8086 Family Assembly Language Programming—Introduction

Objectives

At the conclusion of this chapter, you should be able to:

1. *Write a task list, flowchart, or pseudocode for a simple programming problem.*
2. *Write, code or assemble, and run a very simple assembly language program.*
3. *Describe the use of program development tools such as editors, assemblers, linkers, locators, debuggers, and emulators.*
4. *Properly document assembly language programs.*

PROGRAM DEVELOPMENT STEPS

Defining the Problem

The last chapter showed you the format for assembly language instructions and introduced you to a few 8086 instructions. Developing a program, however, requires more than just writing down a series of instructions. When you want to build a house, it is a good idea to first develop a complete set of plans for the house. From the plans you can see whether the house has the rooms you need, whether the rooms are efficiently placed, and whether the house is structured so that you can easily add on to it if you have more kids. You have probably seen examples of what happens when someone attempts to build a house by just putting pieces together without a plan.

Likewise, when you write a computer program, it is a good idea to start by developing a detailed plan or outline for the entire program. A good outline helps you to break down a large and seemingly overwhelming programming job into small modules which can easily be written, tested, and debugged. The more time you spend organizing your programs, the less time it will take you to write and debug them. You should *never* start writing an assembly language program by just writing down instructions! In this chapter we show you how to develop assembly language programs in a systematic way.

The first step in writing a program is to think very carefully about the problem that you want the program to solve. In other words, ask yourself many times, "What do I really want this program to do?" If you don't do this, you may write a program that works great but does not do what you need it to do. As you think about the problem, it is a good idea to write down exactly what you want the program to do and the order in which you want the program to do it. A good outline helps you to break down a large and seemingly overwhelming programming job into small modules which can easily be written, tested, and debugged. The more time you spend organizing your programs, the less time it will take you to write and debug them. You should *never* start writing an assembly language program by just writing down instructions! In this chapter we show you how to develop assembly language pro-grams in a systematic way. At this point you do not write down program statements, you just write the operations you want in general terms. An example for a simple programming problem might be

1. Read temperature from sensor.
2. Add correction factor of +7.
3. Save result in a memory location.

For a program as simple as this, the three actions desired are very close to the eventual assembly language statements. For more complex problems, however, we develop a more extensive outline before writing the assembly language statements. The next section shows you some of the common ways of representing program operations in a program outline.

Representing Program Operations

The formula or sequence of operations used to solve a programming problem is often called the *algorithm* of the program. The following sections show you two common ways of representing the algorithm for a program or program segment.

FLOWCHARTS

If you have done any previous programming in BASIC or in FORTRAN, you are probably familiar with *flowcharts*. Flowcharts use graphic shapes to represent different types of program operations. The specific operation desired is written in the graphic symbol. Fig. 3.1 shows some of the common flowchart symbols. Plastic templates are available to help you draw these symbols if you decide to use them for your programs.

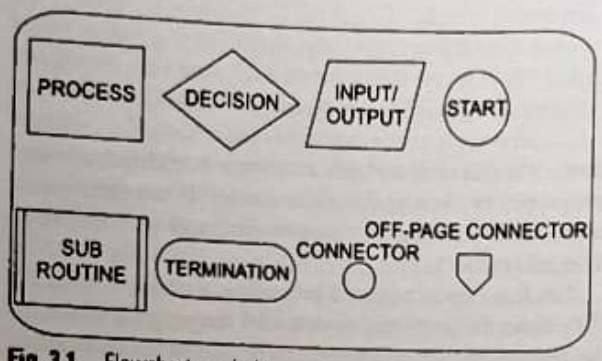

Fig. 3.1 Flowchart symbols.

Figure 3.2, shows a flowchart for a program to read in 24 data samples from a temperature sensor at 1-hour intervals, add 7 to each, and store each result in a memory location.

Fig. 3.2 Flowchart for program to read in 24 data samples from a port, correct each value, and store each in a memory location.

A racetrack- or circular-shaped symbol labeled START is used to indicate the *beginning* of the program. A parallelogram is used to represent an input or an *output* operation. In the example, we use it to indicate reading data from the temperature sensor. A rectangular box symbol is used to represent *simple operations* other than input and output operations. The box containing "add 7" in Fig. 3.2 is an example.

A rectangular box with double lines at each end is often used to represent a *subroutine* or *procedure* that will be written separately from the main program. When a set of operations must be done several times during a program, it is usually more efficient to write the series of operations once as a separate subprogram, then just "call" this subprogram each time it is needed. For example, suppose that there are several places in a program where you need to compute the square root of a number. Instead of writing the series of instructions for computing a square root each time you need it in the program, you can write the instruction sequence once as a separate procedure and put it in memory after the main program. A special instruction allows you to call this procedure each time you need to compute a square root. Another special instruction at the end of the procedure program returns execution to the main program. In the flowchart in Fig. 3.2, we use the double-ended box to indicate that the "wait 1 hour" operation will be programmed as a procedure. Incidentally, the terms *subprogram, subroutine,* and *procedure* all have the same meaning. Chapter 5 shows how procedures are written and used.

A diamond-shaped box is used in flowcharts to represent a *decision* point or crossroad. Usually it indicates that some condition is to be checked at this point in the program. If the condition is found to be *true,* one set of actions is to be done; if the condition is found to be *false,* another set of actions is to be done. In the example flowchart in Fig. 3.2, the condition to be checked is whether 24 samples have been read in and processed. If 24 samples have not been read in and processed, the arrow labeled NO in the flowchart indicates that we want the computer to jump back and execute the read, add, store, and wait steps again. If 24 samples have been read in, the arrow labeled YES in the flowchart of Fig. 3.2 indicates that all the desired operations have been done. The racetrack-shaped symbol at the bottom of the flowchart indicates the *end* of the program.

The two additional flowchart symbols in Fig. 3.1 are *connectors.* If a flowchart column gets to the bottom of the paper, but not all the program has been represented, you can put a small circle with a letter in it at the bottom of the column. You then start the next column at the top of the same paper with a small circle containing the same letter. If you need to continue a flowchart to another page, you can end the flowchart on the first page with the five-sided off-page connector symbol containing a letter or number. You then start the flowchart on the next page with an off-page connector symbol containing the same letter or number.

For simple programs and program sections, flowcharts are a graphic way of showing the operational flow of the program.

We will show flowcharts for many of the program examples throughout this book. Flowcharts, however, have several disadvantages. First, you can't write much information in the little boxes. Second, flowcharts do not present information in a very compact form. For more complex problems, flowcharts tend to spread out over many pages. They are very hard to follow back and forth between pages. Third, and most important, with flowcharts the overall structure of the program tends to get lost in the details. The following section describes a more clearly *structured* and *compact* method of representing the algorithm of a program or program segment.

STRUCTURED PROGRAMMING AND PSEUDOCODE OVERVIEW

In the early days of computers, a single brilliant person might write even a large program single-handedly. The main concerns in this case were, "Does the program work?" and "What do we do if this person leaves the company?" As the number of computers increased and the complexity of the programs being written increased, large programming jobs were usually turned over to a team of programmers. In this case the compatibility of parts written by different programmers became an important concern. During the 1970s it became obvious to many professional programmers that in order for team programming to work, a systematic approach and standardized tools were absolutely necessary.

One suggested systematic approach is called *top-down design*. In this approach, a large programming problem is first divided into major *modules*. The top level of the outline shows the relationship and function of these modules. This top level then presents a one-page overview of the entire program. Each of the major modules is broken down into still smaller modules on following pages. The division is continued until the steps in each module are clearly understandable. Each programmer can then be assigned a module or set of modules to write for the program. Another advantage of this approach is that people who later want to learn about the program can start with the overview and work their way down to the level of detail they need. This approach is the same as drawing the complete plans for a house before starting to build it.

The opposite of top-down design is *bottom-up design*. In this approach, each programmer starts writing low-level modules and hopes that all the pieces will eventually fit together. When completed, the result should be similar to that produced by the top-down design. Most modern programming teams use a combination of the two techniques. They do the top-down design first, then build, test, and link modules starting from the smallest and working upward.

The development of standard programming methods was helped by the discovery that any desired program operation could be represented by three basic types of operation. The first type of operation is *sequence*, which means simply doing a series of actions. The second basic type of operation is *decision*, or *selection*, which means choosing between two alternative actions. The third basic type of operation is *repetition*, or *iteration*, which means repeating a series of actions until some condition is or is not present.

On the basis of this observation, the suggestion was made that programmers use a set of three to seven standard *structures* to represent all the operations in their programs. Actually, only three structures, SEQUENCE, IF-THEN-ELSE, and WHILE-DO, are required to represent any desired program action, but three or four more structures derived from these often make programs clearer. If you have previously written programs in a structured language such as Pascal, then these structures are probably already familiar to you. Fig. 3.3, uses flowchart symbols to represent the commonly used structures so that you can more easily visualize their operation. In actual program documentation, however, English-like statements called *pseudocode* are used rather than the space-consuming flowchart symbols. Fig. 3.3 also shows the pseudocode format and an example for each structure.

Each structure has only *one entry point* and *one exit point*. As you will see later, this feature makes debugging the final program much easier. The output of one structure is connected to the input of the next structure. Program execution then proceeds through a series of these structures.

Any structure can be used within another. An IF- THEN-ELSE structure, for example, can contain a sequence of statements. Any place that the term *statement(s)* appears in Fig. 3.3, one of the other structures could be substituted for it. The term *statement(s)* can also represent a subprogram or procedure that is called to do a series of actions. Now, let's look more closely at these structures.

STANDARD PROGRAMMING STRUCTURES

The structure shown in Fig. 3.3*a* is an example of a simple sequence. In this structure, the actions are simply written down in the desired order. An example is

Read temperature from sensor.

Add correction factor of + 7.

Store corrected value in memory.

Figure 3.3*b* shows an IF-THEN-ELSE example of the decision operation. This structure is used to direct operation to one of two different actions based on some condition. An example is

```
IF temperature less than 70 degrees THEN
   Turn on heater
ELSE
   Turn off heater
```

The example says that if the temperature is below the thermostat setting, we want to turn the heater on. If the temperature is equal to or above the thermostat setting, we want to turn the heater off.

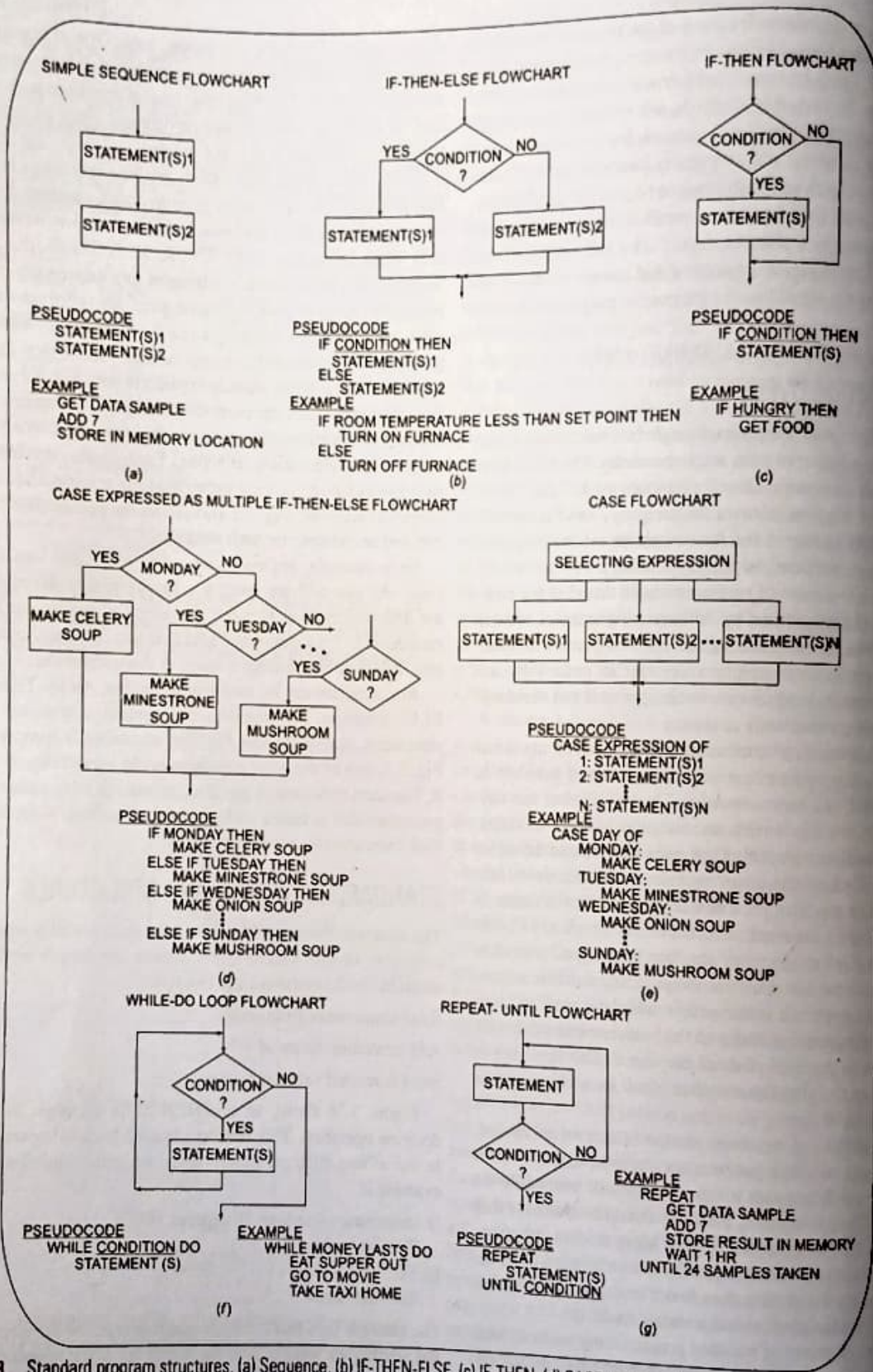

Fig. 3.3 Standard program structures. (a) Sequence, (b) IF-THEN-ELSE, (c) IF-THEN, (d) CASE expressed as nested IF-THEN-ELSE, (e) CASE, (f) WHILE-DO, (g) REPEAT-UNTIL

The IF-THEN structure shown in Fig. 3.3*c* is the same as the IF-THEN-ELSE except that one of the paths contains no action. An example of this is

```
IF hungry THEN
   Get food
```

The assumption for this example is that if you are not hungry, you will just continue on with your next task.

