```
1                                   ; 8086 PROGRAM F4-05.ASM
2                           ;ABSTRACT   : Program produces a packed BCD byte from 2 ASCII-encoded digits
3                                       ; The first ASCII digit (5) is loaded in BL.
4                                       ; The second ASCII digit (9) is loaded in AL.
5                                       ; The result (packed BCD) is left in AL
6                           ;REGISTERS  ; Uses CS, AL, BL, CL
7                           ;PORTS      : None used
8
9  0000                     CODE    SEGMENT
10                                  ASSUME CS:CODE
11 0000  B3 35              START:  MOV  BL, '5'    ; Load first ASCII digit into BL
12 0002  B0 39                      MOV  AL, '9'    ; Load second ASCII digit into AL
13 0004  80 E3 0F                   AND  BL, 0FH    ; Mask upper 4 bits of first digit
14 0007  24 0F                      AND  AL, 0FH    ; Mask upper 4 bits of second digit
15 0009  B1 04                      MOV  CL, 04H    ; Load CL for 4 rotates required
16 000B  D2 C3                      ROL  BL, CL     ; Rotate BL 4 bit positions
17 000D  0A C3                      OR   AL, BL     ; Combine nibbles, result in AL
18 000F                     CODE    ENDS
19                                  END START
```

Fig. 4.5 List file of 8086 assembly language program to produce packed BCD form two ASCII characters

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 Fig. 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. 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.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0s will not be changed. Bits ORed with 0s will become or stay 1's. Again look at Fig. 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 Fig. 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 Fig. 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.

Let's start by taking a look at how the 8086 Unconditional Jump instruction works.

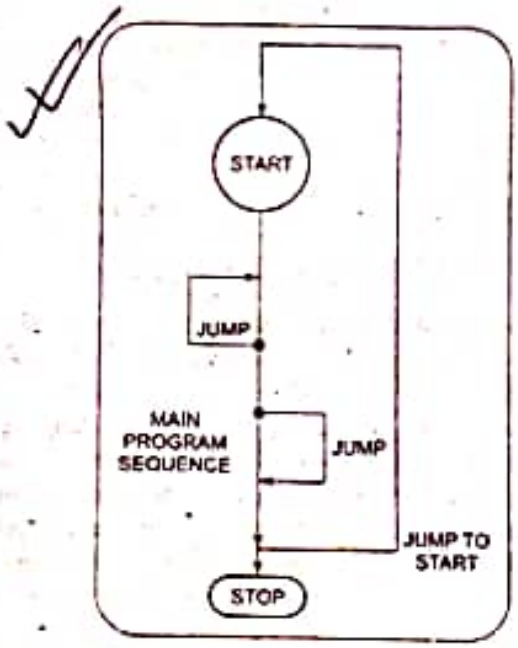

Fig. 4.6 Change in program flow that can be caused by jump instructions

The 8086 Unconditional Jump Instruction

INTRODUCTION

As we said before, Jump instructions can be used to tell the 8086 to start fetching its instructions from some new location rather than from the next sequential location. The 8086 JMP instruction always causes a jump to occur, so this is referred to as an *unconditional* jump.

Remember from previous discussions that the 8086 computes the physical address from which to fetch its next code byte by adding the offset in the instruction pointer register to the code segment base represented by the 16-bit number in the CS register. When the 8086 executes a JMP instruction, it loads a new number into the instruction pointer register, and in some cases it also loads a new number into the code segment register.

If the JMP destination is in the same code segment, the 8086 only has to change the contents of the instruction pointer. This type of jump is referred to as a *near*, or *intrasegment*, jump.

If the JMP destination is in a code segment which has a different name from the segment in which the JMP instruction is located, the 8086 has to change the contents of both CS and IP to make the jump. This type of jump is referred to as *a far*, or *intersegment*, jump.

Near and far jumps are further described as either *direct* or *indirect*. If the destination address for the jump is specified directly as part of the instruction, then the jump is described as *direct*. You can have a direct near jump or a direct far jump. If the destination address for the jump is contained in a register or memory location, the jump is referred to as *indirect* because the 8086 has to go to the specified register or memory location to get the required destination address. You can have an indirect near jump or an indirect far jump.

