starting address is the main module.

CHAPTER 5
EXPRESSIONS AND ADDRESS EXPRESSIONS

ASM86 allows many types of expressions. The rules for valid expressions and address-expressions are precise. You can use them to determine the validity of whatever expressions you may need.

This present section gives some general guidelines and examples from which you may get a feel for the patterns of valid expressions.

To begin, definitions are needed:

VARIABLE — means the name of a location whose contents are intended as data. Its definition will NOT use a colon, e.g.

```
SOUP  DW  2
SALAD  LABEL  BYTE
```

The relevant attributes of a variable are its segment, offset, and type. (See Chapters 2, and 3, and the review later in this chapter.)

LABEL — means the name of a location whose contents are intended as an instruction. Its definition will often use a colon, e.g.,

```
ADD_INGREDIENTS: MOV AX, SOUP
```

but not always, e.g.,

```
FOO PROC FAR
```

and

```
BAZ LABEL NEAR
```

are both labels. The four attributes of labels are segment, offset, CS:value-assumed, and distance (i.e., NEAR or FAR). See also Chapters 2, and 4, and the review later in this chapter.

NUMBER — means simply a name for a numeric value, not a location, e.g., 7. If a name has been given a strictly numeric value, e.g.,

```
DELTA EQU 77
```

or

```
B EQU 11
```

then DELTA and B are NUMBERs rather than variables or labels.

The distinction between variables and labels on one hand, and numbers on the other, is very important in this assembly language. Variables and labels fall into a class of expressions called ADDRESS EXPRESSIONS.

An ADDRESS EXPRESSION is an expression which evaluates to a memory address. Therefore, a variable is an address expression as is a label. An address expression has 3 components:

1. The SEGMENT part
2. The OFFSET part
3. The TYPE

These three components are a necessary part of every legal address expression.

The other class of expressions is NUMERIC EXPRESSIONS or just NUMBERS. Numeric expressions yield a result which is a number. 3 is a number as is 4 * 5 (i.e., 20). Each component of an address expression is a number, but an address expression is NOT a number.

To make clear the distinction between address expressions and numeric expressions consider the OFFSET operator (more completely defined later in this chapter): OFFSET of an address expression returns the offset component of the address expression as a number.

The following example program uses the OFFSET operator to illustrate the distinction between address expressions and numeric expressions:

```
        ASSUME  CS: code, DS: data

data    SEGMENT AT 55H

        DB  0, 1, 2                 ; 3 bytes
e_byte  DB  0FFH

data    ENDS

code    SEGMENT

start:  MOV  AX, data               ; segment base address to AX (i.e., 55H)
        MOV  DS, AX                 ; and then to the DS register.

one:    MOV  AL, 3                  ; the number 3

two:    MOV  AL, e_byte             ; address expression: e_byte

three:  MOV  AL, OFFSET e_byte      ; the number 3

code    ENDS
        END  start
```

The two expressions which are most often confused are shown in the instructions labeled "two:" and "three:",namely e_byte vs. OFFSET e_byte. E_byte is an address expression. Instruction "two:" will move 0FFH, the value in the memory location whose address is e_byte, into the AL register. OFFSET e_byte is a number. The MOV instruction labeled "three:" will move 3, the offset component of the address expression e_byte, into the AL register.

These two MOV instructions are very different; the first must fetch the source (rightmost) operand from the byte of memory 3 bytes from paragraph 55H. The source operand itself is 0FFH. For the MOV labelled "three:" the source operand is the immediate value, 3, which is part of the MOV instruction.

The instructions labeled "one:" and "three:" are identical.

From this example we see that using an address expression with an 8086 instruction means that the operand will come from a memory location. Using a number means that the number itself will be used as an immediate operand.

The distinction between address expressions and numbers is important in other places in the assembly language. The repetition count on a DUP construct (see Chapter 3) and the paragraph number for absolute segments (see Chapter 4) must be numbers, not address expressions.

Permissible Range of Values

The maximum range of values a number can have is −0FFFFH through 0FFFFH. All arithmetic operations are performed using signed two's complement arithmetic. Out of range values get an error message. The following list gives important characteristics of the arithmetic performed by ASM86.

1. NOT 0FFFFH = 0

2. AND, OR, and XOR do not yield results which are out of range.
   e.g., 0000FH AND 0FFF0H = 0. 0FFFFH XOR 0FFFFH =0.

3. All other operators give an overflow message if the result is out of the maximum range.

Since an address expression has 3 numeric components, it is illegal if any component goes out of range. An overflow message will be issued if this happens.

DB will accept values in the range −256 through 255. Any value between −256 and −129 will be stored as a positive value between 1 and 127. The mapping is −129 = 127, . . . −255 = 1, −256 = 0.

DW will accept the entire range of values. Any value between −32,769 and −65,535 will be stored as a positive value between 0 and 32,767. The mapping is −32,769 = 32,767, . . . −65,535 = 1.

Precedence of Operators

Expressions are evaluated left to right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. When two operators have equal precedence, the left-most is evaluated first.

Parentheses can be used to override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost are evaluated first. For example:

```
15/3 + 18/9 = 5 + 2 = 7

15/(3 + 18/9) = 15/(3 + 2) = 15/5 = 3
```

The following list describes the classes of operators in order of increasing precedence:

1. SHORT
2. Logical OR, XOR ; These 3 classes are bit-by-bit
3. Logical AND ; logical operators. Corresponding
4. Logical NOT ; bits in the operands are operated ; on to give corresponding bits in ; the result.
5. Relational Operators: EQ, LT, LE, GT, GE, NE
6. Addition/Subtraction: +, - (both unary and binary)
7. Multiplication/Division: *, /, MOD, SHL, SHR
8. HIGH, LOW
9. The operators to manipulate variables: "name:", PTR, OFFSET, SEG, TYPE, THIS
10. Parenthesized expressions, LENGTH, SIZE, WIDTH, and square brackets.

In (9) and (10) the operands may include BYTE, WORD, DWORD, NEAR, FAR, $, and address references of two types, i.e.

```
a symbolic name

name1 [subscript_expression]
```

Address-expressions may involve BX, BP, SI, or DI enclosed within square brackets. The subscript expressions must evaluate to pure numbers or expressions involving only these registers. Subscripts are discussed later in this chapter.

Alphabetic operators must be separated from their operands by at least one blank. This prevents their being seen as part of a symbolic name in your code.

General Introduction to Operator Classes

Certain high-precedence operators never operate on pure numbers. These 8 can only be used on or with address-expressions: segment-override ("name:") prefix, OFFSET, SEG, square brackets for subscript offsets, WIDTH, LENGTH, SIZE, and TYPE. WIDTH is only for records.

The multiplicative and logical operators can work only with pure numbers.

The remaining 4 operator classes are the relational, additive, high/low, and pointer (PTR). They can be used either with pure numbers, or with labels or variables from the same segment (only). Subexpressions, such as those within parentheses, eventually evaluate to one or the other. Then you can judge the validity of the remaining operations indicated by the full original expression.

Two other operations are defined only to allow use of the current value of this segment's program counter: "THIS", and "$". The operator "THIS" is always used with a "type", i.e., BYTE, WORD, DWORD, NEAR, or FAR. (There is a review of attributes later in this chapter.) If you write:

```
A    EQU    THIS    BYTE
```

you have defined A to be a VARIABLE of type BYTE. A's address is this current segment and this current offset within the segment, i.e., the program counter's present value at this line of code. An equivalent line is:

```
A    LABEL    BYTE
```

If you write:

```
B    EQU    THIS NEAR
```

you have defined B to be a LABEL of type NEAR, i.e., jumps or calls to B will be expected to require at most a 1 word displacement. B's address is again this segment and this current offset in it. An equivalent line is:

```
B    LABEL    NEAR
```

The usage of $ is explained later in this chapter.

Review of Attributes

A VARIABLE is the name of a memory location whose contents are intended for use as data. Variables have 3 attributes: segment, offset, and type. Segment is the name of the block of code in which the variable is defined. Offset is the number of bytes from the beginning of that segment to the line defining the variable. Type is the number of bytes in the basic unit used in that definition: 1 for byte, 2 for word, 4 for double-word. Chapter 3 describes variables in greater detail, particularly about types. Chapter 1 describes segment and offset more fully.

A LABEL is the name of a memory location intended for use as an instruction. Its first 3 attributes are segment, offset, and distance, where segment and offset mean the same as the above for variables. Its final attribute is the CS:ASSUME value in effect when the label was defined.

The distance attribute can be NEAR or FAR: NEAR means the label will be referred to only in segments assembled under the same ASSUME CS:name as the label. Thus all references will assemble into one-word displacements.

FAR means 2 words are needed to be able to access the label: the first is the offset of the label within its segment (where it was defined), and the second is the segment address itself, which must be (put) in CS for such access to occur. Such a replacement of the current contents of CS is handled automatically by a long jump or long call ("long" meaning intersegment, or FAR).

The CS:ASSUME attribute refers to the segment address (name) that you tell the assembler to expect at run-time. This attribute is described more fully in Chapter 4 under the ASSUME directive. It is used by the assembler to define what labels are accessible using the current CS: value (i.e., what labels are presently NEAR). Those which are not so accessible are FAR, and can only be reached by a long jump or call, which fills CS with the needed value.

Additive Operators, + and –

These operators perform 17-bit arithmetic (sign plus 16-bit) integer addition and subtraction between numbers, variables, and labels under certain rules:

1. Absolute numbers may always be added or subtracted from variables, or labels, or absolute or relocatable numbers. When a number is added to a variable (or label), the result is a variable (or label) whose offset is the sum of that number and the original offset of the operand variable (or label).