To represent a situation in which you want to select one of several actions based on some condition, you can use a nested IF-THEN-ELSE structure such as that shown in Fig. 3.3*d*. This everyday example describes the thinking a soup cook might go through. Note that in this example the last IF-THEN has no ELSE after it because all the possible days have been checked. You can, if you want, add the final ELSE to the IF-THEN- ELSE chain to send an error message if the data does not match any of the choices.

The CASE structure shown in Fig. 3.3*e* is really just a compact way to represent a complex IF-THEN- ELSE structure. The choice of action is determined by testing some quantity. The cook or the computer checks the value of the variable called "day" and selects the appropriate actions for that day. Each of the indicated actions, such as "Make celery soup," is itself a sequence of actions which could be represented by the structures we have described. Note that the CASE structure does not contain the final ELSE for an error.

The CASE form is more compact for documentation purposes, and some high-level languages such as Pascal allow you to implement it directly. However, the nested IF-THEN-ELSE structure gives you a much better idea of how you write an assembly language program section to choose between several alternative actions.

The WHILE-DO structure in Fig. 3.3*f* is one form of repetition. It is used to indicate that you want to do some action or sequence of actions as long as some condition is present. This structure represents a *program loop*. The example in Fig. 3.3*f* is

```
WHILE money lasts DO
   Eat supper out.
   Go to movie.
   Take a taxi home.
```

This example shows a sequence of actions you might do each evening until you ran out of money. Note that in this structure, the condition is checked *before* the action is done the first time. You certainly want to check how much money you have before eating out.

Another useful repetition structure is the REPEAT- UNTIL structure shown in Fig. 3.3*g*. You use this structure to indicate that you want the program to repeat some action or series of actions until some condition is present. A good example of the use of this structure is the programming problem we used in the discussion of flowcharts. The example is

```
REPEAT
   Get data sample from sensor.
   Add correction of + 7.
   Store result in a memory location.
   Wait 1 hour.
UNTIL 24 samples taken.
```

Note that in a REPEAT-UNTIL structure, the action(s) is done once before the condition is checked. If you want the condition to be checked before any action is done, then you can write the algorithm with a WHILE-DO structure as follows:

```
WHILE NOT 24 samples DO
   Read data sample from temperature sensor.
   Add correction factor of + 7.
   Store result in memory location.
   Wait 1 hour.
```

Remember, a REPEAT-UNTIL structure indicates that the condition is first checked *after* the statement(s) is performed, so the action or series of actions will always be done at least once. If you don't want this to happen, then use the WHILE-DO, which indicates that the condition is checked *before* any action is taken. As we will show later, the structure you use makes a difference in the actual assembly language program you write to implement it.

The WHILE-DO and REPEAT-UNTIL structures contain a simple IF-THEN-ELSE decision operation. However, since this decision is an implied part of these two structures, we don't indicate the decision separately in them.

Another form of the repetition operation that you might see in high-level language programs is the FOR-DO loop. This structure has the form

```
FOR count = 1 TO n DO
   statement
   statement
```

This FOR DO loop, as it is often called, simply repeats the sequence of actions n times, so for assembly language algorithms we usually implement this type of operation with a REPEAT-UNTIL structure.

Incidentally, if you compare the space required by the pseudocode representation for a program structure with the space required by the flowchart representation for the same structure, the space advantage of pseudocode should be obvious.

Throughout the rest of this book, we show you how to use these structures to represent program actions and how to implement these structures in assembly language.

SUMMARY OF PROGRAM STRUCTURE REPRESENTATION FORMS

Writing a successful program does not consist of just writing down a series of instructions. You must first think carefully about what you want the program to do and how you want the program to do it. Then you must represent the structure of the program in some way that is very clear both to you and to anyone else who might have to work on the program.

One way of representing program operations is with flowcharts. Flowcharts are a very graphic representation, and they are useful for short program segments, especially those that deal directly with hardware. However, flowcharts use a great deal of space. Consequently, the flowchart for even a moderately complex program may take up several pages. It often becomes difficult to follow program flow back and forth between pages. Also, since there are no agreed-upon structures, a poor programmer can write a flowchart which jumps all over the place and is even more difficult to follow. The term "logical spaghetti" comes to mind here.

A second way of representing the operations you want in a program is with a top-down design approach and standard program structures. The overall program problem is first broken down into major functional modules. Each of these modules is broken down into smaller and smaller modules until the steps in each module are obvious. The algorithms for the whole program and for each module are expressed with a standard structure. Only three basic structures. SEQUENCE, IF-THEN-ELSE, and WHILE-DO, are needed to represent any needed program action or series of actions. However, other useful structures such as IF-THEN, REPEAT-UNTIL, FOR-DO, and CASE can be derived from these basic three. A structure can contain another structure of the same type or one of the other types. Each structure has only one entry point and one exit point. These programming structures may seem restrictive, but using them usually results in algorithms which are easy to follow. Also, as we will show you soon, if you write the algorithm for a program carefully with these standard structures, it is relatively easy to translate the algorithm to the equivalent assembly language instructions.

Finding the Right Instruction

After you get the structure of a program worked out and written down, the next step is to determine the instruction statements required to do each part of the program. Since the examples in this book are based on the 8086 family of microprocessors, now is a good time to give you an overview of the instructions the 8086 has for you to use. First, however, is a hint about how to approach these instructions.

You do not usually learn a new language by memorizing an entire dictionary of the language. A better way is to learn a few useful words and practice putting these words together in simple sentences. You can then learn more words as you need them to express more complex thoughts. Likewise, you should not try to memorize all the instructions for a microprocessor at once.

For future reference, Chapter 6 contains a dictionary of all the 8086 instructions with detailed descriptions and examples of each. As an introduction, however, the few pages here contain a list of all the 8086 instructions with a short explanation of each. Skim through the list and pick out a dozen or so instructions that seem useful and understandable. As a start, look for move, input, output, logical, and arithmetic instructions. Then look through the list again to see if you can find the instructions that you might use to do the "read temperature" sensor value from a port, add + 7, and store result in memory" example program.

You can use Chapter 6 as a reference as you write programs. Here we simply list the 8086 instructions in *functional* groups with single-sentence descriptions so that you can see the types of instructions that are available to you. As you read through this section, do not expect to understand all the instructions. When you start writing programs you will probably use this section to determine the type of instruction and Chapter 6 to get the instruction details as you need them. After you have written a few programs, you will remember most of the basic instruction types and will be able to simply look up an instruction in Chapter 6 to get any additional details you need. Chapter 4 shows you in detail how to use the move, arithmetic, logical, jump, and string instructions. Chapter 5 shows how to use the call instructions and the stack.

DATA TRANSFER INSTRUCTIONS

General-purpose byte or word transfer instructions:

MNEMONIC	DESCRIPTION
MOV	Copy byte or word from specified source to specified destination.
PUSH	Copy specified word to top of stack.
POP	Copy word from top of stack to specified location.
PUSHA	(80186/80188 only) Copy all registers to stack.
POPA	(80186/80188 only) Copy words from stack to all registers.
XCHG	Exchange bytes or exchange words.
XLAT	Translate a byte in AL using a table in memory.

Simple input and output port transfer instructions:

IN	Copy a byte or word from specified port to accumulator.
OUT	Copy a byte or word from accumulator to specified port.

Special address transfer instructions:

LEA	Load effective address of operand into specified register.
LDS	Load DS register and other specified register from memory.
LES	Load ES register and other specified register from memory.

Flag transfer instructions:

LAHF	Load (copy to) AH with the low byte of the flag register.
SAHF	Store (copy) AH register to low byte of flag register.
PUSHF	Copy flag register to top of stack.
POPF	Copy word at top of stack to flag register.

ARITHMETIC INSTRUCTIONS

Addition instructions:

ADD	Add specified byte to byte or specified word to word.
ADC	Add byte + byte + carry flag or word + word + carry flag.
INC	Increment specified byte or specified word by 1.
AAA	ASCII adjust after addition.
DAA	Decimal (BCD) adjust after addition.

Subtraction instructions:

SUB	Subtract byte from byte or word from word.
SBB	Subtract byte and carry flag from byte or word and carry flag from word.
DEC	Decrement specified byte or specified word by 1.
NEG	Negate — invert each bit of a specified byte or word and add 1 (form 2's complement).
CMP	Compare two specified bytes or two specified words.
AAS	ASCII adjust after subtraction.
DAS	Decimal (BCD) adjust after subtraction.

Multiplication instructions:

MUL	Multiply unsigned byte by byte or unsigned word by word.
IMUL	Multiply signed byte by byte or signed word by word.
AAM	ASCII adjust after multiplication.

Division instructions:

DIV	Divide unsigned word by byte or unsigned double word by word.
IDIV	Divide signed word by byte or signed double word by word.
AAD	ASCII adjust before division.
CBW	Fill upper byte of word with copies of sign bit of lower byte.
CWD	Fill upper word of double word with sign bit of lower word.

BIT MANIPULATION INSTRUCTIONS

Logical instructions:

NOT	Invert each bit of a byte or word.
AND	AND each bit in a byte or word with the corresponding bit in another byte or word.
OR	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR	Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
TEST	AND operands to update flags, but don't change operands.

Shift instructions:

SHL/SAL	Shift bits of word or byte left, put zero(s) in LSB(s).
SHR	Shift bits of word or byte right, put zero(s) in MSB(s).
SAR	Shift bits of word or byte right, copy old MSB into new MSB.

Rotate instructions:

ROL	Rotate bits of byte or word left, MSB to LSB and to CF.
ROR	Rotate bits of byte or word right, LSB to MSB and to CF.
RCL	Rotate bits of byte or word left, MSB to CF and CF to LSB.
RCR	Rotate bits of byte or word right, LSB to CF and CF to MSB.

STRING INSTRUCTIONS

A *string* is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes. In the list, a "/" is used to separate different mnemonics for the same instruction. Use the mnemonic which most clearly describes the function of the instruction in a specific application. A "B" in a mnemonic is used to specifically indicate that a string of bytes is to be acted upon. A "W" in the mnemonic is used to indicate that a string of words is to be acted upon.

REP	An instruction prefix. Repeat following instruction until CX = 0.
REPE/REPZ	An instruction prefix. Repeat instruction until CX = 0 or zero flag ZF 1.
REPNE/REPNZ	An instruction prefix. Repeat until CX = 0 or ZF = 1.
MOVS/MOVSB/MOVSW	Move byte or word from one string to another.
COMPS/COMPSB/COMPSW	Compare two string bytes or two string words.
INS/INSB/INSW	(80186/80188) Input string byte or word from port.
OUTS/OUTSB/OUTSW	(80186/80188) Output string byte or word to port.
SCAS/SCASB/SCASW	Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX.

LODS/LODSB/LODSW	Load string byte into AL or string word into AX.
STOS/STOSB/STOSW	Store byte from AL or word from AX into string.

PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence.

Unconditional transfer instructions:

CALL	Call a procedure (subprogram), save return address on stack.
RET	Return from procedure to calling program.
JMP	Go to specified address to get next instruction.

Conditional transfer instructions:

A "/" is used to separate two mnemonics which represent the same instruction. Use the mnemonic which most clearly describes the decision condition in a specific program. These instructions are often used after a compare instruction. The terms *below* and *above* refer to unsigned binary numbers. *Above* means larger in magnitude. The terms *greater than* or *less than* refer to signed binary numbers. *Greater than* means more positive.

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

Iteration control instructions:

These instructions can be used to execute a series of instructions some number of times. Here mnemonics separated by a "/" represent the same instruction. Use the one that best fits the specific application.

LOOP	Loop through a sequence of instructions until CX = 0.
LOOPE/LOOPZ	Loop through a sequence of instructions while ZF = 1 and CX ≠ 0.
LOOPNE/LOOPNZ	Loop through a sequence of instructions while ZF = 0 and CX ≠ 0.
JCXZ	Jump to specified address if CX = 0.

If you aren't tired of instructions, continue skimming through the rest of the list. Don't worry if the explanation is not clear to you because we will explain these instructions in detail in later chapters.

Interrupt instructions:

INT	Interrupt program execution, call service procedure.
INTO	Interrupt program execution if OF = 1.
IRET	Return from interrupt service procedure to main program.

