AND INTERFACING

PROGRAMMING AND HARDWARE

SECOND EDITION

DOUGLAS V. HALL

McGRAW-HILL INTERNATIONAL EDITIONS
Computer Science Series

8086 • 80286 • 80386 • 80486

CHAPTER

8086 Family Assembly Language Programming — Introduction

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.

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 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. 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. Figure 3-1, p. 38, 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.

Figure 3-2, p. 38, 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. A racetrack- or circular-shaped symbol labeled START is used to indicate the *beginning*

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 LS 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.

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.

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 setting 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.

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. 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 (p. 61) 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 Figure 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 microprocessor

CHAPTER 4

Implementing Standard Program Structures in 8086 Assembly Language

In Chapter 3 we worked very hard to convince you that you should not try to write programs directly in assembly language. The analogy of building a house without a plan should come to mind here. When faced with a programming problem, you should solve the problem and write the algorithm for the solution using the standard program structures we described. Then you simply translate each step in the flowchart or pseudocode to a group of one to four assembly language instructions which will implement that step. The comments in the assembly language program should describe the functions of each instruction or group of instructions, so you essentially write the comments for the program, then write the assembly language instructions which implement those comments. Once you learn how to implement each of the standard programming structures, you should find it quite easy to translate algorithms to assembly language. Also, as we will show you, the standard structure approach makes debugging relatively easy.

The purposes of this chapter are to show you how to write the algorithms for some common programming problems, how to implement these algorithms in 8086 assembly language, and how to systematically debug assembly language programs. In the process you will also learn more about how some of the 8086 instructions work.

OBJECTIVES

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

1. Write flowcharts or pseudocode for simple programming problems.
2. Implement SEQUENCE, IF-THEN-ELSE, WHILE-DO, and REPEAT-UNTIL program structures in 8086 assembly language.
3. Describe the operation of selected data transfer, arithmetic, logical, jump, and loop instructions.
4. Use based and indexed addressing modes to access data in your programs.
5. Describe a systematic approach to debugging a simple assembly language program using debugger, monitor, or emulator tools.
6. Write a delay loop which produces a desired amount of delay on a specific 8086 system.

SIMPLE SEQUENCE PROGRAMS

Finding the Average of Two Numbers

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

A common need in programming is to find the average of two numbers. Suppose, for example, we know the maximum temperature and the minimum temperature for a given day, and we want to determine the average temperature. The sequence of steps we go through to do this might look something like the following.

Add maximum temperature and minimum temperature.

Divide sum by 2 to get average temperature.

This sequence doesn't look much like an assembly language program, and it shouldn't. The algorithm at this point should be general enough that it could be implemented in any programming language, or on any machine. Once you are reasonably sure of your algorithm, then you can start thinking about the architecture and instructions of the specific microcomputer on which you plan to run the program. Now let's show you how we get from the algorithm to the assembly language program for it.

SETTING UP THE DATA STRUCTURE

One of the first things for you to think about in this process is the data that the program will be working with. You need to ask yourself questions such as:

1. Will the data be in memory or in registers?
2. Is the data of type byte, type word, or perhaps type doubleword?
3. How many data items are there?
4. Does the data represent only positive numbers, or does it represent positive and negative (signed) numbers?

5. For more complex problems, you might ask how the data is structured. For example, is the data in an array or in a record?

Let's assume for this example that the data is all in memory, that the data is of type byte, and that the data represents only positive numbers in the range 0 to 0FFH. The top part of Figure 4-1, between the DATA SEGMENT and the DATA ENDS directives, shows how you might set up the data structure for this program. It is very similar to the data structure for the multiplication example in the last chapter. In the logical segment called DATA, HI_TEMP is declared as a variable of type byte and initialized with a value of 92H. In an actual application, the value in HI_TEMP would probably be put there by another program which reads the output from a temperature sensor. The statement LO_TEMP DB 52H declares a variable of type byte and initializes it with the value 52H. The statement AV_TEMP DB ? sets aside a byte location to store the average temperature, but does not initialize the location to any value. When the program executes, it will write a value to this location.

INITIALIZATION CHECKLIST

Although it does not show in the algorithm, you know from the discussion in Chapter 3 that most programs start with a series of initialization instructions. For this example program, all you have to initialize is the data segment register. The MOV AX,DATA and MOV DS,AX instructions at the start of the program in Figure 4-1 do this.

These instructions load the DS register with the upper 16 bits of the starting address for the data segment. If you are using an assembler, then just put the hex for the upper 16 bits of the address in the MOV AX,DATA instruction in place of the name.