2. Variables and labels may be subtracted only if they are in the same segment.

3. Variables and labels may never be added.

4. One base register and one index register may be added, e.g., [BX + SI]. Absolute or relocatable numbers may be added or subtracted from such registers or expressions. Registers may not be subtracted from numbers.

Examples:

1. MOV AX, ARRAY_START + 6
   This moves the 4th word after ARRAY_START into AX.

2. TABLE2 DW NEW + 17 DUP (2); NEW must be absolute

Square Brackets and the Registers BX, BP, SI and DI

The registers BX, BP, SI, and DI may be used as general purpose registers or as indexing registers. When they are used as indexing registers, the value contained in the register is used as an offset from some segment register. Square brackets distinguish between the two possible usages of the indexing registers. If an indexing register appears in square brackets, then the contents of that register will be used to calculate the offset.

Example:

```
MOV  AX,  BX      ; move the contents of the BX register to AX

MOV  AX,  [BX]    ; This moves a word from memory into AX. The
                  ; word is in the data segment, with segment
                  ; address in DS. The offset from DS is the
                  ; contents of register BX.
```

The assembly language allows [BX] to appear alone as a legal address expression, even though only the offset is present, because there are defaults associated with such usage to make the expression legal:

REGISTER	SEGMENT REGISTER USED	TYPE
[BX]	DS	?
[BP]	SS	?
[SI]	DS	?
[DI]	DS	?

Recall that every address expression has three components: SEGMENT, OFFSET, and TYPE. The segment component of a bracketed indexing register is some segment register. The offset is the contents of the indexing register. The type is unknown. ASM86 uses the type of the other operand in the instruction to determine the type of the bracketed indexing register.

Square brackets have another use in the assembly language, which is described in the section USING SQUARE BRACKETS AS SUBSCRIPTS. The current discussion applies only to square-bracketed expressions appearing alone as address expressions.

Example:

```
MOV  AX,  [BX]    ; since AX is a word, [BX] is given type WORD.

MOV  CL,  [DI]    ; since CL is a byte, [DI] is given type BYTE.
```

Using the type of the other operand is only possible if

1. there is another operand, and

2. the type of the other operand is unambiguous.

Therefore, the following examples do not contain sufficient information:

```
INC  [BX]         ; increment a byte or word? (There is no carry from
                  ; 255 unless a word is being incremented)

MOV  [SI], 3      ; move the byte 3 (8 bits) into the byte at offset [SI]
                  ; or the word 3 (16 bits) into the word at offset[SI]?

JMP  [BP]         ; jump indirect INTER-segment or INTRA-segment
                  ; i.e., is CS to be replaced?
```

and the assembler will issue the error "INSUFFICIENT TYPE INFORMATION TO DETERMINE CORRECT INSTRUCTION." See Chapter 7 for a precise description of how the type is gleaned from the other operands.

In the explanation of the additive operators above, it was pointed out that one base register and one index register may be added. Furthermore, absolute or relocatable numbers may be added or subtracted from such registers or expressions. This may only happen inside square brackets. The following rules govern what is legal inside square brackets when the bracketed expression is to be used by itself as an address expression.

1. Numbers may appear only if a base or index register appears.

2. BX and BP may never appear in the same expression. The same is true for SI and DI.

3. BX or BP may appear alone, with numbers, and/or with SI or DI.

4. SI or DI may appear alone, with numbers, and/or with BX or BP.

5. Operations on numbers are unrestricted, but only additive operations can apply to the base or index registers in this context.

Thus the following expressions are all valid:

```
[BX + DI + (SIZE a) / 2]
[BP]
[BX + 7]
[7 + BP]
[SI -100H]        ; this is legal, but 100H - SI is not
[SI]
[DI + BP]
[BX + SI]
[SI + OFFSET block]
```

But the expressions below are all invalid:

```
[BX * 7]
[BX + BP]
[BP - SI]
[3 - BX]
[BX + DI * TYPE a]
```

The following rule determines the appropriate segment register:

If BP is in the expression Then SS is used, otherwise DS is used.

The TYPE of the bracketed expression is determined as described above when only a single register appeared in brackets.

Expressions in square brackets allow for more sophisticated indexing than if just a single base or index register appears in brackets. The address expression [BX + SI] means that the sum of the contents of BX and SI will be used as an offset from the DS register. [BP + 4] means that the 4 will be added to the contents of the BP register and the sum will be used as an offset from the SS register. [BX - 40H] will subtract 40H from the contents of the BX register. The result will be used as the offset from the DS register. Two sample programs follow.

Example:

Finding the maximum value in an array of at least one element. The program fragment on the following page finds the maximum of an array of words. It uses the first element of the array as a temporary maximum. It then compares each element to the current temporary maximum. If the element in the array is larger than the temporary maximum, it becomes the temporary maximum. When the program has looked at every element in the array, the temporary maximum will be the maximum. The temporary maximum is held in the AX register. The BX register is used to hold the offset of the next element in the array.

```
        ASSUME CS: code, DS: data

data    SEGMENT

values    DW  2100 DUP  (?)
count     DW  ?

data    ENDS

code      SEGMENT

start:    .                          ; initializes segment registers and
          .                          ; reads numbers into the values array. Count
          .                          ; will contain the number of values read.

          MOV   BX, OFFSET values    ; starting offset into BX
          MOV   CX, count            ; the number of values into CX
          MOV   AX, [BX]             ; move the first value into AX as a first guess.
          JMP   testlp

find_max:
          ADD   BX, 2                ; point BX at the next element of the array.
          CMP   AX, [BX]     ; Compare the current max with that next element.
          JG    testlp               ; if AX is still the max, then try next value;
          MOV   AX, [BX]             ; else AX gets maximum found so far
testlp:   LOOP  find_max             ; and we go to test the next value.

done:
          .
          .
          .
code    ENDS
        END   start
```

Example:

The following procedure will print a string on the CRT. It calls the procedure named CRT (defined external) with the character in AL. The string is expected to terminate with a null (0). SI is expected to contain the offset of the string from DS.

```
        ASSUME  CS: code
code    SEGMENT  PUBLIC

        EXTRN   crt: NEAR

print   PROC   NEAR

        MOV  AL, [SI]                ; move next character to print into AL
        CALL crt                     ; print the character
        CMP  BYTE PTR [SI + 1], 0    ; see if the next character is 0
        JE   done                    ; if it is, then go to exit
        INC  SI                      ; else point to next character
        JMP  print                   ; and go to beginning of the loop
done:   RET

print   ENDP

code    ENDS
```

Variable-Manipulation Operators:

"name:", PTR, THIS, SEG, TYPE, OFFSET

Two of these six operators can change an attribute of a variable or label, usually for the duration of one instruction only. These two operators are the segment prefix-byte ("name:"), and the PTR operator.

Segment-Prefix

Every instruction that alters the flow of control or which writes or reads memory (stack included) uses a segment register in computing the necessary memory address (see Chapter 1). This excludes register-only operations.

The assembler decides which segment register to use. For each instruction, this choice depends exclusively on the kind of address-expression you coded and the current values ASSUMEd in the segment registers. The assembler has a fixed algorithm (explained under ASSUME in Chapter 4) for analyzing that expression and deciding which segment register is correct.

If there is more than one correct choice, e.g., because the same segment-base-address is in more than one register, the assembler always chooses the shortest code, i.e., no prefix byte if possible.

You may override the assembler's choice. If you code an explicit override different from the assembler's choice, your choice is used.

A prefix byte is required (and supplied by the assembler) if the desired memory location is only accessible using a segment register different from the one used by hardware default. That is:

1. if the desired memory location is accessible using the default segment register for the specified expression, then no segment prefix byte is needed.
2. if it is not accessible using the current ASSUMEd contents of ANY segment register, the reference is an error.
3. if it is accessible through a segment register different from the default, a segment prefix byte is needed.

The rules for analyzing address expressions and hardware defaults are as follows:

1. Any address expression with type NEAR (i.e., labels) always use the CS register. No override is legal.
2. Any address expression with type FAR will always result in the CS and IP registers getting new values. No override is legal.
3. Instructions which use the SP register implicitly will always use the SS register. No override is possible. These instructions include PUSH, POP, CALL, RET, IRET.
4. If a string instruction uses the DI register to point to an operand, then the ES register will always be used with that operand. No override is legal (see the NOSEGFIX operator in Chapter 7).

ALL other cases involve data references to memory (i.e., address expressions with type BYTE, WORD, or DWORD). The default may be overridden in these cases.

5. If the address expression uses the BP register (i.e., within brackets), then the default segment register is SS. Otherwise it is the DS register.

The expression "segreg:address expression", where segreg is a segment register, creates a new address expression whose segment component is that segment register. If "segreg" is the same as the default segment register for "address expression", then no prefix byte is generated (since it would be redundant). If "segreg" is different from the default, then a prefix byte is generated.