High-level language interface instructions:

ENTER	(80186/80188 only) Enter procedure.
LEAVE	(80186/80188 only) Leave procedure.
BOUND	(80186/80188 only) Check if effective address within specified array bounds.

PROCESSOR CONTROL INSTRUCTIONS

Flag set/clear instructions:

STC	Set carry flag CF to 1.
CLC	Clear carry flag CF to 0.
CMC	Complement the state of the carry flag CF.
STD	Set direction flag DF to 1 (decrement string pointers).
CLD	Clear direction flag DF to 0.
STI	Set interrupt enable flag to 1 (enable INTR input).
CLI	Clear interrupt enable flag to 0 (disable INTR input).

External hardware synchronization instructions:

HLT	Halt (do nothing) until interrupt or reset.
WAIT	Wait (do nothing) until signal on the TEST pin is low.
ESC	Escape to external coprocessor such as 8087 or 8089.
LOCK	An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.

No operation instruction:

NOP — No action except fetch and decode.

Now that you have skimmed through an overview of the 8086 instruction set, let's see whether you found the instructions needed to implement the "read sensor, add + 7, and store result in memory" example program. The IN instruction can be used to read the temperature value from an A/D converter connected to a port. The ADD instruction can be used to add the correction factor of +7 to the value read in. Finally, the MOV instruction can be used to copy the result of the addition to a memory location. A major point here is that breaking down the programming problem into a sequence of steps makes it easy to find the instruction or small group of instructions that will perform each step. The next section shows you how to write the actual program using the 8086 instructions.

Writing a Program

INITIALIZATION INSTRUCTIONS

After finding the instructions you need to do the main part of your program, there are a few additional instructions that you need to determine before you actually write your program. The purpose of these additional instructions is to *initialize* various parts of the system, such as segment registers, flags, and programmable port devices. Segment registers, for example, must be loaded with the upper 16 bits of the address in memory where you want the segment to begin. For our "read temperature sensor, add + 7, and store result in memory" example program, the only part we need to initialize is the data segment register. The data segment register must be initialized so that we can copy the result of the addition to a location in memory. If, for example, we want to store data in memory starting at address 00100H, then we want the data segment register to contain the upper 16 bits of this address, 0010H. The 8086 does not have an instruction to move a number directly into a segment register. Therefore, we move the desired number into one of the 16-bit general-purpose registers, then copy it to the desired segment register. Two MOV instructions will do this.

If you are using the stack in your program, then you must include instructions to load the stack segment register and an instruction to load the stack pointer register with the offset of the top of the stack. Most microcomputer systems contain several programmable peripheral devices, such as ports, timers, and controllers. You must include instructions which send control words to these devices to tell them the function you want them to perform. Also, you usually want to include instructions which set or clear the control flags, such as the interrupt enable flag and the direction flag.

The best way to approach the initialization task is to make a checklist of all the registers, programmable devices, and flags in the system you are working on. Then you can mark the ones you need for a specific program and determine the instructions needed to initialize each part. An initialization list for an 8086-based system, such as the SDK-86 prototyping board, might look like the following.

INITIALIZATION LIST

Data segment register DS
Stack segment register SS
Extra segment register ES
Stack pointer register SP
8255 programmable parallel port
8259A priority interrupt controller
8254 programmable counter
8251A programmable serial port
Initialize data variables
Set interrupt enable flag

As you can see, the list can become quite lengthy even though we have not included all the devices a system might commonly have. Note that initializing the code segment register CS is absent from this list. The code segment register is loaded with the correct starting value by the system command you use to run the program. Now let's see how you put all these parts together to make a program.

A STANDARD PROGRAM FORMAT

In this section we show you how to format your programs if you are going to construct the machine codes for each instruction by *hand.* A later section of this chapter will show you the additional parts you need to add to the program if you are going to use a computer program called an *assembler* to produce the binary codes for the instructions.

To help you write your programs in the correct format, *assembly language coding sheets* such as that shown in Fig. 3.4 are available. The ADDRESS column is used for the address or the offset of a code byte or data byte. The actual code bytes or data bytes are put in the DATA/CODE column. A *label* is a name which represents an address referred to in a jump or call instruction; labels are put in the LABELS column. A label is followed by a colon (:) if it is used by a jump or call instruction in the same code segment. The MNEM column contains the opcode mnemonics for the instructions. The OPERAND(S) column contains the registers, memory locations, or data acted upon by the instructions. A COMMENTS column gives you space to describe the function of the instruction for future reference.

Figure 3.4, shows how instructions for the "read temperature, add +7, store result in memory" program can be written in sequence on a coding sheet. We will discuss here the operation of these instructions to the extent needed. If you want more information, detailed descriptions of the *syntax* (assembly language grammar) and operation of each of these instructions can be found in Chapter 6.

PROGRAMMER: D.V. HALL — SHEET 1 OF 1

PROGRAM TITLE: READ TEMPERATURE AND CORRECT — DATE: 1/1/XX

ABSTRACT: This program reads in a temperature value from a sensor connected to port 05h, adds a correction factor of +7 to the value read in, and then stores the result in a reserved memory location.

PROCEDURES: None called.

REGISTERS USED: AX

FLAGS AFFECTED: All conditional

PORTS: uses 05 as input port

MEMORY: 00100H - DATA: 00200H-0020CH, CODE

ADDRESS	DATA or CODE	LABELS	MNEM.	OPERAND(S)	COMMENTS
00100	XX				Reserve memory location to
00101					store result. This location
00102					will be loaded with a data byte
00103					as read in & corrected by the program.
00104					XX means "don't care" about
00105					contents of location.
00106					
00107					
00108					
00109					
0010A					
0010B					
0010C					
0010D					
0010E					
00109					Code starts here
200	B8		MOV	AX, 0010H	Note break in address
01	10				Initialize DS to point to start of
02	00				memory set aside for storing data
03	8E		MOV	DS, AX	
04	D8				
05	E4		IN	AL, 05H	Read temperature from
06	05				port 05H
07	04		ADD	AL, 07H	Add correction factor of +07
08	07				
09	A2		MOV	[0000], AL	Store result in reserved
0A	00				memory
0B	00				
0C	EE		INT	3	Stop, wait for command
0D					from user
0E					
0F					

Fig. 3.4 Assembly language program on standard coding form.

The first line at the top of the coding form in Fig. 3.4 does not represent an instruction. It simply indicates that we want to set aside a memory location to store the result. This location must be in available RAM so that we can write to it. Address 00100H is an available RAM location on an SDK-86 prototyping board, so we chose it for this example. Next, we decide where in memory we want to start putting the code bytes for the instructions of the program. Again, on an SDK-86 prototyping board, address 00200H and above is available RAM, so we chose to start the program at address 00200H.

The first operation we want to do in the program is to initialize the data segment register. As discussed previously, two MOV instructions are used to do this. The MOV AX, 0010H instruction, when executed, will load the upper 16 bits of the address we chose for data storage into the AX register. The MOV DS, AX instruction will copy this number from the AX register to the data segment register. Now we get to the instructions that do the input, add, and store operations. The IN AL, 05H instruction will copy a data byte from the port 05H to the AL register. The ADD AL, 07 instruction will add 07H to the AL register and leave the result in the AL register. The MOV [0000], AL instruction will copy the byte in AL to a memory location at a displacement of 0000H from the data segment base. In other words, AL will be copied to a physical address computed by adding 0000 to the segment base address represented by the 0010H in the DS register. The result of this addition is a physical address of 00100H, so the result in AL will be copied to physical address 00100H in memory. This is an example of the direct addressing mode described near the end of the previous chapter.

The INT 3 instruction at the end of the program functions as a *breakpoint*. When the 8086 on an SDK-86 board executes this instruction, it will cause the 8086 to stop executing the instructions of your program and return control to the *monitor* or *system program*. You can then use *system commands* to look at the contents of registers and memory locations, or you can run another program. Without an instruction such as this at the end of the program, the 8086 would fetch and execute the code bytes for your program, then go on fetching meaningless bytes from memory and trying to execute them as if they were code bytes.

The next major section of this chapter will show you how to construct the binary codes for these and other 8086 instructions so that you can assemble and run the programs on a development board such as the SDK-86. First, however, we want to use Fig. 3.4 to make an important point about writing assembly language programs.

DOCUMENTATION

In a previous section of this chapter, we stressed the point that you should do a lot of thinking and carefully write down the algorithm for a program before you start writing instruction statements. You should also document the program itself so that its operation is clear to you and to anyone else who needs to understand it.

Each page of the program should contain the name of the program, the page number, the name of the programmer, and perhaps a version number. Each program or procedure should have a heading block containing an *abstract* describing what the program is supposed to do, which procedures it calls, which registers it uses, which ports it uses, which flags it affects, the memory used, and any other information which will make it easier for another programmer to interface with the program.

Comments should be used generously to describe the specific *function* of an instruction or group of instructions in this particular program. Comments should not be just an expansion of the instruction mnemonic. A comment of ";add 7 to AL" after the instruction ADD AL, 07H, for example, would not tell you much about the function of the instruction in a particular program. A more enlightening comment might be ";Add altitude correction factor to temperature." Incidentally, not every statement needs an individual comment. It is often more useful to write a comment which explains the function of a group of instructions.

We cannot overemphasize the importance of clear, concise documentation in your programs. Experience has shown that even a short program you wrote without comments a month ago may not be at all understandable to you now.

CONSTRUCTING THE MACHINE CODES FOR 8086 INSTRUCTIONS

This section shows you how to construct the binary codes for 8086 instructions. Most of the time you will probably use an assembler program to do this for you, but it is useful to understand how the codes are constructed. If you have an 8086-based prototyping board such as the Intel SDK-86 available, knowing how to hand code instructions will enable you to code, enter, and run simple programs.

Instruction Templates

To code the instructions for 8-bit processors such as the 8085, all you have to do is look up the hexadecimal code for each instruction on a one-page chart. For the 8086, the process is not quite as simple. Here's why. There are 32 ways to specify the source of the operand in an instruction such as MOV CX, source. The source of the operand can be any one of eight 16-bit registers, or a memory location specified by any one of 24 memory addressing modes. Each of the 32 possible instructions requires a different binary code. If CX is made the source rather than the destination, then there are 32 ways of specifying the destination. Each of these 32 possible instructions requires a different binary code. There are thus 64 different codes for MOV instructions using CX as a source or as a destination. Likewise, another 64 codes are required to specify all the possible MOVs using CL as a

source or a destination, and 64 more are required to specify all the possible MOVs using CH as a source or a destination. The point here is that, because there is such a large number of possible codes for the 8086 instructions, it is impractical to list them all in a simple table. Instead, we use a *template* for each basic instruction type and fill in bits within this template to indicate the desired addressing mode, data type, etc. In other words, we build up the instruction codes on a bit-by-bit basis.

Different Intel literature shows two slightly different formats for coding 8086 instructions. One format is shown at the end of the 8086 data sheet in Appendix A. The second format is shown along with the 8086 instruction timings in Appendix B. We will start by showing you how to use the templates shown in the 8086 data sheet.

As a first example of how to use these templates, we will build the code for the IN AL, 05H instruction from our example program. To start, look at the template for this instruction in Fig. 3.5*a*. Note that two bytes are required for the instruction. The upper 7 bits of the first byte tell the 8086 that this is an "input from a fixed port" instruction. The bit labeled "W" in the template is used to tell the 8086 whether it should input a byte to AL or a word to AX. If you want the 8086 to input a byte from an 8-bit port to AL, then make the W bit a 0. If you want the 8086 to input a word from a 16-bit port to the AX register, then make the W bit a 1. The 8-bit port address, 05H or 00000101 binary, is put in the second byte of the instruction. When the program is loaded into memory to be run, the first instruction byte will be put in one memory location, and the second instruction byte will be put in the next. Fig. 3.5*c* shows this in hexadecimal form as E4H, 05H.

To further illustrate how these templates are used, we will show here several examples with the simple MOV instruction. We will then show you how to construct the rest of the codes for the example program in Fig. 3.4. Other examples will be shown as needed in the following chapters.

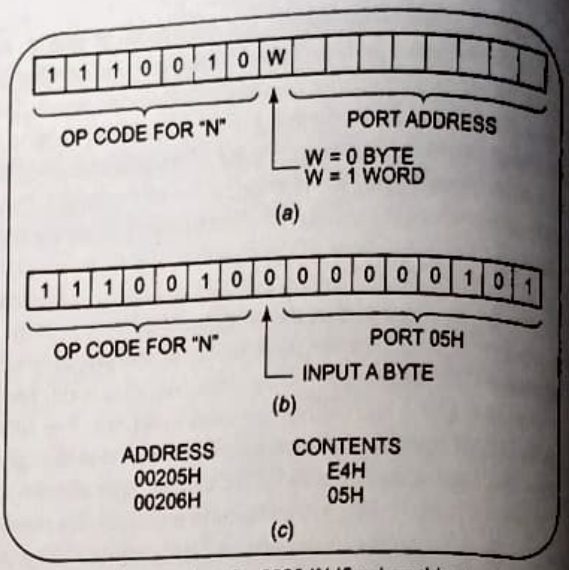

Fig. 3.5 Coding template for 8086 IN (fixed port) instruction. (a) Template. (b) Example1 (c) Hex codes in sequential memory locations.