CHOOSING INSTRUCTIONS TO IMPLEMENT THE ALGORITHM

The next step is to look at the algorithm to determine the major actions that you want the program to perform. If you have written the algorithm correctly, then all you should have to do is translate each step in the algorithm to one to four assembly language instructions which will implement that step.

You want the program to add two byte-type numbers together, so scan through the instruction groups in Chapter 3 to determine which 8086 instruction will do this for you. The ADD instruction is the obvious choice in this case.

Next, find and read the detailed discussion of the ADD instruction in Chapter 6. From the discussion there, you can determine how the instruction works and see if it will do the necessary job. From the discussion of the ADD instruction, you should find that the ADD instruction has the format ADD destination,source. A byte from the specified source is added to a byte in the specified destination, or a word from the specified source is added to a word in the specified destination. (Note that you cannot directly add a byte to a word.) The result in either case is put in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a

```
 1                                ; 8086 PROGRAM    F4-01.ASM
 2                     ;ABSTRACT  : This program averages two temperatures
 3                                ; named HI_TEMP and LO_TEMP and puts the
 4                                ; result in the memory location AV_TEMP.
 5                     ;REGISTERS : Uses DS, CS, AX, BL
 6                     ;PORTS     : None used
 7
 8 0000                DATA    SEGMENT
 9 0000  92                    HI_TEMP  DB  92H    ; Max temp storage
10 0001  52                    LO_TEMP  DB  52H    ; Low temp storage
11 0002  ??                    AV_TEMP  DB   ?     ; Store average here
12 0003                DATA    ENDS
13
14 0000                CODE    SEGMENT
15                             ASSUME CS:CODE, DS:DATA
16 0000  B8 0000s      START:  MOV  AX, DATA       ; Initialize data segment
17 0003  8E D8                 MOV  DS, AX
18 0005  A0 0000r              MOV  AL, HI_TEMP    ; Get first temperature
19 0008  02 06 0001r           ADD  AL, LO_TEMP    ; Add second to it
20 000C  B4 00                 MOV  AH, 00H        ; Clear all of AH register
21 000E  80 D4 00              ADC  AH, 00H        ; Put carry in LSB of AH
22 0011  B3 02                 MOV  BL, 02H        ; Load divisor in BL register
23 0013  F6 F3                 DIV  BL             ; Divide AX by BL. Quotient in AL,
24                                                 ; and remainder in AH
25 0015  A2 0002r              MOV  AV_TEMP, AL    ; Copy result to memory
26 0018                CODE    ENDS
27                             END  START
```

FIGURE 4-1 8086 program to average two temperatures.

memory location. However, in a single instruction the source and the destination cannot both be memory locations. This means that you have to move one of the operands from memory to a register before you can do the ADD.

Another point to consider here is that if you add two 8-bit numbers, the sum can be larger than 8 bits. Adding F0H and 40H, for example, gives 130H. The 8-bit destination will contain 30H, and the carry will be held in the carry flag. This means that to have the complete sum, you must collect the parts of the result in a location large enough to hold all 9 bits. A 16-bit register is a good choice.

To summarize, then, you need to move one of the numbers you want to add into a register, such as AL, add the other number from memory to it, and move any carry produced by the addition to the upper half of the 16-bit register which contains the sum in its lower 8 bits. Now let's take another look at Figure 4-1 to see how you implement this step in the algorithm with 8086 instructions.

The instruction MOV AL,HI_TEMP copies one of the temperatures from a memory location to the AL register. The name HI_TEMP in the instruction represents the direct address or displacement of the variable in the logical segment DATA. The ADD AL,LO_TEMP instruction adds the specified byte from memory to the contents of the AL register. The lower 8 bits of the sum are left in the AL register. If the addition produces a result greater than FFH, the carry flag will be set to a 1. If the addition produces a result less than or equal to FFH, the carry flag will be a 0. In either case, we want to get the contents of the carry flag into the least significant bit of the AH register, so that the entire sum is in the AX register.

The MOV AH,00H instruction clears all the bits of AH to 0's. The ADC AH,00H instruction adds the immediate number 00H plus the contents of the carry flag to the contents of the AH register. The result will be left in the AH register. Since we cleared AH to all 0's before the add, what we are really adding is 00H + 00H + CF. The result of all this is that the carry bit ends up in the least significant bit of AH, which is what we set out to do.

The next major action in our algorithm is to divide the sum of the two temperatures by 2. To determine how this step can be translated to assembly language instructions, look at the instruction groups in the last chapter to see if the 8086 has a Divide instruction. You should find that it has two Divide instructions, DIV and IDIV. DIV is for dividing unsigned numbers, and IDIV is used for dividing signed binary numbers. Since in this example we are dividing unsigned binary numbers, look up the DIV instruction in Chapter 6 to find out how it works.