Example:

```
        ASSUME CS: code, DS: data

data    SEGMENT
m_byte  DB  ?
data    ENDS

code    SEGMENT

 MOV   AL, [BP]          ; SS is the default. [BP] has a segment component
                         ; of SS. No segment prefix byte is generated.
 MOV   AL, DS: [BP]      ; SS is the default. DS is explicitly used. A prefix
                         ; byte is generated so that the DS register is used.
 MOV   AL, m_byte        ; BP is not used, DS is the default. M_byte has a
                         ; segment component of "data". "Data" is ASSUMEd
                         ; in the DS. No prefix is generated.
 MOV   AL, ES: m_byte    ; DS is the default. ES is explicitly used. A prefix
                         ; byte is generated so that the ES register is used.

 ASSUME DS: NOTHING, ES : data

 MOV   AL, m_byte        ; DS is the default. "Data" is not in DS, but is in ES.
                         ; A segment prefix byte for ES is generated.
 MOV   AL, DS: m_byte    ; DS is the default. Ds is explicitly used. No prefix
                         ; byte is generated.
code    ENDS
```

NOTE

The above example is meant to illustrate how the assembler decides when a segment prefix is necessary. It is NOT an example of the proper use of the ASSUME statement. Nor is it an example of when to use the segment override operator.

The following contexts are examples of when a segment register might be used to override an address expression.

- When using the BP register as an indexing register in a segment other than the stack segment. The address expression "[BP]" has a segment component of SS. If BP is used to index into some other segment, then a segment override must be used (e.g., DS: [BP] will index into the current data segment).
- The string instructions which use SI to point to an operand. It is not uncommon to have these operands in the current extra segment. This operand may be overridden with an ES: override if that is the case.
- If a program uses data in the extra segment, then any use of a bracketed register expression to reference the data will require a segment override. For example, in the instruction MOV AX, [SI], the segment component of "[SI]" is the DS register. "ES: [SI]" is necessary if SI is to work as an indexing register into the extra segment. The only exception to this need for an ES: override is the DI operand to string instructions, for which ES is always used.

If you find certain forward-references unavoidable, you may be using a variable whose entire segment is defined later. In such a case, you can put the name of the segment in the ASSUME statement and use it as an override in the relevant instructions:

```
ASSUME CS:CODE, DS:DATA, ES:LATER_SEG
                o
                o
MOV AX, LATER_SEG
MOV ES, AX
                o
                o
MOV AX, LATER_SEG:LATER_VARIABLE
                o
                o
MOV DX, ES:LATER_VARIABLE
                o
                o
```

Both references to LATER_VARIABLE are acceptable, though there is a slight advantage to using LATER_SEG as an override rather than ES in the above example. If you were later to change your choice of segment register, i.e., to use DS instead of ES, only one change (to the ASSUME) would handle all such references, whereas using ES: would require a line-by-line modification. LATER_SEG must eventually be seen as a segment-name...anything else is an error. The only time a segment-override affects more than one instruction is in an EQU statement. When you say:

```
NEW1    EQU    ES:ARRAY[SI]
```

then every later use of NEW1 includes the fact that ES is the segment register used.

The use of groups requires some additional discussion here. If you have not yet studied groups, or do not intend to use them, the next few paragraphs will be of academic interest only. You may prefer to skip to the next section, on the PTR operator.

A group allows one base-address to serve for the multiple segments in the group. The maximum size of a group is naturally 64K bytes, as large a distance from the base-address as can be expressed in a 16-bit offset. Thus the sum of the sizes of the individual segments making up the group may not exceed 64K.

The ultimate order of these segments within the group is not known during assembly. Thus the offset of a variable or address-expression from the group's base-address must be a relocatable quantity. The offset within the variable's segment must be added to the size of any segments in the group which may lie between the base-address and the variable. This is handled automatically by the LOC86 program using information provided by the assembler.

For your use of groups and overrides, you need to remember that variables or address-expressions in a group are of necessity relocatable entities. For example, if G is a group and S is one of the segments in G, and IDENT is a variable in S, then G:IDENT is a relocatable entity whose segment attribute is G, whose offset will be its displacement from the beginning of G, and whose type is the TYPE of IDENT.

The PTR Operator

The PTR operator creates a variable or label. The new variable has the same offset (and segment, if any) of the operand on the right side of PTR, plus the attribute on the left of PTR.

Examples:

After defining a 9 word array by coding:

```
WARRAY  DW  9  DUP  (0)
```

you may later wish to access the 18 bytes as bytes rather than as 9 words. Normally an instruction like

```
MOV AL, WARRAY [SI]
```

would be illegal due to type conflict: the type of AL is BYTE, i.e., 1, and the type of WARRAY is WORD, i.e., 2.

```
MOV AL, BYTE  PTR  WARRAY [SI]
```

is valid, because you have made it explicit that bytes are what you want despite the original definition of WARRAY as words.

That original definition stands unchanged by such an instruction . The attribute alteration applies only to that instruction containing it.

Similarly you can write:

```
JMP  FAR  PTR  LABEL77
```

If you know LABEL77 will later be defined in a different segment from this JMP, requiring 2 words of address in the JMP rather than the 1 word normally assumed.

If there will be more than a few instructions needing PTR to correctly access the same target, you can define a synonym once, using EQU. Thereafter it is simpler to use the synonym. For example:

```
BARRAY  EQU  BYTE  PTR  WARRAY
```

allows the same locations defined as words in WARRAY to be accessed as bytes when referred to as BARRAY, e.g., if SI=3 then:

```
MOV AH, BARRAY [SI]
```

puts the fourth byte of BARRAY into AH (4th because the first is BARRAY [0]).

If you use PTR with pure numbers, you create variables or labels with an offset equal to the number, with the type you specify, and with a segment attribute of 0. Such an expression is legal only when preceded by a segment prefix byte or by the operators TYPE or OFFSET.

Example:

DS:BYTE PTR 77 could be used to temporarily define a byte variable of offset 77 in the DS segment. Without an override, e.g., DS:, this expression would have no segment attribute and thus would be illegal.

MOV AL, DS: BYTE PTR 77 would put into AL the contents of the byte located 77 bytes past the beginning of the current data segment. This practice is deemed less wise and less convenient than naming the desired location and using the name for such a reference; but if the need arose, the expression is valid.

Another example, only slightly more reasonable, might be the case of your knowing from prior work that the offset of a desired word in segment ROUT2 was the same as the offset of the byte variable PATH1 in segment ROUT1. You could then write

```
ASSUME ES:ROUT2
MOV AX, ROUT2: WORD PTR OFFSET PATH1
```

Again, this is a valid construct whose convenience and clarity are questionable.

To round out the picture of PTR, it is also valid to write expressions of this sort:

```
NEAR PTR VARIABLE_NAME
```

or

```
BYTE PTR LABEL_NAME
```

The first enables a programmer to transfer execution control to an area originally defined as data. This practice is usually a mistake (see Note).

The second allows code to be dynamically accessed and tested. Further, it permits a programmer to vary the contents of instructions during execution. This too is usually a mistake (see Note).

NOTE

But it can be done intentionally when there is a need for it, and without disastrous results if extreme caution is observed.

PTR With Indexing Registers

A common use of the PTR operator is with bracketed indexing registers, as follows:

It is impossible to determine what the correct type of some bracketed expressions must be. Recall that the following three uses of such expressions are illegal:

```
INC  [SI]    ; increment a byte or word?
MOV  [DI], 3 ; move a byte with value 3 or a word with value 3?
JMP  [BX]    ; jump INTRA segment indirect or INTER segment indirect?
```

The PTR operator allows this situation to be remedied:

```
INC  BYTE  PTR  [SI]    ; increment the byte pointed to by SI
MOV  WORD  PTR  [DI], 3 ; move 16 bits worth of 3 to the word at DI
JMP  DWORD PTR  [BX]    ; perform an indirect INTER-segment jump.
```

The Operator "THIS"

As mentioned in the Introduction to this Section, the operator "THIS" creates a variable or label of the type you specify, whose offset and segment are the current values of assembly.

The definition of every variable or label includes a type-attribute, e.g.,

```
DATA_TABLES    SEGMENT    PARA    "DATACLASS"
          A                DB              100  DUP  (0)
          X                DW              300  DUP  (47)
          Y                DD              100  DUP  (13)
LOC1:
                           o
                           o
                           o
DATA_TABLE                 ENDS
```

implicitly types A as an array of byte variables, X as an array of word variables, Y as an array of double word variables, and LOC1 as a NEAR label.

"THIS" can be used, at the same place defining the original name, to define an alternate name of different type. As discussed above under PTR, there may be times you need to access locations as bytes when the original definition was words, or vice versa.

The following definitions illustrate such alternate naming:

```
DATA_TABLES    SEGMENT    PARA    'DATACLASS'
          WA               EQU             THIS WORD
          A                DB              100 DUP (0)

          XB               EQU             THIS BYTE
          B                DW              300 DUP (47)

          WY               EQU             THIS WORD
          YB               EQU             THIS BYTE
          Y                DD              100 DUP (13)
LOC1:
          o
          o
          oo
          oo
DATA_TABLES  ENDS
```

WA allows pairs of bytes in A to be accessed as words. The offset and segment attributes of A and WA are the same; they differ as shown in the table below.

Similarly, XB allows each byte of B to be individually accessed.