MOV Instruction Coding Format and Examples

FORMAT

Figure 3.6 shows the coding template or format for 8086 instructions which MOV data from a register to a register, from a register to a memory location, or from a memory location to a register. Note that at least two code bytes are required for the instruction.

The upper 6 bits of the first byte are an opcode which indicates the general type of instruction. Look in the table in Appendix A to find the 6-bit opcode for this MOV register/memory to/from register instruction. You should find it to be 100010.

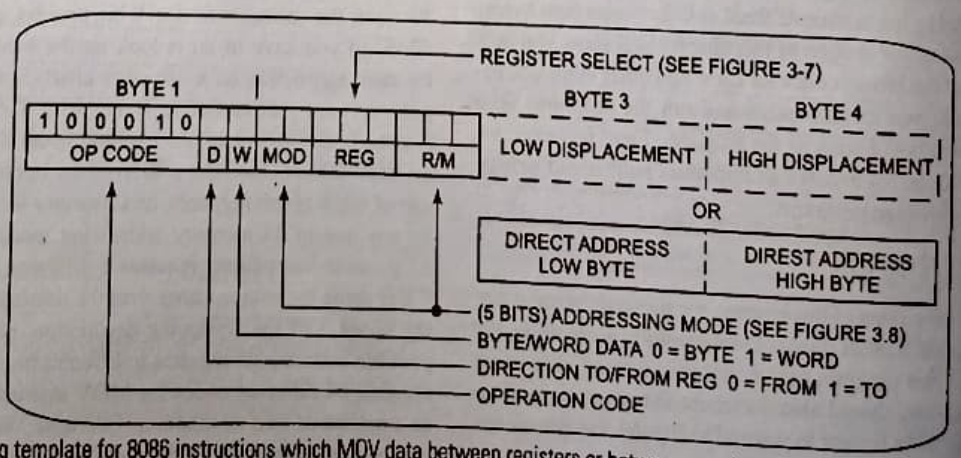

Fig. 3.6 Coding template for 8086 instructions which MOV data between registers or between a register and a memory location.

The W bit in the first word is used to indicate whether a byte or a word is being moved. If you are moving a byte, make W = 0. If you are moving a word, make W = 1.

In this instruction, one operand must always be a register, so 3 bits in the second byte are used to indicate which register is involved. The 3-bit codes for each register are shown in the table at the end of Appendix A and in Fig. 3.7. Look in one of these places to find the code for the CL register. You should get 001.

The D bit in the first byte of the instruction code is used to indicate whether the data is being moved to the register identified in the REG field of the second byte or *from* that register. If the instruction is moving data to the register identified in the REG field, make D = 1. If the instruction is moving data *from* that register, make D = 0.

Now remember that in a MOV instruction, one operand must be a register and the other operand may be a register or a memory location. The 2-bit field labeled MOD and the 3-bit field labeled R/M in the second byte of the instruction code are used to specify the desired addressing mode for the other operand. Fig. 3.8 shows the MOD and R/M bit patterns for each of the 32 possible addressing modes. Here's an overview of how you use this table.

1. If the other operand in the instruction is also one of the eight registers, then put in 11 for the MOD bits in the instruction code. In the R/M bit positions in the instruction code, put the 3-bit code for the other register.
2. If the other operand is a memory location, there are 24 ways of specifying how the execution unit should compute the effective address of the operand in memory. The effective address can be specified directly in the instruction, it can be contained in a register, or it can be the sum of one or two registers and a displacement. The MOD bits are used to indicate whether the address specification in the instruction contains a displacement. The R/M code indicates which register(s) contain part(s) of the effective address. Here's how it works:

   If the specified effective address contains no displacement, as in the instruction MOV CX, [BX] or in the instruction MOV [BX][SI], DX, then make the MOD bits 00 and choose the R/M bits which correspond to the register(s) containing the effective address. For example, if an instruction contains just [BX], the 3-bit R/M code is 111. For an instruction which contains [BX[SI]. the R/M code is 000. Note that for direct addressing, where the displacement of

REGISTER		CODE
	W=1	W=0
AL	AX	000
BL	BX	011
CL	CX	001
DL	DX	010
AH	SP	100
BH	DI	111
CH	BP	101
DH	SI	110
	SEGREG	CODE
	CS	01
	DS	11
	ES	00
	SS	10

Fig. 3.7 Instruction codes for 8086 registers.

MOD / RM	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	d16 (direct address)	[BP] + d8	[BP] + d16	DH	SI
111	[BX]	[BX] + d8	[BX] + d16	BH	DI
		MEMORY MODE		REGISTER MODE	

d8 = 8-bit displacement d16 = 16-bit displacement

Fig. 3.8 MOD and R/M bit patterns for 8086 instructions. The effective address (EA) produced by these addressing modes will be added to the data segment base to form the physical address, except for those cases where BP is used as part of the EA. In that case the EA will be added to the stack segment base to form the physical address. You can use a segment-override prefix to indicate that you want the EA to be added to some other segment base.

the operand from the segment base is specified directly in the instruction, MOD is 00 and R/M is 110. For an instruction using direct addressing, the low byte of the direct address is put in as a third instruction code byte of the instruction, and the high byte of the direct address is put in as a fourth instruction code byte.

3. If the effective address specified in the instruction contains a displacement less than 256 along with a reference to the contents of a register, as in the instruction MOV CX, 43H[BX], then code in MOD as 01 and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV CX, 43H[BX], MOD will be 01 and R/M will be 111. Put the 8-bit value of the displacement in as the third byte of the instruction.
4. If the expression for the effective address contains a displacement which is too large to fit in 8 bits, as in the instruction MOV DX, 4527H[BX], then put in 10 for MOD and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV DX, 4527H[BX], the R/M bits are 111. The low byte of the displacement is put in as a third byte of the instruction. The high byte of the displacement is put in as a fourth byte of the instruction. The examples which follow should help clarify all this for you.

MOV Instruction Coding Examples

All the examples in this section use the MOV instruction template in Fig. 3.6. As you read through these examples, it is a good idea to keep track of the bit-by- bit development on a separate piece of paper for practice.

CODING MOV SP, BX

This instruction will copy a word from the BX register to the SP register. Consulting the table in Appendix A, you find that the 6-bit opcode for this instruction is 100010. Because you are moving a word, W = 1. The D bit for this instruction may be somewhat confusing, however. Since two registers are involved, you can think of the move as either to SP or *from* BX. It actually does not matter which you assume as long as you are consistent in coding the rest of the instruction. If you think of the instruction as moving a word *to* SP, then make D = 1 and put 100 in the REG field to represent the SP register. The MOD field will be 11 to represent register addressing mode. Make the R/M field 011 to represent the other register, BX. The resultant code for the instruction MOV SP, BX will be 10001011 11100011. Fig. 3.9a shows the meaning of all these bits.

If you change the D bit to a 0 and swap the codes in the REG and R/M fields, you will get 10001001 11011100, which is another equally valid code for the instruction. Fig. 3.9b shows the meaning of the bits in this form. This second form, incidentally, is the form that the Intel 8086 Macroassembler produces.

CODING MOV CL, [BX]

This instruction will copy a byte to CL from the memory location whose effective address is contained in BX. The effective address will be added to the data segment base in DS to produce the physical address.

To find the 6-bit opcode for byte 1 of the instruction, consult the table in Appendix A. You should find that this code is 100010. Make D = 1 because data is being moved to register CL. Make W = 0 because the instruction is moving a byte into CL. Next you need to put the 3-bit code which represents register CL in the REG field of the second byte of the instruction code. The codes for each register are shown in Fig. 3.7. In this figure you should find that the code for CL is 001. Now, all you need to determine is the bit patterns for the MOD and R/M fields. Again use the table in Fig. 3.8 to do this. In the table, first find the box containing the desired addressing mode. The box containing [BX], for example, is in the lower left corner of the table. Read the required MOD-bit pattern from the top of the column. In this case, MOD is 00. Then read the required R/M- bit pattern at the left of the box. For this instruction you should find R/M to be 111. Assembling all these bits together should give you 10001010 00001111 as the binary code for the instruction MOV CL, [BX). Fig. 3.10 summarizes the meaning of all the bits in this result.

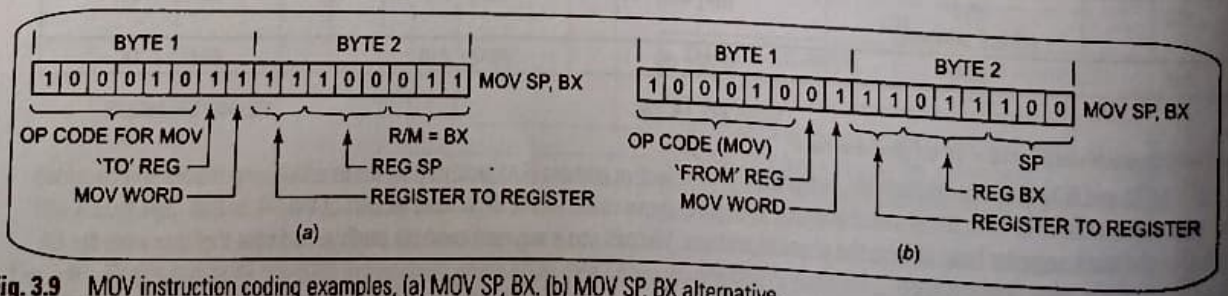

Fig. 3.9 MOV instruction coding examples, (a) MOV SP, BX. (b) MOV SP, BX alternative.

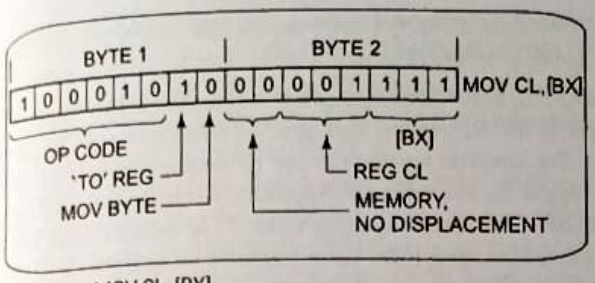

Fig. 3.10 MOV CL, [BX].

CODING MOV 43H[SI], DH

This instruction will copy a byte from the DH register to a memory location. The BIU will compute the effective address of the memory location by adding the indicated displacement of 43H to the contents of the SI register. As we showed you in the last chapter, the BIU then produces the actual physical address by adding this effective address to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. Put 110 in the REG field to represent the DH register. D = 0 because you are moving data *from* the DH register. W = 0 because you are moving a byte. The R/M field will be 100 because SI contains part of the effective address. The MOD field will be 01 because the displacement contained in the instruction, 43H, will fit in 1 byte. If the specified displacement had been a number larger than FFH, then MOD would be 10. Putting all these pieces together gives 10001000 01110100 for the first two bytes of the instruction code. The specified displacement, 43H or 01000011 binary, is put after these two as a third instruction byte. Fig. 3.11 shows this. If an instruction specifies a 16-bit displacement, then the low byte of the displacement is put in as byte 3 of the instruction code, and the high byte of the displacement is put in as byte 4 of the instruction code.

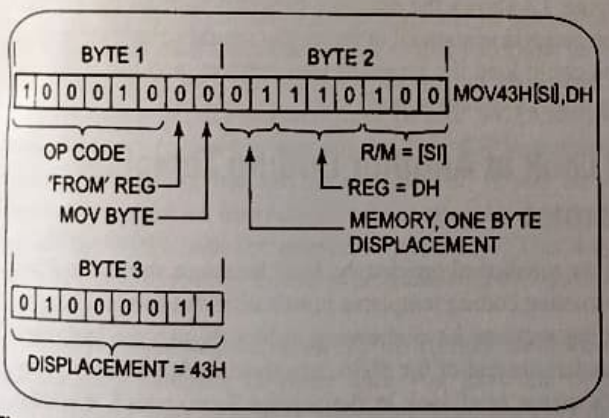

Fig. 3.11 MOV 43H[SI], DH.

CODING MOV CX, [437AH]

This instruction copies the contents of two memory locations into the CX register. The direct address or displacement of the first memory location from the start of the data segment is 437AH. As we showed you in the last chapter, the BIU will produce the physical memory address by adding this displacement to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. Make D = 1 because you are moving data to the CX register, and make W = 1 because the data being moved is a word. Put 001 in the REG field to represent the CX register, then consult Fig. 3.8 to find the MOD and R/M codes. In the first column of the figure, you should find a box labeled "direct address," which is the name given to the addressing mode used in this instruction. For direct addressing, you should find MOD to be 00 and R/M to be 110. The first two code bytes for the instruction, then, are 10001011 00001110. These two bytes will be followed by the low byte of the direct address, 7AH (01111010 binary), and the high byte of the direct address, 43H (01000011 binary). The instruction will be coded into four successive memory addresses as 8BH, 0EH, 7AH, and 43H. Fig. 3.12 spells this out in detail.