The DIV instruction can be used to divide a 16-bit number in AX by a specified byte in a register or in a memory location. After the division, an 8-bit quotient is left in the AL register, and an 8-bit remainder is left in the AH register. The DIV instruction can also be used to divide a 32-bit number in the DX and AX registers by a 16-bit number from a specified register or memory location. In this case, a 16-bit quotient is left in the AX register, and a 16-bit remainder is left in the DX register. In either case, there is a problem if the quotient is too large to fit in AX for a 32-bit divide or AL for a 16-bit divide. Fortunately, the data in the example here is such that the problem will not arise. In a later chapter we discuss what to do about this problem.

Remember from the previous discussion that the sum of the two temperatures is already positioned in the AX register as required by the DIV operation. Before we can do the DIV operation, however, we have to get the divisor, 02H, into a register or memory location to satisfy the requirements of the DIV instruction. A simple way to do this is with the MOV BL,02H instruction, which loads the immediate number 02H into the BL register. Now you can do the divide operation with the instruction DIV BL. The 8-bit quotient from the division will be left in the AL register.

The algorithm doesn't show it, but in our discussion of the data structure we said that the minimum, maximum, and average temperatures were all in memory locations. Therefore, to complete the program, you have to copy the quotient in AL to the memory location we set aside for the average temperature. As shown in Figure 4-1, the instruction MOV AV_TEMP,AL will copy AL to this memory location.

NOTE: We could have used the remainder from the division in AH to round off the average temperature to the nearest degree, but that would have made the program more complex than we wanted for this example.

SUMMARY OF CONVERTING AN ALGORITHM TO ASSEMBLY LANGUAGE

The first step in converting an algorithm to assembly language is to set up the data structure that the algorithm will be working with. The next step is to write at the start of the code segment any instructions required to initialize variables, segment registers, peripheral devices, etc. Then determine the instructions required to implement each of the major actions in the algorithm, and decide how the data must be positioned for these instructions. Finally, insert the MOV or other instructions required to get the data into the correct position for these instructions.

A Few Comments about the 8086 Arithmetic Instructions

The 8086 has instructions to add, subtract, multiply, and divide. It can operate on signed or unsigned binary numbers, BCD numbers, or numbers represented in ASCII. Rather than put a lot of arithmetic examples at this point in the book, we show arithmetic examples with each arithmetic instruction description in Chapter 6. The description of the MUL instruction in Chapter 6, for example, shows how unsigned binary numbers are multiplied. Also we show other arithmetic examples as needed throughout the rest of the book. If you need to do some arithmetic operations with an 8086, there are a few instructions in addition to the basic add, subtract,

the program executes to this breakpoint, you can check AL to see if the division produced the results you predicted. If the 8086 is working at all, it will almost always do operations such as this correctly, so recheck your predictions if you disagree with it.

It helps your frustration level if you make a game of thinking where to put breakpoints to track down the little bug that is messing up your program. With a little practice you should soon develop an efficient debugging algorithm of your own using the specific tools available on your system. In the next chapter we show you how to use a more powerful debugger to run and debug programs in an IBM PC-type computer.

Converting Two ASCII Codes to Packed BCD

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

Computer data is often transferred as a series of 8-bit ASCII codes. If, for example, you have a microcomputer connected to an SDK-86 board and you type a 9 on an ASCII-encoded computer terminal keyboard, the 8-bit ASCII code sent to the SDK-86 will be 00111001 binary, or 39H. If you type a 5 on the keyboard, the code sent to the computer will be 00110101 binary or 35H, the ASCII code for 5. As shown in Table 1-2, the ASCII codes for the numbers 0 through 9 are 30H through 39H. The lower nibble of the ASCII codes contains the 4-bit BCD code for the decimal number represented by the ASCII code.

For many applications, we want to convert the ASCII code to its simple BCD equivalent. We can do this by simply replacing the 3 in the upper nibble of the byte with four 0's. For example, suppose we read in 00111001 binary or 39H, the ASCII code for 9. If we replace the upper 4 bits with 0's, we are left with 00001001 binary or 09H. The lower 4 bits then contain 1001 binary, the BCD code for 9. Numbers represented as one BCD digit per byte are called *unpacked BCD*.

For applications in which we are going to perform mathematical operations on the BCD numbers, we usually combine two BCD digits in a single byte. This form is called *packed BCD*. Figure 4-2 shows examples of ASCII, unpacked BCD, and packed BCD. The problem we are going to work on here is how to convert two numbers from ASCII code form to unpacked BCD form and then pack the two BCD digits into one byte. Figure 4-2 shows in numerical form the steps we want the program to perform. When you are writing a program which manipulates data such as this, a numerical example will help you visualize the algorithm.