WY can be used to access each WORD in the Y array individually. YB allows every BYTE to be directly addressed. WY, YB, and Y have identical segment and offset attributes; they differ only in type, size, and length:

VARIABLE	SEGMENT	OFFSET	TYPE	LENGTH	SIZE
WA	DATA_TABLES	0	2	1	2
A	DATA_TABLES	0	1	100	100
XB	DATA_TABLES	100	1	1	1
B	DATA_TABLES	100	2	300	600
WY	DATA_TABLES	700	2	1	2
YB	DATA_TABLES	700	1	1	1
Y	DATA_TABLES	700	4	100	400
LOC1	DATA_TABLES	1100	NEAR	—	—

NOTE

"$" is an abbreviation for THIS NEAR; it was used in earlier INTEL assemblers in self-relative jumps, i.e.,

```
JMP  $ + 4
```

meant to transfer control from this point in the instruction sequence to the instruction byte 4 bytes further on. This type of instruction is less useful in the 8086. The generality of the assembler mnemonics allows different-length machine instructions to be assembled from similar-looking source code, depending on the attributes of the operands. Thus it is inconvenient and dangerous for you to supply the precise number of bytes, e.g., 4 in the above example. It is much easier to label the target location and put the label in the jump command, i.e.

```
    JMP  SHORT  @1          ;@ 1 is target label
          o
          o
          o
@1: MOV  AX, NEW_SEG_VAL
    MOV  ES, AX
      o
      o
```

SEG, TYPE, and OFFSET

These operators create numbers, by separating out 1 of the 3 attributes of variables or labels.

Since the variables defined above were all in segment DATA_TABLES, then SEG A = SEG X = SEG Y = DATA_TABLES. This operator can be used in the ASSUME statement or in building addresses, or in any expression you find useful.

TYPE of a variable gives the number of bytes in the unit of definition.

```
TYPE A = TYPE XB = TYPE YB = 1
TYPE B = TYPE WA = TYPE WY = 2
TYPE Y = 4
TYPE LOC1 = NEAR
```

In the last line, the type must be either NEAR or FAR because LOC1 is a label rather than a variable. NEAR is the default.

The offset of a variable or label (address-expression) means the distance in bytes from the value of the segment-name to the location of that variable or label. The actual value assigned to a segment-name is always a paragraph number, as discussed in Chapter 1, and in Chapter 4 under the Segment directive. However, the real beginning of that segment, i.e., the first byte assembled for it, can be assigned an address up to 15 bytes higher by the LOCATE program. The determining factor is the align-type on that Segment directive. This means the relative offset seen at assembly time may be changed by the time final addresses are assigned and execution begins. If the segment is PUBLIC, it is combined with other segments of the same name from different modules, possibly adding even more bytes to the offset.

Thus when you wish to refer to the offset of some variable or label, it is often incorrect to use the relative offset from the assembly — you must use the OFFSET operator instead. The assembler then passes along information enabling LOCATE to fill in the final correct offset without further attention on your part.

One typical example is using the offset as the first address of an array, moving the offset into a base register and accessing array members by an index register:

```
ASSUME DS:SEG  B
            o
            o
            o
            MOV     BX, OFFSET B
            MOV     SI, 0
MORE:       o
            o
            ADD AX, [BX + SI]   ; equivalent is [BX] [SI], see below.
            o
            o
            o
            JMP MORE
```

Although array B in DATA_TABLES has a relative offset of 100, the OFFSET operator assures that even if B's ultimate absolute offset is different from 100, access to the B array will be made correctly.

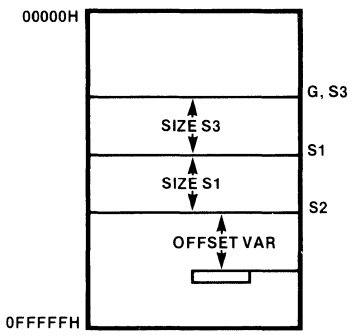
It is also possible to use the OFFSET operator to generate correct offsets within a group. Suppose you have a group G containing segments S1, S2, and S3, with VAR a variable in S2. The expression OFFSET VAR will give you the offset of VAR within S2, its segment. Since S2 is in G, you may need to refer on occasion to G:VAR, the location VAR as seen from the base-address of the group G.

OFFSET G:VAR then gives you the current offset at execution time, after LOCATE assigns absolute addresses to all the segments in G (S2 may be first or last or second). The use of G: in this expression is absolutely necessary, since without it the OFFSET operator will supply only the offset of VAR within the segment where it was defined. (See also LEA in Chapter 6.)

Example:

(NOTE—you need not master the details in this example. The assembler and locate facility handle it for you. These details are provided solely as a matter of more complete information, as to why and how the offset in a group is created.)

This is a picture of one way the OFFSET G:VAR might turn out.



The value of OFFSET G:VAR will include the sum of the size of S3, the size of S1, and OFFSET S2:VAR, but this sum must be adjusted for the align-types and ending points of S3, S2, and S1.

Suppose S3 has the align-type BYTE and is 24 bytes long (18H), and S1 and S2 are align-type WORD and are each 55 bytes long (37H). Suppose VAR is defined as the fifth word in S2, i.e., its relative offset is 8. Suppose further that during LOCATE, G and S3 are assigned the address 12343H, i.e., base-address 12340H (paragraph number 1234H) and offset 3. (See Chapter 1 on addressing and Chapter 4 on the Segment Directive).

Then the last byte of S3 falls at location 1235AH. S1 cannot begin at the next byte, 1235BH, because word alignment means the first byte of the segment falls at an even address. Thus S1 begins at 1235CH, with 1235BH unused. Similarly, S2 begins at 12394H, with 12393H unused.

Now what is OFFSET G:VAR? It is:

the size of S3 + the size of S1 + OFFSET S2:VAR + adjustment, or

24 + 55 + 8 + 5 = 92, or 5CH.

The 5 is added in to adjust for both the initial offset (3) of G (beyond its base address 12340H), and the 2 bytes left unused by aligning S1 and S2. Thus this offset, 5CH, plus the base-address of G, 12340H, gives the address of VAR, 1239CH, exactly 8 bytes past the beginning of S2.

Parentheses, Length, Size, Width, Square Brackets

These 5 operators have high precedence. WIDTH applies only to RECORDS and fields thereof, and is discussed only in Chapter 3.

Parentheses in an expression indicates a subexpression which needs to be reduced to a single number or variable before the other operators in the full expression can be done. In writing A * (B + C), you need the result of the addition before multiplying by A.

The LENGTH operator tells how many units (of whatever type) were allocated by the original line defining the variable. In the prior example of DATA_TABLES,

LENGTH A = 300, LENGTH Y = 100.

The SIZE operator tells how many bytes were defined by that original line of code. This number is calculated from the units and the type, by the formula:

SIZE name = LENGTH name * TYPE name.

SIZE A = 100
SIZE B = 600
SIZE Y = 400

As an example of using some of these operators, the following code sequence clears a block of storage in the ES segment, given that BLOCK was defined as a byte array.

```
ASSUME ES:SEG BLOCK, CS: RE_INIT_SEG
        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     LENGTH BLOCK
ZER:    MOV     BLOCK [DI], 0
        ADD     DI,     1
        LOOP    ZER
```

(The LOOP instruction automatically decrements CX by 1 and transfers to ZER, until CX becomes zero. See Chapter 6.)

If you wanted to use this sequence in several different modules to handle different byte or word arrays, then instead of:

```
ADD DI,1
```

you would write:

```
ADD  DI, TYPE BLOCK
```

the code then appears

```
ASSUME ES:SEG BLOCK, CS: RE_INIT_SEG

        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     LENGTH BLOCK
ZER:    MOV     BLOCK [DI], 0
        ADD     DI,     TYPE BLOCK
        LOOP    ZER
```

Then for each separate copy of this sequence, used with different arrays, the assembler would generate the correct "move", and reset your DI pointer/subscript by 1 or 2 depending on the type of block, i.e., the number of bytes per unit.

Although the following might take more execution time, it would be functionally equivalent if you left the ADD as it was and made these 2 changes:

1. change LENGTH to SIZE

2. change BLOCK [DI] to BYTE PTR BLOCK [DI]

The code would then read

```
ASSUME ES: SEG BLOCK, CS: RE_INIT_SEG

        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     SIZE BLOCK
ZER:    MOV     BYTE PTR BLOCK [DI], 0
        ADD     DI,     1
        LOOP    ZER
```

(This would also handle DD arrays. Note that each copy of this sequence would use a different array name corresponding to the particular definition. To avoid this and have only one copy, you would have to make the sequence into a procedure: see Chapter 4) The following is another way of achieving the same effects:

```
        LEA   DI, BLOCK
        MOV   AX, SEG  BLOCK
        MOV   ES, AX
        MOV   CX, SIZE  BLOCK
        MOV   AL, 0
        CLD
REP STOS BYTE PTR BLOCK   ; see Chapter on Instructions
```

Square Brackets Used As Subscripts

The usage of square brackets in the expression "BLOCK[DI]" is different from square brackets when used alone (e.g., just "[DI]"). This new usage of square brackets is called subscripting. It is similar, but not identical, to subscripting in a higher level language like FORTRAN or PL/M. Using an expression in square brackets as a subscript has the effect of adding the quantity in brackets to the offset component of the address expression appearing to the left of the brackets. This address expression may not appear to the right of the brackets.)

The value to the left of the subscript must be an address expression. The result of a subscript operation is always an address expression and must have a data type (e.g., BYTE, WORD, or DWORD). It is not legal to subscript labels.

Thus we see that BLOCK[DI] is an address expression. Moreover, the offset component of BLOCK[DI] is the same as [OFFSET BLOCK + DI]. Also, the address expression [BX][SI] is legal, since [BX] is a legal address expression using the defaults discussed above. An equivalent expression to [BX][DI] is [BX + DI]. The former involves subscripting, the latter does not. The result is the same.

The segment and type components of the new address expression are the same as those components of the address expression to the left of the subscript.

Recall the first rule for legal expressions in square brackets:

> Numbers may appear if at least one of the base or index registers appear.