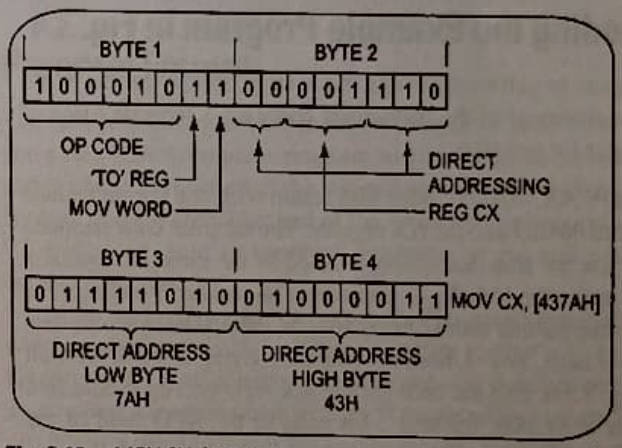

Fig. 3.12 MOV CX, [437AH].

CODING MOV CS:[BX], DL

This instruction copies a byte from the DL register to a memory location. The effective address for the memory location is contained in the BX register. Normally an effective address in BX will be added to the data segment base in DS to produce the physical memory address. In this instruction, the CS: in front of [BX] indicates that we want the BIU to add the effective address to the code segment base in CS to produce the physical address. The CS: is called a *segment override prefix*.

When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put in memory *before* the code for the rest of the instruction. The code byte for the segment override prefix has the format 001XX110. You insert a 2-bit code in place of the X's to indicate which segment base you want the effective address to be added to. As shown in Fig. 3.7, the codes for these 2 bits are as follows: ES = 00, CS = 01, SS = 10, and DS = 11.

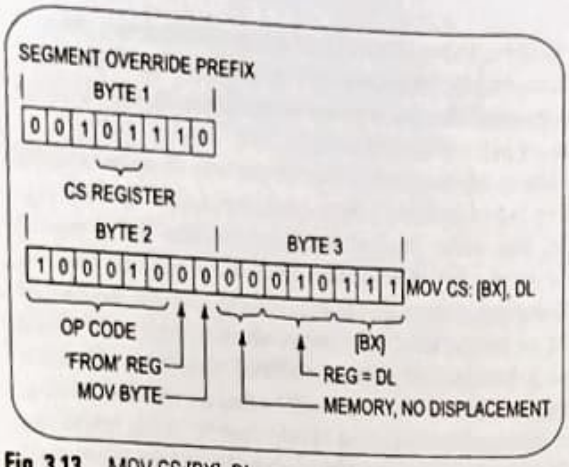

Fig. 3.13 MOV CS:[BX], DL

The segment override prefix byte for CS, then, is 00101110. For practice, code out the rest of this instruction. Fig. 3.13 shows the result you should get and how the code for the segment override prefix is put before the other code bytes for the instruction.

Coding the Example Program in Fig. 3.4

Again, as you read through this section, follow the bit- by-bit development of the instruction codes on a separate piece of paper for practice.

MOV AX, 0010H This instruction will load the immediate word 0010H into the AX register. The simplest code template to use for this instruction is listed in the table in Appendix A under the "MOV — Immediate to register" heading. The format for this instruction is 1011 W REG, data byte low, data byte high. W = 1 because you are moving a word. Consult Fig. 3.7 to find the code for the AX register. You should find this to be 000. Put this 3-bit code in the REG field of the instruction code. The completed instruction code byte is 10111000. Put the low byte of the immediate number, 10H, in as the second code byte. Then put the high byte of the immediate data, 00H, in as the third code byte. The resultant sequence of code bytes, then, will be B8H, 10H, 00H.

MOV DS, AX This instruction copies the contents of the AX register into the data segment register. The template to use for coding this instruction is found in the table in Appendix A under the heading "MOV — Register/memory to segment register." The format for this template is 10001110 MOD 0 segreg R/M. Segreg represents the 2-bit code for the desired segment register, as shown in Fig. 3.7. These codes are also found in the table at the end of Appendix A. The segreg code for the DS register is 11. Since the other operand is a register, MOD should be 11. Put the 3-bit code for the AX register, 000, in the R/M field. The resultant codes for the two code bytes should then be 10001110 11011000, or 8EH D8H.

IN AL, 05H This instruction copies a byte of data from port 05H to the AL register. The coding for this instruction was described in a previous section. The code for the instruction is 11100100 00000101 or E4H 05H.

ADD AL, 07H This instruction adds the immediate number 07H to the AL register and puts the result in the AL register. The simplest template to use for coding this instruction is found in the table in Appendix A under the heading "ADD — Immediate to accumulator." The format is 0000010 W, data byte, data byte. Since you are adding a byte, W = 0. The immediate data byte you are adding will be put in the second code byte. The third code byte will not be needed because you are adding only a byte. The resultant codes, then, are 00000100 00000111 or 04H 07H.

MOV [0000], AL This instruction copies the contents of the AL register to a memory location. The direct address or displacement of the memory location from the start of the data segment is 0000H. The code template for this instruction is found in the table in Appendix A under the heading "MOV — Accumulator to memory." The format for the instruction is 1010001 W, address low byte, address high byte. Since the instruction moves a byte, W = 0. The low byte of the direct address is written in as the second instruction code byte, and the high byte of the direct address is written in as the third instruction code byte. The codes for these 3 bytes, then, will be 10100010 00000000 00000000 or A2H 00H 00H.

INT 3 In some 8086 systems this instruction causes the 8086 to stop executing your program instructions, return to the monitor program, and wait for your next command. According to the format table in Appendix A, the code for a type 3 interrupt is the single byte 11001100 or CCH.

SUMMARY OF HAND CODING THE EXAMPLE PROGRAM

Figure 3.4 shows the example program with all the instruction codes in sequential order as you would write them so that you could load the program into memory and run it. Codes are in HEX to save space.

A Look at Another Coding Template Format

As we mentioned previously, Intel literature shows the 8086 instruction coding templates in two different forms. The preceding sections have shown you how to use the templates found at the end of the 8086 data sheet in Appendix A. Now let's take a brief look at the second form, which is shown along with the instruction clock cycles in Appendix B.

The only difference between the second form for the templates and the form we discussed previously is that the D and W bits are not individually identified. Instead, the complete opcode bytes are shown for each version of an instruction. For example, in Appendix B, the opcode byte for the MOV memory 8, register 8 instruction is shown as 88H, and the

opcode byte for the MOV memory 16, register 16 instruction is shown as 89H. If you compare these codes with those derived from Appendix A, you will see that the only difference between the two codes is the W bit. For the 8-bit move, W = 0, and for the 16-bit move, W = 1.

One important point to make about using the templates in Appendix B is that for operations involving two registers, the register identified in the REG field is not consistent from instruction to instruction. For the MOV instructions, the templates in Appendix B assume that the 3-bit code for the source register is put in the REG field of the MOD/RM instruction byte, and the 3-bit code for the destination register is put in the R/M field of the MOD/RM instruction byte. According to Appendix B, the template for a 16-bit register-to-register move is 89H followed by the MOD reg R/M byte. In this template, D = 0, so the 3-bit code for the source register will be put in the reg field. Using this template, then, the instruction MOV BX, CX is coded as 10001001 11001011 or 89H CBH.

For the ADD, ADC, SUB, SBB, AND, OR, and XOR instructions which involve two registers, the templates in Appendix B show D = 1. To be consistent with these templates, then, you have to put the 3-bit code for the destination register in the reg field in the instruction.

It really doesn't matter whether you use the templates in Appendix A or those in Appendix B, as long as you are consistent in coding each instruction.

A Few Words about Hand Coding

If you have to hand code 8086 assembly language programs, here are a few tips to make your life easier. First, check your algorithm very carefully to make sure that it really does what it is supposed to do. Second, initially write down just the assembly language statements and comments for your program. You can check the table in the appendix to determine how many bytes each instruction takes so that you know how many blank lines to leave between instruction statements. You may find it helpful to insert three or four NOP instructions after every nine or ten instructions. The NOP instruction doesn't do anything but kill time. However, if you accidentally leave out an instruction in your program, you can replace the NOPs with the needed instruction(s). This way you don't have to rewrite the entire program after the missing instruction.

After you have written down the instruction statements, recheck very carefully to make sure you have the right instructions to implement your algorithm. Then work out the binary codes for each instruction and write them in the appropriate places on the coding form.

Hand coding is laborious for long programs. When writing long programs, it is much more efficient to use an assembler. The next section of this chapter shows you how to write your programs so that you can use an assembler to produce the machine codes for the instructions.

WRITING PROGRAMS FOR USE WITH AN ASSEMBLER

If you have an 8086, assembler available, you should learn to use it as soon as possible. Besides doing the tedious task of producing the binary codes for your instruction statements, an assembler also allows you to refer to data items by name rather than by their numerical offsets. As you should soon see, this greatly reduces the work you have to do and makes your programs much more readable. In this section we show you how to write your programs so that you can use an assembler on them.

NOTE: The assembly language programs in the rest of this book were assembled with TASM 1.0 from Borland International or MASM 5.1 from Microsoft Corp. TASM is faster, but the program format for these two assemblers is essentially the same. If you are using some other assembler, check the manual for it to determine any differences in syntax from the examples in this book.

Program Format

The best way to approach this section seems to be to show you a simple, but complete, program written for an assembler and explain the function of the various parts of the program. By now you are probably tired of the "read temperature, add +7, and store result in memory" program, so we will use another example.

Figure 3.14, shows an 8086 assembly language program which multiplies two 16-bit binary numbers to give a 32-bit binary result. If you have a microcomputer development system or a microcomputer with an 8086 assembler to work on, this is a good program for you to key in, assemble, and run to become familiar with the operation of your system. (A sequence of exercises in the accompanying lab manual explains how to do this.) In any case, you can use the structure of this example program as a model for your own programs.

In addition to program instructions, the example program in Fig. 3.14 contains directions to the assembler. These directions to the assembler are commonly called *assembler directives* or *pseudo operations*. A section at the end of Chapter 6 lists and describes for your reference a large number of the available assembler directives. Here we will discuss the basic assembler directives you need to get started writing programs. We will introduce more of these directives as we need them in the next two chapters.

SEGMENT and ENDS Directives

The SEGMENT and ENDS directives are used to identify a group of data items or a group of instructions that you want to be put together in a particular segment. These directives

```
            ; 8086 PROGRAM F3-14.ASM
;ABSTRACT   : This program multiplies the two 16-bit words in the memory
            ; locations called MULTIPLICAND and MULTIPLIER. The result
            ; is stored in the memory location, PRODUCT
;REGISTERS  : Uses CS, DS, AX, DX
;PORTS      : None used

DATA_HERE   SEGMENT
            MULTIPLICAND  DW 204AH      ; First word here
            MULTIPLIER    DW 382AH      ; Second word here
            PRODUCT       DW 2 DUP(0)   ; Result of multiplication here
DATA_HERE   ENDS

CODE_HERE   SEGMENT
            ASSUME    CS:CODE_HERE, DS:DATA_HERE
START:      MOV  AX, DATA_HERE          ; Initialize DS register
            MOV  DS, AX
            MOV  AX, MULTIPLICAND       ; Get one word
            MUL  MULTIPLIER             ; Multiply by second word
            MOV  PRODUCT, AX            ; Store low word of result
            MOV  PRODUCT+2, DX          ; Store high word of result
            INT  3                      ; Wait for command from user
CODE_HERE   ENDS
            END START

;     Programs to be run using a debugger in DOS must include the START: label and the
; START after the END followed by a carriage return. Programs to be downloaded and run need
; only the END directive followed by a carriage return.
```

Fig. 3.14 Assembly language source program to multiply two 16-bit binary numbers to give a 32-bit result.

are used in the same way that parentheses are used to group like terms in algebra. A group of data statements or a group of instruction statements contained between SEGMENT and ENDS directives is called a *logical segment.* When you set up a logical segment, you give it a name of your choosing. In the example program, the statements DATA_HERE SEGMENT and DATA_HERE ENDS set up a logical segment named DATA_HERE. There is nothing sacred about the name DATA_HERE. We simply chose this name to help us remember that this logical segment contains data statements. The statements CODE_HERESEGMENT and CODE_HERE ENDS in the example program set up a logical segment named CODE_HERE which contains instruction statements. Most 8086 assemblers, incidentally, allow you to use names and labels of up to 31 characters. You can't use spaces in a name, but you can use an underscore as shown to separate words in a name. Also, you can't use instruction mnemonics as segment names or labels. Throughout the rest of the program you will refer to a logical segment by the name that you give it when you define it.

A logical segment is not usually given a physical starting address when it is declared. After the program is assembled and perhaps linked with other assembled program modules, it is then assigned the physical address where it will be loaded in memory to be run.

Naming Data and Addresses — EQU, DB, DW, and DD Directives

Programs work with three general categories of data: constants, variables, and addresses. The value of a constant does not change during the execution of the program. The number 7 is an example of a constant you might use in a program. A variable is the name given to a data item which can change during the execution of a program. The current temperature of an oven is an example of a variable. Addresses are referred to in many instructions. You may, for example, load an address into a register or jump to an address.

Constants, variables, and addresses used in your programs can be given names. This allows you to refer to them by name rather than having to remember or calculate their value each time you refer to them in an instruction. In other words, if you give names to constants, variables, and addresses, the assembler can use these names to find a desired data item or address when you refer to it in an instruction. Specific directives are used to give names to constants and variables in your programs. Labels are used to give names to addresses in your programs.