The algorithm for this problem can be stated simply as

Convert first ASCII number to unpacked BCD.

Convert second ASCII number to unpacked BCD.

Move first BCD nibble to upper nibble position in byte.

Pack two BCD nibbles in one byte.

Now let's see how you can implement this algorithm in 8086 assembly language.

ASCII	5	0011 0101	= 35H
ASCII	9	0011 1001	= 39H
UNPACKED BCD	5	0000 0101	= 05H
UNPACKED BCD	9	0000 1001	= 09H
UNPACKED BCD 5 MOVED TO UPPER NIBBLE		0101 0000	= 50H
PACKED BCD	59	0101 1001	= 59H

FIGURE 4-2 ASCII, unpacked BCD, and packed BCD examples.

THE DATA STRUCTURE AND INITIALIZATION LIST

For this example program, let's assume that the ASCII code for 5 was received and put in the BL register, and the second ASCII code was received and left in the AL register. Since we are not using memory for data in this program, we do not need to declare a data segment or initialize the data segment register. Incidentally, in a real application this program would probably be a procedure or a part of a larger program.

MASKING WITH THE AND INSTRUCTION

The first operation in the algorithm is to convert a number in ASCII form to its unpacked BCD equivalent. This is done by replacing the upper 4 bits of the ASCII byte with four 0's. The 8086 AND instruction can be used to do this operation. Remember from basic logic or from the review in Chapter 1 that when a 1 or a 0 is ANDed with a 0, the result is always a zero. ANDing a bit with a 0 is called *masking* that bit because the previous state of the bit is hidden or masked. To mask 4 bits in a word, then, all you do is AND each bit you want to mask with a 0. A bit ANDed with a 1, remember, is not changed.

According to the description of the AND instruction in Chapter 6, the instruction has the format AND destination,source. The instruction ANDs each bit of the specified source with the corresponding bit of the specified destination and puts the result in the specified destination. The source can be an immediate number, a register, or a memory location specified in one of those 24 different ways. The destination can be a register or a memory location. The source and the destination must both be bytes, or they must both be words. The source and the destination cannot both be memory locations in an instruction.

For this example the first ASCII number is in the BL register, so we can just AND an immediate number with this register to mask the desired bits. The upper 4 bits of the immediate number should be 0's because these correspond to the bits we want to mask in BL. The lower 4 bits of the immediate number should be 1's because we want to leave these bits unchanged. The immediate number, then, should be 00001111 binary or 0FH. The instruction to convert the first ASCII number is AND BL,0FH. When this instruction executes, it will leave the desired unpacked BCD in BL. Figure 4-3 shows how this will work for an ASCII number of 35H initially in BL.

ASCII 5	0011	0101
MASK	0000	1111
RESULT	0000	0101

FIGURE 4-3 Effects of ANDing with 1's and 0's.

For the next action in the algorithm, we want to perform the same operation on a second ASCII number in the AL register. The instruction AND AL,0FH will do this for us. After this instruction executes, AL will contain the unpacked BCD for the second ASCII number.

MOVING A NIBBLE WITH THE ROTATE INSTRUCTION

The next action in the algorithm is to move the 4 BCD bits in the first unpacked BCD byte to the upper nibble position in the byte. We need to do this so that the 4 BCD bits are in the correct position for packing with the second BCD nibble. Take another look at Figure 4-2 to help you visualize this. What we are effectively doing here is swapping or exchanging the top nibble with the bottom nibble of the byte. If you check the instruction groups in Chapter 3, you will find that the 8086 has an Exchange instruction, XCHG, which can be used to swap two bytes or to swap two words. The 8086 does not have a specific instruction to swap the nibbles in a byte. However, if you think of the operation that we need to do as shifting or rotating the BCD bits 4 bit positions to the left, this will give you a good idea which instruction will do the job for you. The 8086 has a wide variety of rotate and shift instructions. For now, let's look at the rotate instructions. There are two instructions, ROL and RCL, which rotate the bits of a specified operand to the left. Figure 4-4 shows in diagram form how these two instructions work. For ROL, each bit in the specified register or memory location is rotated 1 bit position to the left. The bit that was the MSB is rotated around into the LSB position. The old MSB is also copied to the carry flag. For the RCL instruction, each bit of the specified register or memory location is also rotated 1 bit position to the left. However, the bit that was in the MSB position is moved to the carry flag, and the bit that was in the carry flag is moved into the LSB position. The C in the middle of the mnemonic should help you remember that the carry flag is included in the rotated loop when the RCL instruction executes.