This rule is relaxed in the case of subscripts; numeric expressions are allowed to appear alone in square brackets if the brackets are used as a subscript. Since the subscript is added BLOCK[3] would be the same as BLOCK + 3.

Examples:

For byte arrays of the same size:

```
        ASSUME CS: SWITCH_SEG, DS: DATA_TABLEs

        MOV    CX, LENGTH ARRAY1
        MOV    BX, OFFSET ARRAY2
        MOV    SI, 0
NEXT:   MOV     AL,ARRAY1[SI]
        MOV   BYTE  PTR [BX] [SI], AL
        INC    SI
        LOOP   NEXT
```

This sequence above fills each byte of ARRAY2 with the contents of the respective byte in ARRAY1. For byte or word arrays of the same size, you can change INC SI to ADD SI, TYPE ARRAY1. If the sizes are different, the loop must run on the smaller, i.e., both the MOV CX command and the one at NEXT must use the smaller array name, putting the other name in the MOV BX command.

If the type are different, you may be making a mistake, but you can achieve your goal using PTR and a separate index. Here are three examples, using the earlier definitions in the segment DATA_TABLES:

```
DATA_TABLES      SEGMENT PARA 'DATACLASS'

  A  DB          100  DUP  (0)
  B  DW          300  DUP  (47)
  Y  DD          100  DUP  (13)
     o
     o

DATA_TABLES ENDS

  SWITCH      SEGMENT   PARA   'CODE1CLASS'
                o
                o
                o
              ASSUME   CS:SWITCH, DS: DATA_TABLES
              MOV   CX, LENGTH A
              MOV   BX, OFFSET B
              MOV   SI,  0
              MOV   DI,  0

(EXAMPLE #1) NEXT:      MOV    AL, A[DI]
                        MOV    [BX] [SI], AL
                        ADD    SI, TYPE B
                        ADD    DI, TYPE A
                        LOOP   NEXT
```

This loop places each of A's 100 bytes into the low-order byte of each of B's first 100 words, such that A[#] = LOW B[#]

```
(EXAMPLE #2) NEXT:      MOV    AL, A[DI]
                        MOV    [BX] [SI + 1], AL
                        ADD    SI,TYPE B
                        ADD    DI,TYPE A
                        LOOP   NEXT
```

This performs similarly to Example 1 above but the high-bytes of B's first 100 words are filled instead, such that A[#] = HIGH B [#]

```
(EXAMPLE #3) NEXT:      MOV   AX, WORD PTR A[DI]
                        MOV   [BX] [SI], AX
                        ADD   SI, TYPE B
                        ADD   DI, TYPE B

                        CMP   DI, SIZE A
                        JGE   A__USED__PROC

                        LOOP  NEXT
                A__USED__UP:  CALL NEW__PROC
                              o
                              o
                              o
  SWITCH    ENDS
```

This makes the first 100 bytes of B identical with A's 100 bytes. Instead of the compare and conditional jump, you could have halved the size of A put into CX, i.e., the first line after ASSUME could have read

```
   MOV  CX,  LENGTH  A/2
```

or

```
   MOV  CX,  LENGTH  A  SHR  1
```

Example 3 could be accomplished using the string block-move instructions in the following two ways:

1.

```
      ASSUME  CS: SWITCH,DS:  DATA__TABLES,
      &       ES: DATA__TABLES          ; the block-move
                                        ; requires ES
      MOV   CX,  LENGTH  A
      LEA   SI,  A
      LEA   DI,  B
      OLD
   X3:  REP   MOVS BYTE PTR B, A
```

2.

```
      ASSUME     CS; SWITCH,DS: DATA__TABLES,
      &          ES; DATA__TABLES
      MOV   CX,  LENGTH A/TYPE B
      LEA   SI,A
      LEA   DI,B
      CLD
   X3:  REP   MOVS B, WORD PTR A   ; half the number of moves
```

Using Square Brackets in Transfers

(Indirect Transfers)

Square brackets do have another use, and it relies on an interpretation similiar to the above. When [BX] is used, it is taken to mean "use (the data at) the location whose offset is in BX". In the examples above, data was being stored into such a location, but it is equally valid to write

```
MOV  AH, DS: [BX] [SI]
```

This causes the accumulator's high-byte to receive the data stored at the SIth byte past the location whose offset is in BX, in the segment whose base-address is in DS. (Effectively this amounts to the byte whose address is the contents of SI plus the contents of BX plus 16 times the contents of DS.) If you were to write

```
MOV  AX, DS:[BX] [SI]
```

then the full-word accumulator would be filled with the word in DS beginning at the SIth byte past the location whose offset is in BX.

The OTHER use of square brackets is in jumps or calls. Jumps and calls are always in the CS segment, i.e., using the contents of CS as the paragraph-number of the segment-base-address. When you write

```
JMP  BX     ;DIRECT JUMP
```

control is transferred to a location in the current CS segment using the contents of BX as an offset from the current CS base-address. Thus if BX contains OC12H, JMP BX transfers to the location 0C12H beyond the beginning of the CS segment.

If, however, you write

```
JMP  WORD  PTR  [BX]     ; INDIRECT JUMP
```

the square brackets mean "use the contents of the word whose offset is in BX." This usage automatically means the data is in the DS segment. Thus the contents of BX are used as an offset (to the segment address in DS) to locate the indicated word. Then that word's contents are used for the transfer, replacing the Instruction Pointer.

In the example above, BX contained 0C12H. When the instruction you code is JMP WORD PTR [BX], the transfer does not use 0C12H as the offset to CS, but rather uses the contents of DS:0C12H as the offset. If the word at 0C12H in the DS segment contains 0FAFH, JMP WORD PTR [BX] transfers control to 0FAFH in the current CS segment.

The WORD PTR preceding [BX] indicates use of one word as an offset, leaving the contents of CS unchanged as the segment address. The only other choice for this construction is

```
JMP  DWORD  PTR  [BX]
```

which would use two words at that address in DS. The first, as above, is the offset. The next word is used to replace the contents of CS, making this an intersegment jump (also called "long" or "FAR").

Subscripting such a reference is also legal, e.g.,

```
JMP  WORD  PTR  [BX] [SI]
```

which would replace the IP with the word beginning at the SIth byte post the location whose offset is in BX. Similarly, it is valid to use a subscripted address expression in a jump, e.g.,

```
JMP  TABLE  [BX] [DI]
```

where the type of entry in the table must be word or doubleword.

The remainder of this chapter discusses the following operators: SHORT, OR, XOR, AND, NOT*, /, MOD, SHL, SHR, HIGH, and LOW. Relocatable numbers and expressions are discussed in Appendix G.

All constants are stored internally as 17-bit numbers, that is, a left-most bit for the sign of the constant (0=plus, 1=minus) and a 16-bit value. The signs of the operands are affected by the logical and shifting operators, as is discussed in each case.

SHORT

Causes the assembler to expect only 1 byte to be enough to hold the ultimate value of the expression. This means machine code will only be generated if the evaluation of the expression yields a single byte value. If the result is larger than a byte, you will get an error message. If an expression using SHORT undergoes further arithmetic, the operator loses all effect. Applying SHORT to a backward reference has no effect.

OR, XOR

Create the inclusive or exclusive logical "or" of the operands. Inclusive-or means the result has a 1 in all bit positions where either operand had a 1. Exclusive-or means the result has a 1 where only one operand had a 1 and the other had a 0, and the result has a 0 where both operands had the same value, i.e., both 1s or both 0s.

Examples:

```
    11010110B                    11010110B
OR  01010101B      but     XOR   01010101B
    11010111B                    10000011B
```

Showing these in a vertical format makes it easier to compare bit by bit, but the usual form would be horizontal in an instruction, i.e.,

```
VALUE_D6H    EQU    11010110B    ; = 0D6H
VALUE_55H    EQU    01010101B    ; = 55H
MOV    AX, VALUE_D6H OR VALUE_55H ; AX = D7H.
MOV    BX, 11010110B XOR 01010101B ; BX = 83H.
```

AND

Creates the logical conjunction of the 2 operands, meaning the result has a 1 only in those bit positions where BOTH operands had a 1.

Examples:

(actually, the sign bit is included, as above);

```
    11010110B                1 11111111 11111011    (-5)
AND 01010101B        AND     0 00000000 00010101    (21)
    01010100B                0 00000000 00 010001   =1 7
```

AND is sometimes used to select certain bits or a pattern of bits out of a larger value. This is sometimes called masking the desired bits, or masking out the rejected bits. If you use the mask 00001111B ANDed with some byte in memory, the result will be a byte whose lower half is the same as the original, with an upper half of all zeroes:

```
    10111101B            10010110B
AND 00001111B    AND     00001111B
    00001101B            00000110B
```

(See also discussion of Records in Chapter 3.)
When AND is combined with "ORs", AND is done first:

```
MASK1   EQU   00001111B

1. MOV MEM_WORD, 10010111B  AND  MASK1  XOR  1110B

2. MOV MEM_WORD, 10010111B  XOR  MASK1  AND  1110B
```

The instruction at (1) moves 00001001B into MEM_WORD, as follows:

```
    10010111B            00000111B
AND 00001111B    XOR     00001110B
    00000111B            00001001B
```

Whereas the instruction at (2) moves 10011000B into MEM_WORD, as follows:

```
(MASK 1 =)          1111B              10010111B
            AND     1110B      XOR     00001110B
                    1110B              10011001B
```

NOT

Forms the "ones" complement of its operand, i.e., each original zero becomes a one and each original one becomes a zero.

```
NOT 01011011B = 10100100B
```

When NOT is combined with ANDs or ORs, NOT is done first:

```
MOV     MEM_WORD, 97H   AND   NOT   MASK1
```

moves 10010000B into MEM_WORD, as follows:

```
NOT MASK1 = NOT 00001111B = 11110000B
97H = 10010111B              AND 10010111B
                                 10010000B
```

```
MOV     DX, 97H   AND   NOT   MASK1   XOR   1110B
```

moves 10011110B into DX, because XOR is done last.

```
MOV   BX, 97H   XOR   NOT   MASK1   AND   1110B
```

moves 97H into BX. NOT is done first, AND second, XOR last. The AND produces all zeros, so the XOR effectively duplicates the 97H.

```
((any_value   OR 0 = any_value
  any_value XOR 0 = any_value
  any_value AND 0FFFFH = any_value ))
```

NOTE: if NOT yields 10000H, it is converted to zero: NOT 0FFFFH =0

Relational Operators: EQ, NE, LT, LE, GT, GE

These operators compare 2 operands, giving a result of all 1's if the specified relation is true, and all 0's if not. In the order listed above, the operators mean: equal, not equal, less than, less than or equal (i.e., not greater), greater than, greater than or equal (i.e., not less). To compare variables or labels, they must be defined in the same segment. These operators compare the offsets of such operands.