THE EQU DIRECTIVE

The EQU, or *equate,* directive is used to assign names to constants used in your programs. The statement

CORRECTION_FACTOR EQU 07H, in a program such as our previous example, would tell the assembler to insert the value 07H every time it finds the name CORRECTION_FACTOR in a program statement. In other words, when the assembler reads the statement ADD AL, CORRECTION_FACTOR, it will automatically code the instruction as if you had written it ADD AL, 07H. Here's the advantage of using an EQU directive to declare constants at the start of your program. Suppose you use the correction factor of + 07H 23 times in your program. Now the company you work for changes the brand of temperature sensor it buys, and the new correction factor is + 09H. If you used the number 07H directly in the 23 instructions which contain this correction factor, then you have to go through the entire program, find each instruction that uses the correction factor, and update the value. Murphy's law being what it is, you are likely to miss one or two of these, and the program won't work correctly. If you used an EQU at the start of your program and then referred to CORRECTION_FACTOR by name in the 23 instructions, then all you do is change the value in the EQU statement from 07H to 09H and reassemble the program. The assembler automatically inserts the new value of 09H in all 23 instructions.

DB, DW, AND DD DIRECTIVES

The DB, DW, and DD directives are used to assign names to variables in your programs. The DB directive after a name specifies that the data is of *type byte*. The program statement OVEN_TEMPERATURE DB 27H, for example, declares a variable of type byte, gives it the name OVEN_TEMPERATURE, and gives it an initial value of 27H. When the binary code for the program is loadedinto memory to be run, the value 27H will be loadedinto the memory location identified by the name OVEN_TEMPERATURE DB 27H.

As another example, the statement CONVERSION_FACTORS DB 27H, 48H, 32H, 69H will declare a data structure (array) of 4 bytes and initialize the 4 bytes with the specified 4 values. If you don't care what value a data item is initialized to, then you can indicate this with a "?," as in the statement TARE_WEIGHT DB ?

NOTE: Variables which are changed during the operation of a program should also be initialized with program instructions so that the program can be rerun from the start without reloading it to initialize the variables.

DW is used to specify that the data is of *type word* (16 bits), and DD is used to specify that the data is of *type doubleword* (32 bits). The example program in Fig. 3.14 shows three examples of naming and initializing word-type data items.

The first example, MULTIPLICAND DW 204AH, declares a data word named MULTIPLICAND and initializes that data word with the value 204AH. What this means is that the assembler will set aside two successive memory locations and assign the name MULTIPLICAND to the first location. As you will see, this allows us to access the data in these memory locations by name. The MULTIPLICAND DW 204AH statement also indicates that when the final program is loaded into memory to be run, these memory locations will be loaded with (initialized to 204AH. Actually, since this is an Intel microprocessor, the first address in memory will contain the low byte of the word, 4AH, and the second memory address will contain the high byte of the word, 20H.

The second data declaration example in Fig. 3.14, MULTIPLIER DW 3B2AH, sets aside storage for a word in memory and gives the starting address of this word the name MULTIPLIER. When the program is loaded, the first memory address will be initialized with 2AH, and the second memory location with 3BH.

The third data declaration example in Fig. 3.14, PRODUCT DW 2 DUP(0), sets aside storage for two words in memory and gives the starting address of the first word the name PRODUCT. The DUP(0) part of the statement tells the assembler to initialize the two words to all zeros. When we multiply two 16-bit binary numbers, the product can be as large as 32 bits, so we must set aside this much space to store the product. We could have used the DD directive to declare PRODUCT a doubleword, but since in the program we move the result to PRODUCT one word at a time, it is more convenient to declare PRODUCT 2 words.

Figure 3.15 shows how the data for MULTIPLICAND, MULTIPLIER, and PRODUCT will actually be arranged in memory starting from the base of the DATA_HERE segment. The first byte of MULTIPLICAND, 4AH, will be at a displacement of zero from the segment base, because MULTIPLICAND is the first data item declared in the logical segment DATA_HERE. The displacement of the second byte of MULTIPLICAND is 0001. The displacement of the first byte of MULTIPLIER from the segment base is 0002H, and the displacement of the second byte of MULTIPLIER is 0003H.

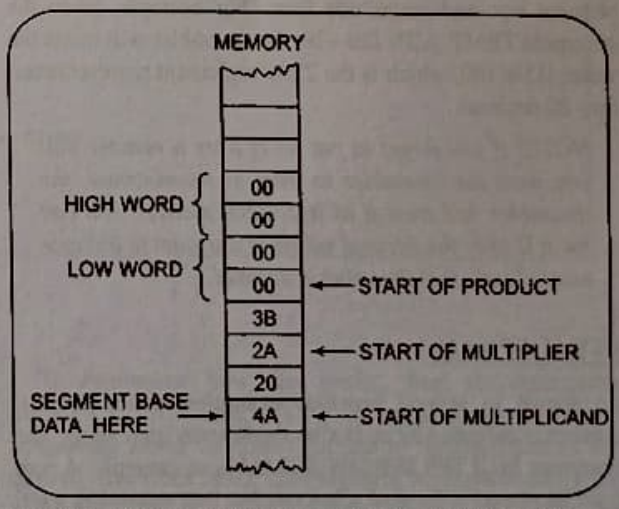

Fig. 3.15 Data arrangement in memory for multiply program.

These are the displacements that we would have to figure out for each data item if we were not using names to refer to them.

If the logical segment DATA_HERE is eventually put in ROM or EPROM, then MULTIPLICAND will function as a constant, because it cannot be changed during program execution. However, if DATA_HERE is eventually put in RAM, then MULTIPLICAND can function as a variable because a new value could be written in those memory locations during program execution.

Types of Numbers Used in Data Statements

All the previous examples of DB, DW, and DD declarations use hexadecimal numbers, as indicated by an "H" after the number. You can, however, put in a number in any one of several other forms. For each form you must tell the assembler which form you are using.

BINARY

For example, when you use a binary number in a statement, you put a "B" after the string of 1's and 0's to let the assembler know that you want the number to be treated as a binary number. The statement TEMP_ MAX DB 01111001B is an example. If you want to put in a negative binary number, write the number in its 2's complement sign-and-magnitude form.

DECIMAL

The assembler treats a number with no identifying letter after it as a decimal number. The assembler automatically converts a decimal number in a statement to binary so that the value can be loaded into memory. Given the statement TEMP_ MAX DB 49, for example, the assembler will automatically convert the 49 decimal to its binary equivalent, 00110001. If you indicate a negative number in a data declaration statement, the assembler will convert the number to its 2's complement sign-and-magnitude form. For example, given the statement TEMP_MIN DB –20, the assembler will insert the value 11101100, which is the 2's complement representation for –20 decimal.

NOTE: If you forget to put an H after a number that you want the assembler to treat as hexadecimal, the assembler will treat it as a decimal number. You can put a D after the decimal values if you want to indicate more clearly that the value is decimal.

HEXADECIMAL

As shown in several previous examples, a hexadecimal number is indicated by an H after the hexadecimal digits. The statement MULTIPLIER DW 3B2AH is an example. A zero must be placed in front of a hex number that starts with a letter; for example, the number AH must be written 0AH.

BCD

Remember from Chapter 1 that in BCD each decimal digit is represented by its 4-bit binary equivalent. The decimal number 37, for example, is represented in BCD as 00110111. As you can see, this number is equal to 37H. The only way you can tell whether the number 00110111 represents BCD 37 or hexadecimal 37 is by how it is used in the program! The point here is that if you want the assembler to initialize a variable with the value 37 BCD, you put an H after the number. The statement SECONDS DB 59H, for example, will initialize the variable SECONDS with 01011001, the BCD representation of 59.

ASCII

You can declare a data structure (array) containing a sequence of ASCII codes by enclosing the letters or numbers after a DB in single quotation marks. The statement BOY1 DB 'ALBERT', for example, tells the assembler to declare a data item named BOY1 that has six memory locations. It also tells the assembler to put the ASCII code for A in the first memory location, the ASCII code for L in the second, the ASCII code for B in the third, etc. The assembler will automatically determine the ASCII codes for the letters or numbers within the quotes. Note that this ASCII trick can be used only with the DB directive.

Accessing Named Data with Program Instructions

Now that we have shown you how a data structure can be set up, let's look at how program instructions access this data. Temporarily skipping over the first two instructions in the CODE_HERE section of the program in Fig. 3.16, find the instruction MOV AX, MULTIPLICAND. This instruction, when executed, will copy a word from the memory location named MULTIPLICAND to the AX register. Here's how this works.

When the assembler reads through this program the first time, it automatically calculates the offset of each of the named data items from the segment base DATA_ HERE. In Fig. 3.15 you can see that the displacement of MULTIPLICAND from the segment base is 0000. This is because MULTIPLICAND is the first data item declared in the segment. The assembler, then, will find that the displacement of MULTIPLICAND is 0000H. When the assembler reads the program the second time to produce the binary codes for the instructions, it will insert this displacement as part of the binary code for the instruction MOV AX, MULTIPLICAND. Since we know that the displacement of MULTIPLICAND is 0000, we could have written the instruction as MOV AX, [0000]. However, there would be a problem if we later changed the program by adding another data item before MULTIPLICAND in DATA_HERE. The displacement of MULTIPLICAND would be changed. Therefore, we would have to remember to

```
Turbo Assembler  Version 1.0                                Page 1

                                        ; 8086 PROGRAM F3-14.ASM
                                ;ABSTRACT  : This program multiplies the two 16-bit words in the memory
                                        ; locations called MULTIPLICAND and MULTIPLIER. The result
                                        ; is stored in the memory location, PRODUCT
                                ;REGISTERS : Uses CS, DS, AX, DX
                                ;PORTS     : None used

 0000                           DATA_HERE    SEGMENT
 0000  204A                                  MULTIPLICAND  DW 204AH      ; First word here
 0002  382A                                  MULTIPLIER    DW 382AH      ; Second word here
 0004  02*(0000)                             PRODUCT       DW 2 DUP(0)   ; Result of multiplication here
 0008                           DATA_HERE    ENDS

 0000                           CODE_HERE    SEGMENT
                                             ASSUME    CS:CODE_HERE, DS:DATA_HERE
 0000  B8 0000s                 START:       MOV  AX, DATA_HERE          ; Initialize DS register
 0003  8E D8                                 MOV  DS, AX
 0005  A1 0000r                              MOV  AX, MULTIPLICAND       ; Get one word
 0008  F7 26 0002r                           MUL  MULTIPLIER             ; Multiply by second word
 000C  A3 0004r                              MOV  PRODUCT, AX            ; Store low word of result
 000F  89 16 0006r                           MOV  PRODUCT+2, DX          ; Store high word of result
 0013  CC                                    INT  3                      ; Wait for command from user
 0014                           CODE_HERE    ENDS
                                             END START


Turbo Assembler  Version 1.0                                Page 2
Symbol Table


Symbol Name                     Type   Value

??DATE                          Text   "04-06-89"
??FILENAME                      Text   "F3-14   "
??TIME                          Text   "07:41:58"
??VERSION                       Number 0100
@CPU                            Text   0101H
@CURSEG                         Text   CODE_HERE
@FILENAME                       Text   F3-14
@WORDSIZE                       Text   2
MULTIPLICAND                    Word   DATA_HERE:0000
MULTIPLIER                      Word   DATA_HERE:0002
PRODUCT                         Word   DATA_HERE:0004
START                           Near   CODE_HERE:0000

Groups & Segments               Bit Size Align  Combine Class

CODE_HERE                       16  0014 Para   none
DATA_HERE                       16  0008 Para   none
```

Fig. 3.16 Assembler listing for example program in Figure 3.14.

go through the entire program and correct the displacement in all instructions that access MULTIPLICAND. If you use a name to refer to each data item as shown, the assembler will automatically calculate the correct displacement of that data item for you and insert this displacement each time you refer to it in an instruction.

To summarize how this works, then, the instruction MOV AX, MULTIPLICAND is an example of direct addressing where the direct address or displacement of the desired data word in the data segment is represented by the name MULTIPLICAND. For instructions such as this, the assembler will automatically calculate the displacement of

the named data item from the start of the segment and insert this value as part of the binary code for the instruction. This can be seen on line 18 of the assembler listing shown in Fig. 3.16. When the instruction executes, the BIU will add the displacement contained in the instruction to the data segment base in DS to produce the 20-bit physical address of the data word named MULTIPLICAND.

The next instruction in the program in Fig. 3.16 is another example of direct addressing using a named data item. The instruction MUL MULTIPLIER multiplies the word from the memory location named MULTIPLIER in DATA_HERE by the word in the AX register. When the assembler reads through this program the first time, it will find that the displacement of MULTIPLIER in DATA_HERE is 0002H. When it reads through the program the second time, it inserts this displacement as part of the binary code for the MUL instruction, as shown on line 19 in Fig. 3.16. When the MUL MULTIPLIER instruction executes, the BIU will add the displacement contained in the instruction to the data segment base in DS to address MULTIPLIER in memory. After the multiplication, the low word of the result is left in the AX register, and the high word of the result is left in the DX register.

The next instruction, MOV PRODUCT, AX, in the program in Fig. 3.16 copies the low word of the result from AX to memory. The low byte of AX will be copied to a memory location named PRODUCT. The high byte of AX will be copied to the next higher address, which we can refer to as PRODUCT + 1. As you can see on line 20 in Fig. 3.16, the displacement of PRODUCT, 0004H, is inserted in the code for the MOV PRODUCT, AX instruction.