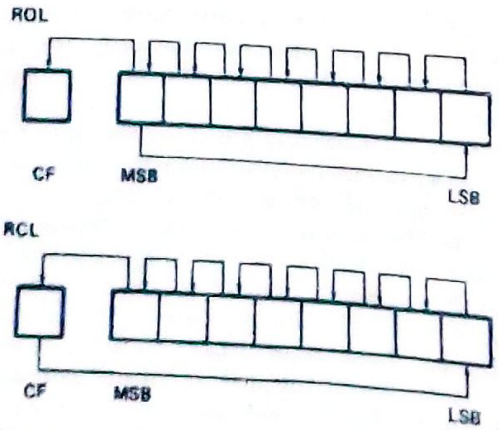

FIGURE 4-4 ROL instruction and RCL instruction operations for byte operands.

In the example program we really don't want the contents of the carry flag rotated into the operand, so the ROL instruction seems to be the one we want. If you consult the ROL instruction description in Chapter 6, you will find that the instruction has the format ROL destination,count. The destination can be a register or a memory location. It can be a byte location or a word location. The count can be the immediate number 1 specified directly in the instruction, or it can be a number previously loaded into the CL register. The instruction ROL AL,1, for example, will rotate the contents of AL 1 bit position to the left. We could repeat this instruction four times to produce the shift of 4 bit positions that we need for our BCD packing problem. However, there is an easier way to do it. We first load the CL register with the number of times we want to rotate AL. The instruction MOV CL,04H will do this. Then we use the instruction ROL BL,CL to do the rotation. When it executes, this instruction will automatically rotate BL the number of bit positions loaded into CL. Note that for the 80186 you can write the single instruction ROL BL,04H to do this job.

Now that we have determined the instructions needed to mask the upper nibbles and the instructions needed to move the first BCD digit into position, the only thing left is to pack the upper nibble from BL and the lower nibble from AL into a single byte.

COMBINING BYTES OR WORDS WITH THE ADD OR THE OR INSTRUCTION

You can't use a standard MOV instruction to combine two bytes into one as we need to do here. The reason is that the MOV instruction copies an operand from a specified source to a specified destination. The previous contents of the destination are lost. You can, however, use an ADD or an OR instruction to pack the two BCD nibbles.

As described in the previous program example, the ADD instruction adds the contents of a specified source to the contents of a specified destination and leaves the result in the specified destination. For the example program here, the instruction ADD AL,BL can be used to combine the two BCD nibbles. Take a look at Figure 4-2 to help you visualize this addition.

Another way to combine the two nibbles is with the OR instruction. If you look up the OR instruction in Chapter 6, you will find that it has the format OR destination,source. This instruction ORs each bit in the specified source with the corresponding bit in the specified destination. The result of the ORing is left in the specified destination. Remember from basic logic or the review in Chapter 1 that ORing a bit with a 1 always produces a result of 1. ORing a bit with a 0 leaves the bit unchanged. To set a bit in a word to a 1, then, all you have to do is OR that bit with a word which has a 1 in that bit position and 0's in all the other bit positions. This is similar to the way the AND instruction is used to clear bits in a word to 0's. See the OR instruction description in Chapter 6 for examples of this.

```
                        ; 8086 PROGRAM F4-05.ASM
;ABSTRACT   : Program produces a packed BCD byte from 2 ASCII-encoded digits
                        ; The first ASCII digit (5) is loaded in BL.
                        ; The second ASCII digit (9) is loaded in AL.
;REGISTERS  ; Uses CS, AL, BL, CL
;PORTS      : None used

 0000
CODE    SEGMENT
        ASSUME CS:CODE
 0000  B3 35    START:  MOV  BL, '5'    ; Load first ASCII digit into BL
 0002  B0 39            MOV  AL, '9'    ; Load second ASCII digit into AL
 0004  80 E3 0F         AND  BL, 0FH    ; Mask upper 4 bits of first digit
 0007  24 0F            AND  AL, 0FH    ; Mask upper 4 bits of second digit
 0009  B1 04            MOV  CL, 04H    ; Load CL for 4 rotates required
 000B  D2 C3            ROL  BL, CL     ; Rotate BL 4 bit positions
 000D  0A C3            OR   AL, BL     ; Combine nibbles, result in AL
 000F           CODE    ENDS
                        END START
```

FIGURE 4-5 List file of 8086 assembly language program to produce packed BCD from two ASCII characters.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0's will not be changed. Bits ORed with 1's will become or stay 1's. Again look at Figure 4-2 to help you visualize this operation.

SUMMARY OF BCD PACKING PROGRAM

If you compare the algorithm for this program with the finished program in Figure 4-5, you should see that each step in the algorithm translates to one or two assembly language instructions. As we told you before, developing the assembly language program from a good algorithm is really quite easy because you are simply translating one step at a time to its equivalent assembly language instructions. Also, debugging a program developed in this way is quite easy because you simply single-step or breakpoint your way through it and check the results after each step. In the next section we discuss the 8086 JMP instructions and flags so we can show you how you implement some of the other programming structures in assembly language.