These operators yield a 16-bit result of all ones if the relation is TRUE, or all zeroes if it is FALSE. One way to use them in expressions, then, is to use AND:

```
MOV     AL, 25 AND A NE B
```

will mean either

```
MOV     AL, 25    or   MOV AL, 0
```

depending on the assembly-time value of A being unequal or equal to the value of B, respectively.

```
TAB DW ((NEW LT OLD) AND OLD + (NEW GE OLD) AND NEW) DUP (1)
```

TAB is a word array, whose length is the larger of NEW and OLD.

NOTE: Variables and labels may only be compared if they are in the same segment. Numbers may not be compared to variables or labels.

Multiplicative and Shifting Operators

* Multiplication

/ Division. (Division by zero causes an error.)

MOD Modulo. Result is the remainder caused by a division operation (7 MOD 3 = 1). Absolute-number operands only.

Examples:

1. The following expressions generate the bit pattern for the ASCII character A (65D or 41H):

```
5+30*2

(25/5)+(30*2)

5+(-30*-2)
```

2.
```
SAMPLE_NUMBER  MOD  8
```

NOTE

The MOD operator must be separated from its operands by spaces.

If SAMPLE_NUMBER has the value 25, the expression above evaluates to the value 1.

Shift Operators

The shift operators are as follows:

Operator	Meaning
y SHR x	Shift operand 'y' to the right 'x' bit positions.
y SHL x	Shift operand 'y' to the left 'x' bit positions.

The shift operators do not "wraparound" any bits shifted out of the byte (as do the rotate instructions, for which there are no corresponding operators). Bit positions vacated by shifting are zero-filled. The shift operator must be separated from its operands by spaces. Both y and x must be absolute numbers.

Example:

Assume that NUMBER has the value 01010101. The effect of the shift operators is as follows:

```
NUMBR SHR 2     00010101

NUMBR SHL 1     10101010
```

A shift one bit position to the left has the effect of multiplying a value by two; a shift one bit position to the right has the effect of dividing a value by two. Therefore shifting N positions left is equivalent to multiplying by 2 to the Nth, and shifting right N positions effectively divides by 2 to the Nth. Shifting by a negative number reverses the direction of the shift, i.e.,

```
NUMBR  SHR  -2
```

is identical to

```
NUMBR  SHL  2
```

The sign bit is not affected by SHL or SHR.

Byte Isolation Operators

The byte isolation operators are as follows:

Operator	Meaning
HIGH	Isolate high-order 8 bits of 16-bit value.
LOW	Isolate low-order 8 bits of 16-bit value.

In some cases, you need to deal only with one byte of a word. This is the function of the HIGH and LOW operators. You can take the HIGH or LOW of a relocatable quantity. These operators are for support of 8080-to-8086 translation only, and are not recommended for 8086 programming.

If it occurs that HIGH or LOW is applied again to a relocatable quantity (called RQ), then the following will result:

```
LOW   LOW    RQ = LOW    RQ
LOW   HIGH   RQ = HIGH   RQ
HIGH  LOW    RQ = 0
HIGH  HIGH   RQ = 0
```

CHAPTER 6
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.

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)

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:

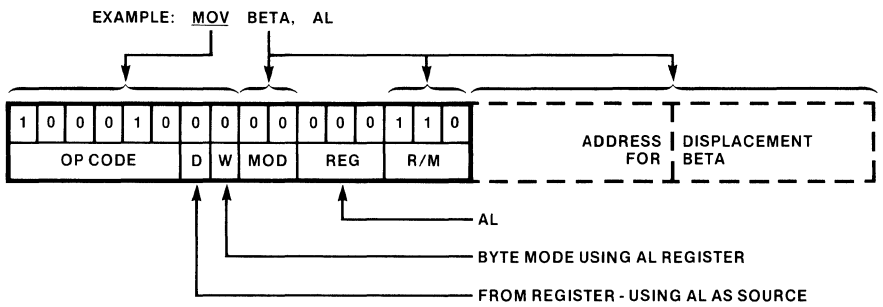
Name
Constant
Delimiter

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 6-157 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 7 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:

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

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

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

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

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

In all cases above, the source operand is addressed by SI and the destination operand is addressed by DI. Only in CMPS 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 6-2 shows the available instructions, the conditions associated with them, and their interpretation.

Table 6-2. 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	CF = 1	"below" or "not above or equal"
JBE or JNA	(CF or ZF) = 1	"below or equal" or "not above"
JP or JPE	PF = 1	"parity" or "parity even"
J0	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	CF = 0	"not below" or "above or equal"
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).

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

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

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:

0 0 0 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) 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:

0 0 0 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: 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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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

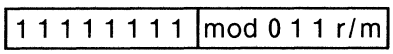
Timing: 20 clocks

Examples:

```
CALL  FAR_LABEL
CALL  FAR_PROC
```

CALL

Inter-Segment Indirect:



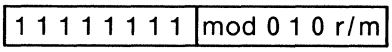
DEST = (EA), SEG = (EA + 2)

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 1). 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 Chapters 4 and 5.

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 (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

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 1). 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 Chapters 4 and 5.

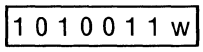
Whenever an address-expression including a variable name is used, i.e. a name for a memory location containing data, the MODRM byte will be followed by 2 bytes giving the computed displacement from the segment-base-address.

CMPS (Compare byte string, compare word string)

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:



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.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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.

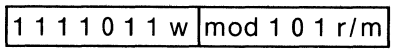
IMUL
(Integer multiply accumulator by register-or-memory; signed)

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:



(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 1). 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 Chapters 4 and 5.

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.

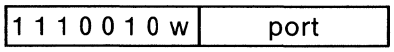
IN (Input byte and input word)

Operation: The contents of the accumulator are replaced by the contents of the designated port.

(DEST)←(SRC)

Encoding:

Fixed Port:



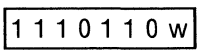
if w = 0 then SRC = port, DEST = AL
else SRC = port + 1:port, DEST = AX

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)

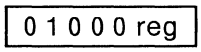
Operation: The specified operand is incremented by 1. There is no carry out of the most-significant bit.

(DEST) ← (DEST) + 1

See note.

Encoding:

Register Operand: (Word)



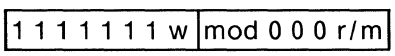
DEST = REG

Timing: 2 clocks

Examples:

```
INC  AX
INC  DI
```

Memory or Register Operand:



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 1). 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 Chapters 4 and 5.

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

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).

JA

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.

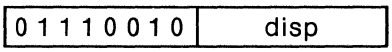
JB

JB and JNAE (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 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:



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.

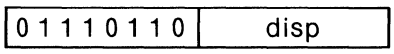
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

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.

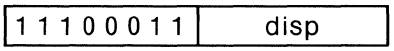
JCXZ

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.

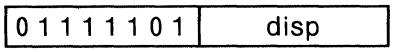
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

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 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:

0 1 1 1 1 1 0 0	disp

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.

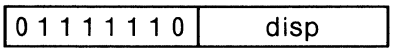
"Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

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:



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

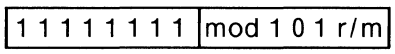
DEST = offset, SEG = seg

Timing: 7 clocks

Examples:

```
JMP  LABEL_DECLARED_FAR
JMP  FAR  PTR  LABEL_NAME
JMP  FAR  PTR  NEAR_LABEL
```

Inter-Segment Indirect:



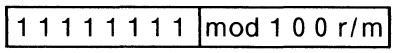
DEST = (EA), SEG = (EA + 2)

Timing: 16 + EA clocks

Examples:

```
JMP  VAR_DOUBLEWORD
JMP  DWORD  PTR  [BX] [SI]
JMP  ALPHA [BP] [DI]
```

Intra-Segment or Intra-Group Indirect:



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 1). 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 Chapters 4 and 5.

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:

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
```

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)

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
```

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:

0 1 1 1 0 1 0 1	disp

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.

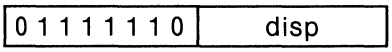
JNG

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:



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:

0 1 1 1 1 1 0 0	disp

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.

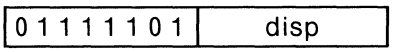
JNL

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.