The following instruction in the program, MOV PRODUCT + 2, DX, copies the high word of the multiplication result from DX to memory. When the assembler reads this instruction, it will add the indicated "2" to the displacement it calculated for PRODUCT and insert the result as part of the binary code for the instruction, as shown on line 21 in Fig. 3.16. Therefore, when the instruction executes, the low byte of DX will be copied to memory at a displacement of PRODUCT + 2. The high byte of DX will be copied to a memory location which we can refer to as PRODUCT + 3. Fig. 3.15 shows how the two words of the product are put in memory. Note that the lower byte of a word is always put in the lower memory address.

This example program should show you that if you are using an assembler, names are a very convenient way of specifying the direct address of data in memory. In the next section we show you how to refer to addresses by name.

Naming Addresses — Labels

One type of name used to represent addresses is called a *label*. Labels are written in the label field of an instruction statement or a directive statement. One major use of labels is to represent the destination for jump and call instructions. Suppose, for example, we want the 8086 to jump back to some previous instruction over and over. Instead of computing the numerical address that we want the 8086 to jump to, we put a label in front of the destination instruction and write the jump instruction as JMP label:. Here is a specific example.

```
NEXT:  IN AL, 05H ;     Get data sample from port 05H;
                        Process data value read in
       .
       .
       JMP NEXT ;       Get next data value and process
```

If you use a label to represent an address, as shown in this example, the assembler will automatically calculate the address that needs to be put in the code for the jump instruction. The next two chapters show many examples of the use of labels with jump and call instructions.

Another example of using a name to represent an address is in the SEGMENT directive statement. The name DATA_HERE in the statement DATA_HERE SEG_MENT, for example, represents the starting address of a segment named DATA_HERE. Later we show you how we use this name to initialize the data segment register, but first we will discuss some other parts you need to know about in the example program in Fig. 3.14.

The ASSUME Directive

An 8086 program may have several logical segments that contain code and several that contain data. However, at any given time the 8086 works directly with only four physical segments: a *code segment*, a *data segment*, a *stack segment*, and an *extra segment*. The ASSUME directive tells the assembler which logical segment to use for each of these physical segments at a given time.

In Fig. 3.14, for example, the statement ASSUME CS:CODE_HERE, DS:DATA_HERE tells the assembler that the logical segment named CODE_HERE contains the instruction statements for the program and should be treated as a code segment. It also tells the assembler that it should treat the logical segment DATA_HERE as the data segment for this program. In other words, the DS:DATA_HERE part of the statement tells the assembler that for any instruction which refers to data in the data segment, data will be found in the logical segment DATA_HERE. The ASSUME . . . DS:DATA_HERE, for example, tells the assembler that a named data item such as MULTIPLICAND is contained in the logical segment called DATA_HERE. Given this information, the assembler can construct the binary codes for the instruction. As we explained before, the displacement of MULTIPLICAND from the start of the DATA_HERE segment will be inserted as part of the instruction by the assembler.

If you are using the stack segment and the extra segment in your program, you must include terms in the ASSUME statement to tell the assembler which logical segments to use for each of these. To do this, you might add terms such as SS:STACK_HERE, ES:EXTRA_HERE. As we will show later, you can put another ASSUME directive later in the

program to tell the assembler to use different logical segments from that point on.

If the ASSUME directive is not completely clear to you at this point, don't worry. We show many more examples of its use throughout the rest of the book. We introduced the ASSUME directive here because you need to put it in your programs for most 8086 assemblers. You can use the ASSUME statement in Fig. 3.14 as a model of how to write this directive for your programs.

Initializing Segment Registers

The ASSUME directive tells the assembler the names of the logical segments to use as the code segment, data segment, stack segment, and extra segment. The assembler uses displacements from the start of the specified logical segment to code out instructions. When the instructions are executed, the displacements in the instructions will be added to the segment base addresses represented by the 16-bit numbers in the segment registers to produce the actual physical addresses. The assembler, however, cannot directly load the segment registers with the upper 16 bits of the segment starting addresses as needed.

The segment registers other than the code segment register must be initialized by program instructions before they can be used to access data. The first two instructions of the example program in Fig. 3.14 show how you initialize the data segment register. The name DATA_HERE in the first instruction represents the upper 16 bits of the starting address you give the segment DATA_HERE. Since the 8086 does not allow us to move this immediate number directly into the data segment register, we must first load it into one of the general-purpose registers, then copy it into the data segment register. MOV AX, DATA-HERE loads the upper 16 bits of the segment starting address into the AX register. MOV DS, AX copies this value from AX to the data segment register. This is the same operation we described for hand coding the example program in Fig. 3.4, except that here we use the segment name instead of a number to refer to the segment base address. In this example we used the AX register to pass the value, but any 16-bit register other than a segment register can be used. If you are hand coding your program, you can just insert the upper 16 bits of the 20-bit segment starting address in place of DATA_HERE in the instruction. For example, if in your particular system you decide to locate DATA_HERE at address 00300H, DS should be loaded with 0030H. If you are using an assembler, you can use the segment name to refer to the segment base address, as shown in the example.

If you use the stack segment and the extra segment in a program, the stack segment register and the extra segment register must be initialized by program instructions in the same way.

When the assembler reads through your assembly language program, it calculates the displacement of each named variable from the start of the logical segment that contains it. The assembler also keeps track of the displacement of each instruction code byte from the start of a logical segment. The CS:CODE_HERE part of the ASSUME statement in Fig. 3.14 tells the assembler to calculate the displacements of the following instructions from the start of the logical segment CODE_HERE. In other words, it tells the assembler that when this program is run, the code segment register will contain the upper 16 bits of the address where the logical segment CODE_HERE was located in memory. The instruction byte displacements that the assembler is keeping track of are the values that the 8086 will put in the instruction pointer (IP) to fetch each instruction byte.

There are several ways in which the CS register can be loaded with the code segment base address and the instruction pointer can be loaded with the offset of the instruction byte to be fetched next. The first way is with the command you give your system to execute a program starting at a given address. A typical command of this sort is G = 0010:0000 <CR>. (<CR> means "press the return key.") This command will load CS with 0010 and load IP with 0000. The 8086 will then fetch and execute instructions starting from address 00100, the address produced when the BIU adds IP to the code segment base in the CS register.

As we will show you in the next two chapters, jump and call instructions load new values in IP, and in some cases they load new values in the CS register.

The END Directive

The END directive, as the name implies, tells the assembler to stop reading. Any instructions or statements that you write after an END directive will be ignored.

ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT TOOLS

Introduction

For all but the very simplest assembly language programs, you will probably want to use some type of *microcomputer development system* and *program development tools* to make your work easier. A typical system might consist of an IBM PC-type microcomputer with at least several hundred kilobytes of RAM, a keyboard and video display, floppy and/or hard disk drives, a printer, and an emulator. Fig. 3.17 shows an Applied Microsystems ES 1800 16-bit emulator which can be added to an IBM PC/AT or compatible computer to produce a complete 8086/80186/80286 development system.

The following sections give you an introduction to several common program development tools which you use with a system such as this. Most of these tools are programs which you run to perform some function on the program you are

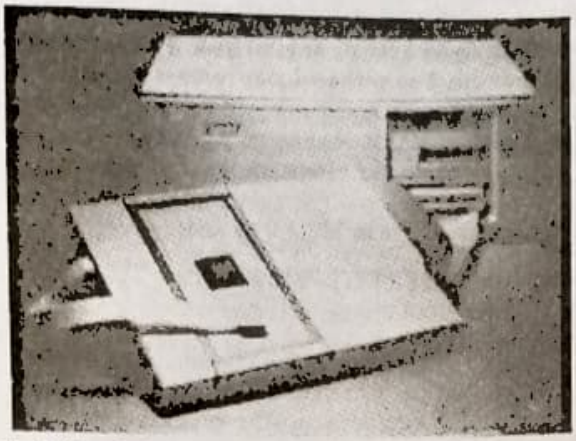

Fig. 3.17 Applied Microsystems ES 1800 16-bit emulator. (Applied Microsystems Corp.)

writing. You will have to consult the manuals for your system to get the specific details, but this section should give you an overview of the steps involved in developing an assembly language program. An accompanying lab manual takes you through the use of all these tools with the SDK-86 board and an IBM PC- type computer.

Editor

An *editor* is a program which allows you to create a file containing the assembly language statements for your program. Examples of suitable editors are PC Write, Wordstar, and the editor that comes with some assemblers.

Figure 3.14 shows an example of the format you should use when typing in your program. The actual position of each field on a line is not important, but you must put the fields of each statement in the correct order, and you must leave at least one blank between fields. Whenever possible, we like to line the fields up in columns so that it is easier to read the program.

As you type in your program, the editor stores the ASCII codes for the letters and numbers in successive RAM locations. If you make a typing error, the editor will let you back up and correct it. If you leave out a program statement, the editor will let you move everything down and insert the line. This is much easier than working with pencil and paper, even if you type as slowly as I do.

When you have typed in all of your program, you then save the file on a floppy or hard disk. This file is called a *source file*. The next step is to process the source file with an assembler. Incidentally, if you are going to use the TASM or MASM assembler, you should give your source file name the extension .ASM. You might, for instance, give the example source program in Fig. 3.14 a name such as MULTIPLY.ASM.

Assembler

As we told you earlier in the chapter, an *assembler* program is used to translate the assembly language mnemonics for instructions to the corresponding binary codes. When you run the assembler, it reads the source file of your program from the disk where you saved it after editing. On the first pass through the source program, the assembler determines the displacement of named data items, the offset of labels, etc., and puts this information in a *symbol table*. On the second pass through the source program, the assembler produces the binary code for each instruction and inserts the offsets, etc., that it calculated during the first pass.

The assembler generates two files on the floppy or hard disk. The first file, called the *object file*, is given the extension .OBJ. The object file contains the binary codes for the instructions and information about the addresses of the instructions. After further processing, the contents of this file will be loaded into memory and run. The second file generated by the assembler is called the *assembler list file* and is given the extension .LST. Figure 3.16 shows the assembler list file for the source program in Fig. 3.14. The list file contains your assembly language statements, the binary codes for each instruction, and the offset for each instruction. You usually send this file to a printer so that you will have a printout of the entire program to work with when you are testing and troubleshooting the program. The assembler listing will also indicate any typing or syntax (assembly language grammar) errors you made in your source program.

To correct the errors indicated on the listing, you use the editor to reedit your source program and save the corrected source program on disk. You then reassemble the corrected source program. It may take several times through the edit-assemble loop before you get all the syntax errors out of your source program.

> *NOTE: The assembler only finds syntax errors; it will not tell you whether your program does what it is supposed to do. To determine whether your program works, you have to run the program and test it.*

Now let's take a closer look at some of the information given on the assembler listing in Fig. 3.16. The leftmost column in the listing gives the offsets of data items from the start of the data segment and the offsets of code bytes from the start of the code segment. Note that the assembler generates only offsets, not absolute physical addresses. A linker or locator will be used to assign the physical starting addresses for the segments.

As evidence of this, note that the MOV AX, DATA_HERE statement is assembled with some blanks after the basic instruction code because the start of DS is not known at the time the program is assembled.

The trailer section of the listing in Fig. 3.16 gives some additional information about the segments and names used in the program. The statement CODE_HERE 16 0014 Para none, for example, tells you that the segment CODE_HERE is 14H bytes long. The statement MULTIPLIER Word DATA_HERE:0002 tells you that

MULTIPLIER is a variable of type word and that it is located at an offset of 0002 in the segment DATA_HERE.

Linker

A *linker* is a program used to join several object files into one large object file. When writing large programs, it is usually much more efficient to divide the large program into smaller *modules*. Each module can be individually written, tested, and debugged. Then, when all the modules work, their object modules can be linked together to form a large, functioning program. Also, the object modules for useful programs — a square root program, for example — can be kept in a *library file* and linked into other programs as needed.

NOTE: On IBM PC-type computers, you must run the LINK program on your .OBJ file, even if it contains only one assembly module.

The linker produces a *link file* which contains the binary codes for all the combined modules. The linker also produces a *link map* file which contains the address information about the linked files. The linker, however, does not assign absolute addresses to the program; it assigns only relative addresses starting from zero. This form of the program is said to be *relocatable* because it can be put anywhere in memory to be run. The linkers which come with the TASM or MASM assemblers produce link files with the .EXE extension.

If your program does not require any external hardware, you can use a program called a *debugger* to load and run the .EXE file. We will tell you more about debuggers later. The debugger program which loads your program into memory automatically assigns physical starting addresses to the segments.

If you are going to run your program on a system such as an SDK-86 board, then you must use a *locator program* to assign physical addresses to the segments in the .EXE file.

Locator

A *locator* is a program used to assign the specific addresses of where the segments of object code are to be loaded into memory. A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a .EXE file to a .BIN file which has physical addresses. You can then use the SDKCOM1 program from Chapter 12 to download the .BIN file to the SDK-86 board. The SDKCOM 1 program can also be used to run the program and debug it on the SDK-86 board.

Debugger

If your program requires no external hardware or requires only hardware accessible directly from your microcomputer, then you can use a *debugger* to run and debug your program. A debugger is a program which allows you to load your object code program into system memory, execute the program, and troubleshoot or "debug" it. The debugger allows you to look at the contents of registers and memory locations after your program runs. It allows you to change the contents of registers and memory locations and rerun the program. Some debuggers allow you to stop execution after each instruction so that you can check or alter memory and register contents. A debugger also allows you to set a *breakpoint* at any point in your program. If you insert a breakpoint, the debugger will run the program up to the instruction where you put the breakpoint and then stop execution. You can then examine register and memory contents to see whether the results are correct at that point. If the results are correct, you can move the breakpoint to a later point in the program. If the results are not correct, you can check the program up to that point to find out why they are not correct.