JUMPS, FLAGS, AND CONDITIONAL JUMPS

Introduction

The real power of a computer comes from its ability to choose between two or more sequences of actions based on some condition, repeat a sequence of instructions *as long as* some condition exists, or repeat a sequence of instructions *until* some condition exists. *Flags* indicate whether some condition is present or not. *Jump* instructions are used to tell the computer the address to fetch its next instruction from. Figure 4-6 shows in diagram form the different ways a Jump instruction can direct the 8086 to fetch its next instruction from some place in memory other than the next sequential location.

The 8086 has two types of Jump instructions, conditional and unconditional. When the 8086 fetches and decodes an Unconditional Jump instruction, it always goes to the specified jump destination. You might use this type of Jump instruction at the end of a program so that the entire program runs over and over, as shown in Figure 4-6.

When the 8086 fetches and decodes a Conditional Jump instruction, it evaluates the state of a specified flag to determine whether to fetch its next instruction from the jump destination location or to fetch its next instruction from the next sequential memory location.

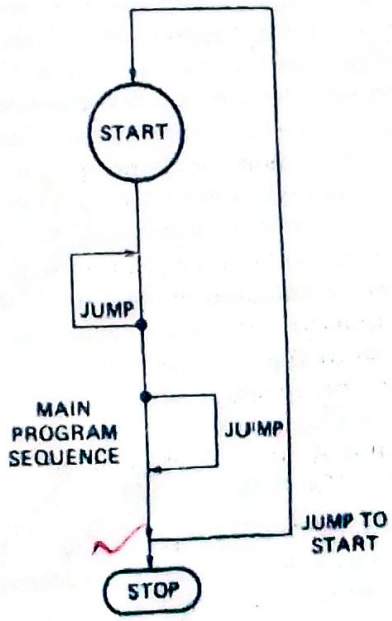

FIGURE 4-6 Change in program flow that can be caused by jump instructions.

```
                        ; 8086 PROGRAM
                        ;ABSTRACT  : This program illustrates a
                        ;REGISTERS : Uses CS, AX
                        ;PORTS     : None used

1
2
3                       CODE    SEGMENT
4                               ASSUME CS:CODE
5                               JMP THERE       ; Skip over a series of instructions
6 0000                          NOP             ; Dummy instructions to represent those
7                               NOP             ; instructions skipped over
8 0000  EB 03 90                MOV AX, 0000H   ; Zero accumulator before addition instructions
9 0003  90              THERE:  NOP             ; Dummy instruction to represent continuation of execution
10 0004 90
11 0005 B8 0000         CODE    ENDS
12 0008 90                      END
13 0009
14
```

FIGURE 4-9 List file of program demonstrating "forward" JMP.

JMP. The displacement is calculated by counting the number of bytes from the next address after the JMP instruction to the destination. If the displacement is negative (backward in the program), then it must be expressed in 2's complement form before it can be written in the instruction code template.

Now let's look at another simple example program, in Figure 4-9, to see how you can jump ahead over a group of instructions in a program. Here again we use a label to give a name to the address that we want to JMP to. We also use NOP instructions to represent the instructions that we want to skip over and the instructions that continue after the JMP. Let's see how this JMP instruction is coded.

When the assembler reads through the source file for this program, it will find the label "THERE" after the JMP mnemonic. At this point the assembler has no way of knowing whether it will need 1 or 2 bytes to represent the displacement to the destination address. The assembler plays it safe by reserving 2 bytes for the displacement. Then the assembler reads on through the rest of the program. When the assembler finds the specified label, it calculates the displacement from the instruction after the JMP instruction to the label. If the assembler finds the displacement to be outside the range of −128 bytes to +127 bytes, then it will code the instruction as a direct within-segment near JMP with 2 bytes of displacement. If the assembler finds the displacement to be within the −128- to +127-byte range, then it will code the instruction as a direct within-segment short-type JMP with a 1-byte displacement. In the latter case, the assembler will put the code for a NOP instruction, 90H, in the third byte it had reserved for the JMP instruction. The instruction codes for the JMP THERE instruction on line 8 of Figure 4-9 demonstrate this. As shown in the instruction template in Figure 4-7, EBH is the basic opcode for the direct within-segment short JMP. The 03H represents the displacement to the JMP destination. Since we are jumping forward in this case, the displacement is a positive number. The 90H in the next memory byte is the code for a NOP instruction. The displacement is calculated from the offset of this NOP instruction, 0002H, to the offset of the destination label, 0005H. The difference of 03H between these two is the displacement you see coded in the instruction.