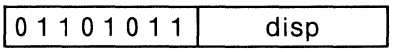
NOTE: The target label must be within −128 to +127 bytes of this instruction.

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:

0 1 1 1 1 0 0 1	disp

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.

JPE

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 1 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.

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 1). 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 Chapters 4 and 5.

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

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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)

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
(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.

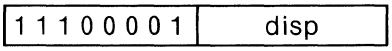
The target label must be within −128 to + 127 bytes of this instruction.

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:

1 1 1 0 0 0 0 1	disp

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

1 0 0 0 1 1 0 0	mod 0reg r/m

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

1 0 0 0 1 0 d w	mod reg r/m	addr-low*	addr-high*

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 1). 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 Chapters 4 and 5.

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)

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:

1 0 1 0 0 1 0 w

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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.

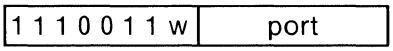
OUT (Output byte and output word)

Operation: The contents of the designated port are replaced by the contents of the accumulator.

(DEST) ← (SRC)

Encoding:

Fixed Port:



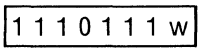
if w = 0 then SRC = AL, DEST = port
else SRC = AX, DEST = port + 1:port
(0 < port < 255)

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 (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 1). 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 Chapters 4 and 5.

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 1). Except for I/O and interrupts, the choice of segment register depends on the address-expression in the assembly language source lines and on the ASSUME directive which applies to that line. See Chapters 4 and 5.

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

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 1). 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 Chapters 4 and 5.

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

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 1r/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 1). 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 Chapters 4 and 5.

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

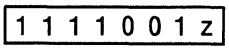
REP/REPZ/REPE/REPNE/REPNZ (Repeat string operation)

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:



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

1 1 0 0 0 0 1 1

Timing: 8 clocks

Example: RET

Intra-Segment and Add Immediate to Stack Pointer:

1 1 0 0 0 0 1 0	data-low	data-high

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:

1 1 0 0 1 0 1 1

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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.

SAL

SHL and SAL (Shift logical left and shift arithmetic left)

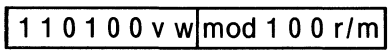
Operation: The specified destination (leftmost) operand is shifted left COUNT times. Its high order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move up" one position, e.g., the value of the third bit is replaced by the value of the second bit. The vacated 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:




```
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 1). 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 Chapters 4 and 5.

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)

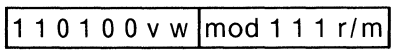
Operation: The specified destination (leftmost) operand is shifted right COUNT times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the destination "move down" one position, e.g., the value of the second bit is replaced by the value of the third bit. The vacated 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:




```
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 1). 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 Chapters 4 and 5.

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:

0 0 0 1 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) 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:

0 0 0 1 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) 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 1). 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 Chapters 4 and 5.

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

SCAS (Scan byte string or scan word string)

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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

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

STOS (Store byte string or store word string)

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 1). 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 Chapters 4 and 5.

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 1). 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 Chapters 4 and 5.

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

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)

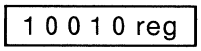
3) The former contents of the destination are moved from the work register into the source operand

(SRC) ← (Temp)

Flags Affected: None

Type 1:

Register Operand with Accumulator:



SRC = REG, DEST = AX

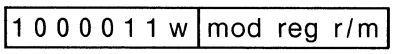
Timing: 3 clocks

Examples:

```
XCHG  AX,  BX
XCHG  SI,  AX
XCHG  CX,  AX
```

Type 2:

Memory or Register Operand with Register Operand:



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 1). 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 Chapters 4 and 5.

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

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 1). 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 Chapters 4 and 5.

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 7
CODE MACROS INTRODUCTION

In this chapter the word macro always means codemacro.

A macro 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 macro. 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 macro 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 macro 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 macro.

In fact, the MOV and ADD instruction above are macros. The assembler's entire instruction set is defined and implemented as a large number of macros. (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 code macro, to distinguish it from text macros. The latter are more familiar to programmers because previous assembly languages have included such a facility. Text macros are not discussed in this manual. The presentation below will describe creating and using code macros.

These macros are encoded at macro definition time into a very compact form, so that all defined codemacros may reside simultaneously in memory. Each definition specifies a certain combination of parameters and will match only those. Other combinations of parameters may be accommodated by redefining the codemacro. Multiple definitions of the same codemacro name are chained together; so that when the codemacro is called, each link of the chain can be checked for a match of operands.

Since the 8086 instruction set consists of codemacros, it is natural to refer to a codemacro being called as an "instruction"; and to refer to its actual parameters as "operands".

For example, the language has an ADD instruction that works properly with any general register or memory location as a destination operand or as a source operand, and also works with immediate-data operands. This is achieved by defining 11

codemacros to generate the 11 different machine instructions appropriate to these different cases and combinations. The correct one is used because the specification of its formal parameters is matched by the actual parameters supplied in your source code. The details of how this works are covered in this chapter.

The definition of a codemacro begins with a line specifying its name and a list of its formal parameters, if any:

```
CODEMACRO name [formal_list]
```

or

```
CODEMACRO name PREFX
```

where formal_list is a list of 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 name
```

On this line, the name is optional, but including it is a good idea to keep things clear, and to provide an error check for your code. If given, the name must match the name given in the CODEMACRO line.

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 STC

Codemacro  PUSHF
DB 9CH     ; pushes all flags into top word on stack.
Endm PUSHF

Codemacro  ADD dst:Ab, src:Db
DB 04H
DB src
Endm ADD
```

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 ADD
```

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 ADD
```

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 ADD

Codemacro ADD dst:Ew,src:Db (-128, 127)
SEGFIX dst
DB 83H
MODRM 0,dst
DB src
Endm ADD
```

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 JMP

Codemacro JE place:Cb
DB 74H
RELB place
Endm JE
```