The point here is that the debugger commands help you to quickly find the source of a problem in your program. Once you find the problem, you can then cycle back and correct the algorithm if necessary, use the editor to correct your source program, reassemble the corrected source program, relink, arid run the program again.

A basic debugger comes with the DOS for most IBM PC-type computers, but more powerful debuggers such as Borland's Turbo Debugger and Microsoft's Codeview debugger make debugging much easier because they allow you to directly see the contents of registers and memory locations change as a program executes. In a later chapter we show you how to use one of these debuggers.

Microprocessor prototyping boards such as the SDK- 86 contain a debugger program in ROM. On boards such as this, the debugger is commonly called a *monitor program* because it lets you monitor program activity. The SDK-86 monitor program, for example, lets you enter and run programs, single-step through programs, examine register and memory contents, and insert breakpoints.

Emulator

Another way to run your program is with an *emulator,* such as that shown in Fig. 3.17. An emulator is a mixture of hardware and software. It is usually used to test and debug the hardware and software of an external system, such as the prototype of a microprocessor-based instrument. Part of the hardware of an emulator is a multiwire cable which connects the host system to the system being developed. A plug at the end of the cable is plugged into the prototype system in place of its microprocessor. Through this connection the software of the emulator allows you to download your object code program into RAM in the system being tested and run it. Like a debugger, an emulator allows you to load and run programs, examine and change the contents of registers, examine and change the contents of memory locations, and insert breakpoints in the program. The emulator also takes a "snapshot" of the contents of registers, activity on the address and data bus, and the state of the flags as each instruction executes.

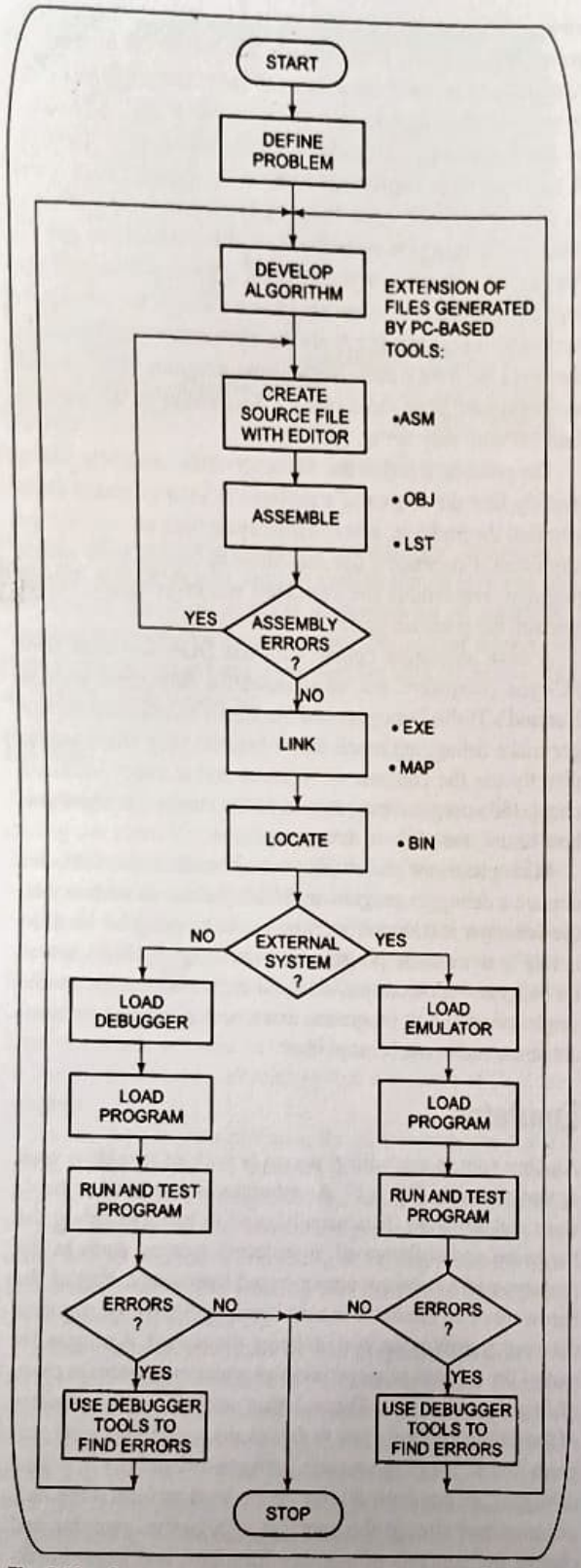

Fig. 3.18 Program development algorithm

The emulator stores this *trace data*, as it is called, in a large RAM. You can do a printout of the trace data to see the results that your program produced on a step-by-step basis.

Another powerful feature of an emulator is the ability to use either system memory or the memory on the prototype for the program you are debugging. In a later chapter we discuss in detail the use of an emulator in developing a microprocessor-based product.

Summary of the Use of Program Development Tools

Figure 3.18 summarizes the steps in developing a working program. This may seem complicated, but if you use the accompanying lab manual to go through the process a couple of times, you will find that it is quite easy.

The first and most important step is to think out very carefully what you want the program to do and how you want the program to do it. Next, use an editor to create the source file for your program. Assemble the source file. If the assembler list file indicates any errors in your program, use the editor to correct these errors. Cycle through the edit-assemble loop until the assembler tells you on the listing that it found no errors. If your program consists of several modules, then use the linker to join their object modules into one large object module. If your system requires it, use a locate program to specify where you want your program to be put in memory. Your program is now ready to be loaded into memory and run. Note that Fig. 3.18 also shows the extensions for the files produced by each of the development programs.

If your program does not interact with any external hardware other than that connected directly to the system, then you can use the system debugger to run and debug your program. If your program is intended to work with external hardware, such as the prototype of a microprocessor-based instrument, then you will probably use an emulator to run and debug your program. We will be discussing and showing the use of these program development tools throughout the rest of this book.

Checklist of Important Terms and Concepts in This Chapter

If you do not remember any of the terms or concepts in the following list, use the index to find them in the chapter.

- *Algorithm*
- *Flowcharts and flowchart symbols*
- *Structured programming*
- *Pseudocode*
- *Top-down and bottom-up design methods*
- *Sequence, repetition, and decision operations*

- *SEQUENCE, IF-THEN-ELSE, IF-THEN, nested IF-THEN-ELSE, CASE, WHILE-DO, REPEAT-UNTIL programming structures*
- *8086 instructions: MOV, IN, OUT, ADD, ADC, SUB, SBB, AND, OR, XOR, MUL, DIV*
- *Instruction mnemonics*
- *Initialization list*
- *Assembly language program format*
- *Instruction template: W bit, MOD, R/M, D bit*
- *Segment-override prefix*
- *Assembler directives: SEGMENT, ENDS, END, DB, DW, DD, EQU, ASSUME*
- *Accessing named data items*
- *Editor*
- *Assembler*
- *Linker: library file, link files, link map, relocatable*
- *Locator*
- *Debugger, monitor program*
- *Emulator, trace data*

Review Questions and Problems

•1. List the major steps in developing an assembly language program.

•2. What is the main advantage of a top-down design approach to solving a programming problem?

•3. Why should you develop a detailed algorithm for a program before writing down any assembly language instructions?

•4. (a) What are the three basic structure types used to write the algorithm for a program?
(b) What is the advantage of using only these structures when writing the algorithm for a program?

•••5. A program is like a recipe. Use a flowchart or pseudocode to show the algorithm for the following recipe. The operations in it are sequence and repetition. Instead of implementing the resulting algorithm in assembly language, implement it in your microwave and use the result to help you get through the rest of the book.

Peanut Brittle:

1 cup sugar
0.5 cup white corn syrup
1 cup unsalted peanuts
1 teaspoon butter
1 teaspoon vanilla
1 teaspoon baking soda

i) Put sugar and syrup in 1.5-quart casserole (with handle) and stir until thoroughly mixed.
ii) Microwave at HIGH setting for 4 minutes.
iii) Add peanuts and stir until thoroughly mixed.
iv) Microwave at HIGH setting for 4 minutes. Add butter and vanilla, stir until well mixed, and microwave at HIGH setting for 2 more minutes.
v) Add baking soda and gently stir until light and foamy. Pour mixture onto nonstick cookie sheet and let cool for 1 hour. When cool, break into pieces. Makes 1 pound.

••6. Use a flowchart or pseudocode to show the algorithm for a program which gets a number from a memory location, subtracts 20H from it, and outputs 01H to port 3AH if the result of the subtraction is greater than 25H.

•••7. Given the register contents in Fig. 3.19, answer the following questions:
(a) What physical address will the next instruction be fetched from?
(b) What is the physical address for the top of the stack?

•••8. Describe the operation and results of each of the following instructions, given the register contents shown in Fig. 3.19. Include in your answer the physical address or register that each instruction will get its operands from and the physical address or register in which each instruction will put the result. Use the instruction descriptions in Chapter 6 to help you. Assume that the following instructions are independent, not sequential, unless listed together under a letter.

(a) MOV AX, BX
(b) MOV CL, 37H
(c) INC BX
(d) MOV CX, [246BH]
(e) MOV CX, 246BH
(f) ADD AL, DH
(g) MUL BX
(h) DEC BP
(i) DIV BL
(j) SUB AX, DX
(k) OR CL, BL
(l) NOT AH
(m) ROL BX, 1
(n) AND AL, CH
(o) MOV DS, AX
(p) ROR BX, CL

					DATA SEGMENT	
ES	6000				5000CH	D7
CS	4000				50008H	9A
SS	7000				5000AH	7C
DS	5000				50009H	DB
IP	43E8				50008H	C3
SP	0000				50007H	B2
BP	2468				50006H	49
SI	4C00				50005H	21
DI	7D00				50004H	89
					50003H	71
					50002H	22
					50001H	4A
					50000H	3B

	AH	AL		BH	BL
AX	42	35	BX	07	5A

	CH	CL		DH	DL
CX	00	04	DX	33	02

Fig. 3.19 8086 register and memory contents for Problems 7, 8, and 10.

(q) AND AL, OFH
(r) MOV, AX, [BX]
(s) MOV [BX] [SI], CL

**9. See if you can spot the grammatical (syntax) errors in the following instructions (use Chapter 6 to help you):
(a) MOV BH, AX
(b) MOV DX, CL
(c) ADD AL, 2073H
(d) MOV 7632H, CX
(e) IN BL, 04H

***10. Show the results that will be in the affected registers or memory locations after each of the following groups of instructions executes. Assume that each group of instructions starts with the register and memory contents shown in Fig. 3.19. (Use Chapter 6.)
(a) ADD BL, AL
MOV [00041, BL
ROR DI, 04
(b) MOV CL, 04
(c) ADD AL, BH
DAA
(d) MOV BX. 000AH
MOV AL, [BX]
SUB AL, CL
INC BX
MOV [BX), AL

***11. Write the 8086 instruction which will perform the indicated operation. Use the instruction overview in this chapter and the detailed descriptions in Chapter 6 to help you.
(a) Copy AL to BL.
(b) Load 43H into CL.
(c) Increment the contents of CX by 1.
(d) Copy SP to BP.
(e) Add 07H to DL.
(f) Multiply AL times BL.
(g) Copy AX to a memory location at offset 245AH in the data segment,
(h) Decrement SP by 1.
(i) Rotate the most significant bit of AL into the least significant bit position,
(j) Copy DL to a memory location whose offset is in BX.
(k) Mask the lower 4 bits of BL.
(l) Set the most significant bit of AX to a 1, but do not affect the other bits,
(m) Invert the lower 4 bits of BL, but do not affect the other bits.

***12. Construct the binary code for each of the following 8086 instructions.
(a) MOV BL, AL
(b) MOV [BX], CX
(c) ADD BX, 59H[DI]
(d) SUB [2048], DH
(e) XCHG CH, ES:[BX]
(f) ROR AX, 1
(g) OUT DX, AL
(h) AND AL, OFH
(i) NOP
(j) IN AL, DX

***13. Describe the function of each assembler directive and instruction statement in the short program shown in Fig. 3.20.

```
;PRESSURE READ PROGRAM

DATA_HERE SEGMENT
     PRESSURE  DB  0        ;storage for pressure
DATA_HERE ENDS

PRESSURE_PORT      EQU 04H  ;Pressure sensor connected
                            ; to port 04H
CORRECTION_FACTOR EQU 07H   ;Current correction factor
                            ; of 07

CODE_HERE SEGMENT
     ASSUME CS:CODE_HERE, DS:DATA_HERE
     MOV  AX, DATA_HERE
     MOV  DS, AX
     IN   AL, PRESSURE_PORT
     ADD  AL, CORRECTION_FACTOR
     MOV  PRESSURE, AL
CODE_HERE ENDS
          END
```

Fig. 3.20 Program for Problem 13.

**14. Describe how an assembly language program is developed and debugged using system tools such as editors, assemblers, linkers, locators, emulators, and debuggers.

***15. Write the pseudocode representation for the flowchart in Fig. 3.18.