If you are hand coding a program such as this, you will probably know how far it is to the label, and you can leave just 1 byte for the displacement if that is enough. If you are using an assembler and you don't want to waste the byte of memory or the time it takes to fetch the extra NOP instruction, you can write the instruction as JMP SHORT label. The SHORT operator is a promise to the assembler that the destination will not be outside the range of −128 to +127 bytes. Trusting your promise, the assembler then reserves only 1 byte for the displacement.

Note that if you are making a JMP from an address near the start of a 64-Kbyte segment to an address near the end of the segment, you may not be able to get there with a jump of +32,767. The way you get there is to JMP backward around to the desired destination address. An assembler will automatically do this for you.

One advantage of the direct near- and short-type JMPs is that the destination address is specified *relative* to the address of the instruction after the JMP instruction. Since the JMP instruction in this case does not contain an absolute address or offset, the program can be loaded anywhere in memory and still run correctly. A program which can be loaded anywhere in memory to be run is said to be *relocatable*. You should try to write your programs so that they are relocatable.

Now that you know about unconditional JMP instructions, we will discuss the 8086 flags, so that we can show how the 8086 Conditional Jump instructions are used to implement the rest of the standard programming structures.

The 8086 Conditional Flags

The 8086 has six *conditional flags*. They are the *carry* flag (CF), the *parity* flag (PF), the *auxiliary carry* flag (AF), the *zero* flag (ZF), the *sign* flag (SF), and the *overflow* flag (OF). Chapter 1 shows numerical examples of some of the conditions indicated by these flags. Here we review these conditions and show how some of the important 8086 instructions affect these flags.

THE CARRY FLAG WITH ADD, SUBTRACT, AND COMPARE INSTRUCTIONS

If the addition of two 8-bit numbers produces a sum greater than 8 bits, the carry flag will be set to a 1 to indicate a carry into the next bit position. Likewise, if

the addition of two 16-bit numbers produces a sum greater than 16 bits, then the carry flag will be set to a 1 to indicate that a final carry was produced by the addition.

During subtraction, the carry flag functions as a borrow flag. If the bottom number in a subtraction is larger than the top number, then the carry/borrow flag will be set to indicate that a borrow was needed to perform the subtraction.

The 8086 compare instruction has the format CMP destination,source. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. The comparison is done by subtracting the contents of the specified source from the contents of the specified destination. Flags are updated to reflect the result of the comparison, but neither the source nor the destination is changed. If the source operand is greater than the specified destination operand, then the carry/borrow flag will be set to indicate that a borrow was needed to do the comparison (subtraction). If the source operand is the same size as or smaller than the specified destination operand, then the carry/borrow flag will not be set after the compare. If the two operands are equal, the zero flag will be set to a 1 to indicate that the result of the compare (subtraction) was all 0's. Here's an example and summary of this for your reference.

CMP BX, CX

condition	CF	ZF
CX > BX	1	0
CX < BX	0	0
CX = BX	0	1

The compare instruction is very important because it allows you to easily determine whether one operand is greater than, less than, or the same size as another operand.

THE PARITY FLAG

Parity is a term used to indicate whether a binary word has an even number of 1's or an odd number of 1's. A binary number with an even number of 1's is said to have *even parity*. The 8086 parity flag will be set to a 1 after an instruction if the lower 8 bits of the destination operand has an even number of 1's. Probably the most common use of the parity flag is to determine whether ASCII data sent to a computer over phone lines or some other communications link contains any errors. In Chapter 14 we describe this use of parity.

THE AUXILIARY CARRY FLAG

This flag has significance in BCD addition or BCD subtraction. If a carry is produced when the least significant nibbles of 2 bytes are added, the auxiliary carry flag will be set. In other words, a carry out of bit 3 sets the auxiliary carry flag. Likewise, if the subtraction of the least significant nibbles requires a borrow, the auxiliary carry/borrow flag will be set. The auxiliary carry/borrow flag is used only by the DAA and DAS instructions. Consult the DAA and DAS instruction descriptions in Chapter 6 and the BCD operation examples section of Chapter 1 for further discussion of addition and subtraction of BCD numbers.

THE ZERO FLAG WITH INCREMENT, DECREMENT, AND COMPARE INSTRUCTIONS

As the name implies, this flag will be set to a 1 if the result of an arithmetic or logic operation is zero. For example, if you subtract two numbers which are equal, the zero flag will be set to indicate that the result of the subtraction is zero. If you AND two words together and the result contains no 1's, the zero flag will be set to indicate that the result is all 0's.