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        RET
```

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 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 ASM86 Operator's Guide.

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   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

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

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 SI_ptr:Eb
  Segfix    SI_ptr
  DB        0ACH
  EndM

CodeMacro   LodS SI_ptr:Ew
  Segfix    SI_ptr
  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:Eb, DI_ptr:Eb
  NoSegfix  ES, SI_ptr
  Segfix    DI_ptr
  DB        0A4H
  EndM

CodeMacro   MovS  SI_ptr:Ew, DI_ptr:Ew
  NoSegfix  ES, SI_ptr
  Segfix    DI_ptr
  DB        0A5H
  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:Eb
  NoSegfix  ES, DI_ptr
  DB        0AEH
  EndM
```

```
CodeMacro   ScaS  DI_ptr:Ew
  NoSegfix  ES, DI_ptr
  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:Eb
  NoSegfix  ES, DI_ptr
  DB        0AAH
  EndM

CodeMacro   StoS  DI_ptr:Ew
  NoSegfix  ES, DI_ptr
  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:E
  Segfix    table
  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).

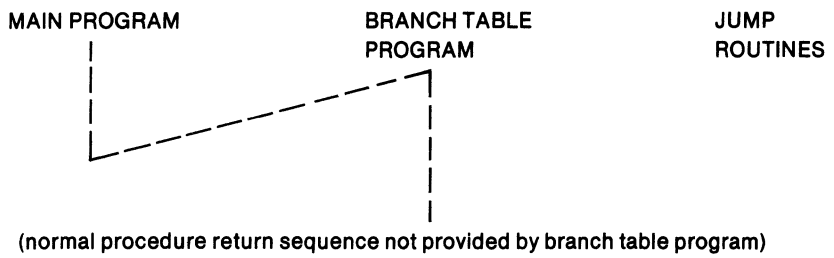
Examples 3, 4, and 5 discuss additional methods of passing data and parameters to procedures, illustrating the use of both the registers and the stack for passing parameters. Examples 6 and 7 cover multibyte addition and subtraction. Interrupt procedures and timing loops are described in examples 8 and 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.

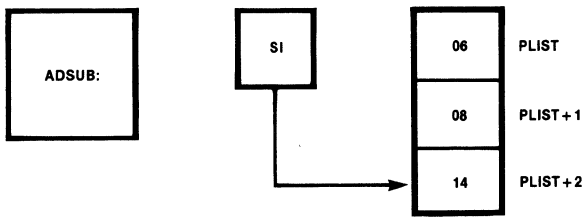
Example 3 shows the use of memory to pass parameters. Registers are used for this in Example 4. Example 5 uses a stack.

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:

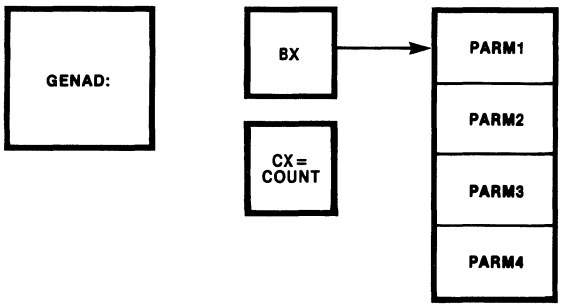
```
PARAMS  SEGMENT
PLIST       DB      6
            DB      8
            DB      ?
LIST2       DB      10
            DB      35
            DB      ?
              .
              .
              .
PARAMS  ENDS
STACK   SEGMENT
        DW  4  DUP  (?)
STACK_TOP  LABEL WORD
STACK   ENDS
ADDING  SEGMENT
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
```

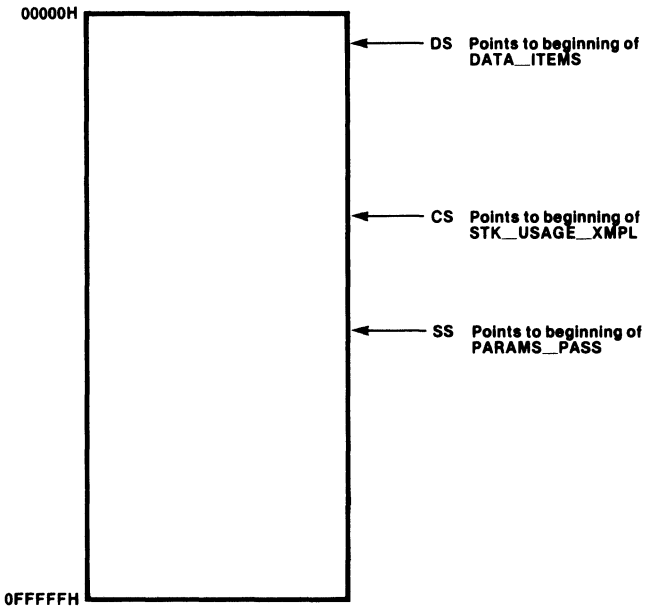


Figure 1

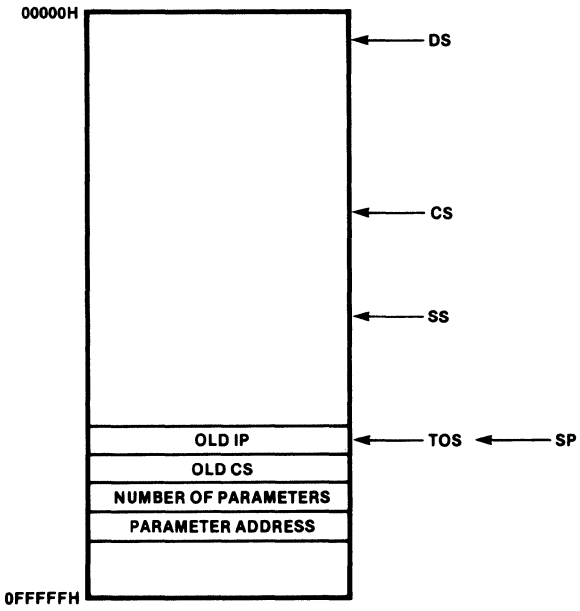


Figure 2

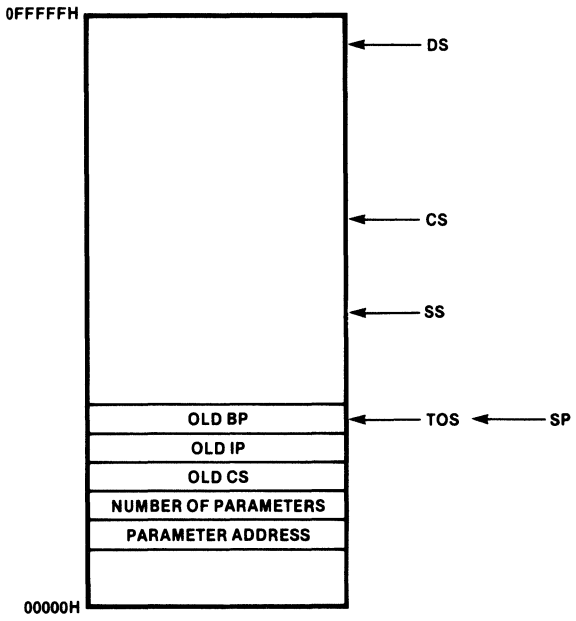


Figure 3

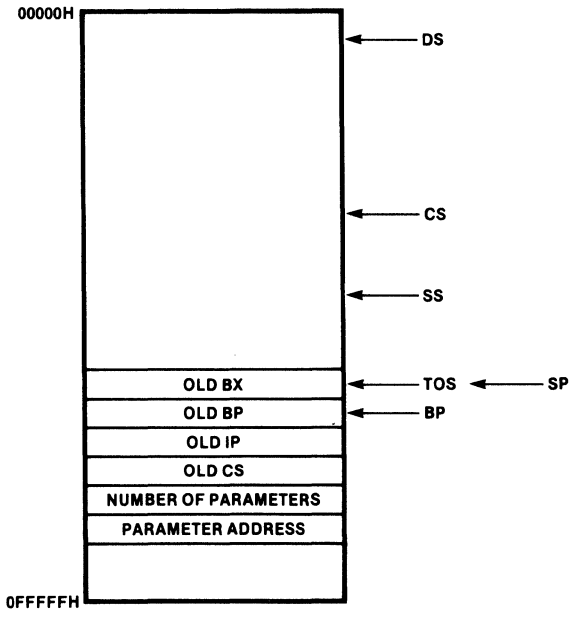


Figure 4

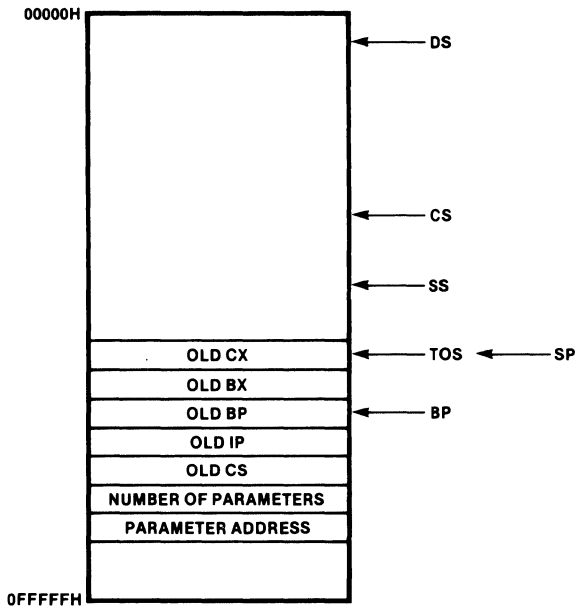


Figure 5

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	AL,PORT	BYTE OUTPUT (AL) TO PORT
E7 11100111		OUT	AX,PORT	WORD OUTPUT (AX) TO PORT
E8 11101000		CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001		JMP	DISP16	DIRECT INTRA SEGMENT JUMP
EA 11101010		JMP	DISP16,SEG16	DIRECT INTER SEGMENT JUMP
EB 11101010		JMP	DISP8	DIRECT INTRA SEGMENT JUMP
EC 11101010		IN	AL,DX	BYTE INPUT FROM PORT (DX) TO REG AL
ED 11101010		IN	AX,DX	WORD INPUT FROM PORT (DX) TO REG AX
EE 11101010		OUT	DX	BYTE OUTPUT (AL) TO PORT (DX)
EF 11101010		OUT	DX	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)

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 i,d	WAIT	PUSHF	POPF	SAHF	LAHF
A	**TEST b,i,a**	**TEST w,i,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 i,d	JMP si,d	**IN v,b**	**IN v,w**	**OUT v,b**	**OUT v,w**
F	CLC	STC	CLI	STI	CLD	STD	Grp 2 b,r/m	Grp 2 w r/m

where

mod□r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	—	SAR
Grp 1	TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2	INC	DEC	CALL id	CALL i,id	JMP id	JMP i id	PUSH	—

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

b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed to accum
id = indirect
is = immed byte, sign ext.
I = 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	FAC	JMP	LAHF	POPF	STC
AAA	CLI	HLT	JNA	LDS	PUSH	STD
AAD	CMC	IDIV	JNAE	LEA	PUSHf	STI
AAM	CMP	IMUL	JNB	LES	RCL	STOS
AAS	CMPS	IN	JNBE	LOCK	RCR	SUB
ADC	CS	INC	JNE	LODS	REPE	TEST
ADD	CWD	INT	JNG	LOOP	REPNE	WAIT
AH	CX	INTO	JNGE	LOOPE	REPNZ	XCHG
AL	DAA	IRET	JNLE	LOOPNE	REPZ	XLAT
AX	DAS	JA	JNO	LOOPNZ	RET	??SEG
BH	DEC	JAE	JNP	LOOPZ	ROR	
BL	DH	JB	JNS	MOV	SAHF	
BP	DI	JBE	JNZ	MOVS	SAL	
BX	DIV	JCXZ	JO	MUL	SAR	
CALL	DL	JE	JP	NEG	SBB	
CBW	DS	JG	JPE	NIL	SCAS	
CH	DX	JGE	JPO	NOP	SI	
CL	ES	JL	JS	OUT	SP	
CLC	ESC	JLE	JZ	POP	SS	

NON-CONFLICTING KEYWORDS

DEBUG	NOPRINT
ERRORPRINT	NOSYMBOLS
MEMORY	OBJECT
NODEBUG	PAGING
NOERRORPRINT	PRINT
NOOBJECT	STACK
NOPAGING	SYMBOLS

HANDS-OFF KEYWORDS

ABS	ENDS	LOW	PREFX	STACK
ASSUME	EQ	LT	PROC	THIS
AT	EQU	MASK	PROCLEN	TYPE
BYTE	EXITM	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.

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet 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 of 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 for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of 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 _____

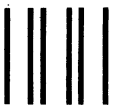
ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

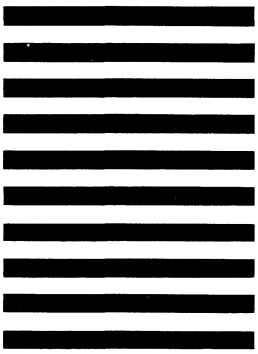
Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments 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.