Besides the more obvious arithmetic and logic instructions, there are a few other very useful instructions which also affect the zero flag. One of these is the compare instruction CMP, which we discussed previously with the carry flag. As shown there, the zero flag will be set to a 1 if the two operands compared are equal.

Another important instruction which affects the zero flag is the decrement instruction, DEC. This instruction will decrement (or, in other words, subtract 1 from) a number in a specified register or memory location. If, after decrementing, the contents of the register or memory location are zero, the zero flag will be set. Here's a preview of how this is used. Suppose that we want to repeat a sequence of actions nine times. To do this, we first load a register with the number 09H and execute the sequence of actions. We then decrement the register and look at the zero flag to see if the register is down to zero yet. If the zero flag is not set, then we know that the register is not yet down to zero, so we tell the 8086, with a Jump instruction, to go back and execute the sequence of instructions again. The following sections will show many specific examples of how this is done.

The increment instruction, INC destination, also affects the zero flag. If an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result in the destination will be all 0's. The zero flag will be set to indicate this.

THE SIGN FLAG—POSITIVE AND NEGATIVE NUMBERS

When you need to represent both positive and negative numbers for an 8086, you use 2's complement sign-and-magnitude form as described in Chapter 1. In this form, the most significant bit of the byte or word is used as a sign bit. A 0 in this bit indicates that the number is positive. A 1 in this bit indicates that the number is negative. The remaining 7 bits of a byte or the remaining 15 bits of a word are used to represent the magnitude of the number. For a positive number, the magnitude will be in standard binary form. For a negative number, the magnitude will be in 2's complement form. After an arithmetic or logic instruction executes, the sign flag will be a copy of the most significant bit of the destination byte or the destination word. In addition to its use with signed arithmetic operations, the sign flag can be used to determine whether an operand has been decremented beyond zero. Decrementing 00H, for example, will give FFH. Since the MSB of FFH is a 1, the sign flag will be set.

THE OVERFLOW FLAG

This flag will be set if the result of a signed operation is too large to fit in the number of bits available to represent it. To remind you of what *overflow* means, here is an example. Suppose you add the 8-bit signed number 01110101 (+117 decimal) and the 8-bit signed number 00110111 (+55 decimal). The result will be 10101100 (+172 decimal), which is the correct binary result in this case, but is too large to fit in the 7 bits allowed for the magnitude in an 8-bit signed number. For an 8-bit signed number, a 1 in the most significant bit indicates a negative number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

The 8086 Conditional Jump Instructions

As we stated previously, much of the real power of a computer comes from its ability to choose between two courses of action depending on whether some condition is present or not. In the 8086 the six conditional flags indicate the conditions that are present after an instruction. The 8086 Conditional Jump instructions look at the state of a specified flag(s) to determine whether the jump should be made or not.

Figure 4-10 shows the mnemonics for the 8086 Conditional Jump instructions. Next to each mnemonic is a brief explanation of the mnemonic. Note that the terms *above* and *below* are used when you are working with unsigned binary numbers. The 8-bit unsigned number 11000110 is above the 8-bit unsigned number 00111001, for example. The terms *greater* and *less* are used when you are working with signed binary numbers. The 8-bit signed number 00111001 is greater (more positive) than the 8-bit signed number 11000110, which represents a negative number. Also shown in Figure 4-10 is an indication of the flag conditions that will cause the 8086 to do the jump. If the specified flag conditions are not present, the 8086 will just continue on to the next instruction in sequence. In other words, if the jump condition is not met, the Conditional Jump instruction will effectively function as a NOP. Suppose, for example, we have the instruction JC SAVE, where SAVE is the label at the destination address. If the carry flag is set, this instruction will cause the 8086 to jump to the instruction at the SAVE: label. If the carry flag is not set, the instruction will have no effect other than taking up a little processor time.

All conditional jumps are *short-type* jumps. This means that the destination label must be in the same code segment as the jump instruction. Also, the destination address must be in the range of −128 bytes to +127 bytes from the address of the instruction after the Jump instruction. As we show in later examples, it is important to be aware of this limit on the range of conditional jumps as you write your programs.

The Conditional Jump instructions are usually used after arithmetic or logic instructions. They are very commonly used after Compare instructions. For this case, the Compare instruction syntax and the Conditional Jump instruction syntax are such that a little trick makes it very easy to see what will cause a jump to occur. Here's the trick. Suppose that you see the instruction sequence

```
CMP BL, DH
JAE HEATER_OFF
```

in a program, and you want to determine what these instructions do. The CMP instruction compares the byte

MNEMONIC	CONDITION TESTED	"JUMP IF . . ."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Note: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

FIGURE 4-10 8086 Conditional Jump instructions.