Figure 4.7 shows the coding templates for the four basic types of unconditional jumps. As you can see, for the direct types, the destination offset, and, if necessary, the segment base are included directly in the instruction. The indirect types of jumps use the second byte of the instruction to tell the 8086 whether the destination offset (and segment base, if necessary) is contained in a register or in memory locations specified with one of the 24 address modes we introduced you to in the last chapter.

The JMP instruction description in Chapter 6 shows examples of each type of jump instruction, but in most of your programs you will use a direct near-type JMP instruction, so in the next section we will discuss in detail how this type works.

UNCONDITIONAL JUMP INSTRUCTION TYPES—OVERVIEW

The 8086 Unconditional Jump instruction, JMP, has five different types. Figure 4.7 shows the names and instruction coding templates for these five types. We will first summarize how these five types work to give you an overview; then we will describe in detail the two types you need for your programs at this point. The JMP instruction description in Chapter 6 shows examples of each of the five types.

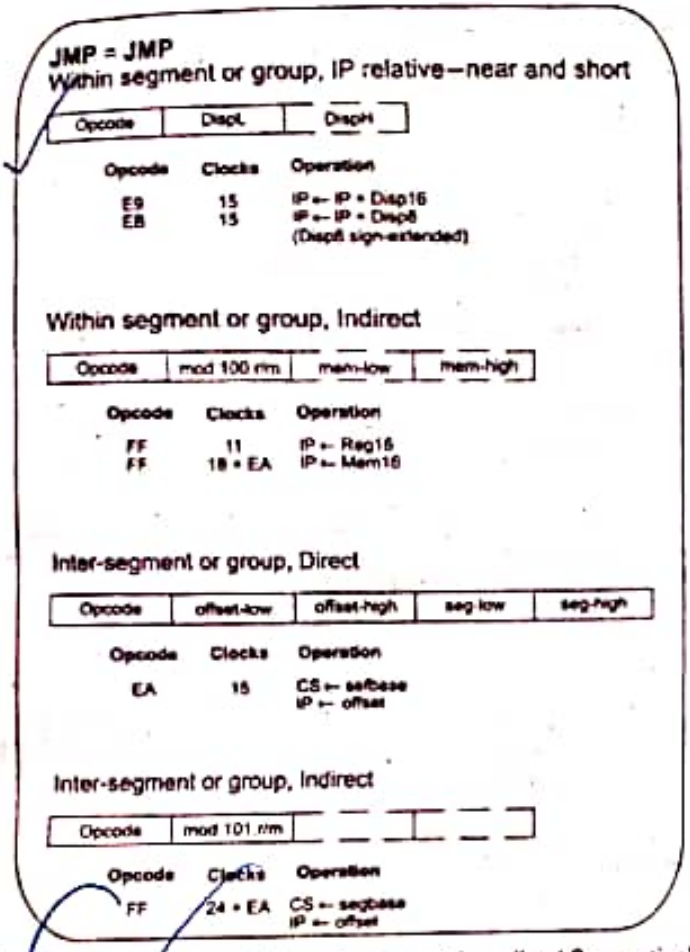

Fig. 4.7 8086 Unconditional Jump instructions. (Intel Corporation)

THE DIRECT NEAR- AND SHORT-TYPE JMP INSTRUCTIONS

As we described previously, a near-type jump instruction can cause the next instruction to be fetched from anywhere in the current code segment. To produce the new instruction fetch address, this instruction adds a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer register. A 16-bit signed displacement means that the jump can be to a location anywhere from +32,767 to −32,768 bytes from the current instruction pointer location. A positive displacement usually means you are jumping ahead in the program, and a negative displacement usually means that you are jumping "backward" in the program.

A special case of the direct near-type jump instruction is the direct short-type jump. If the destination for the jump is within a displacement range of +127 to −128 bytes from the current instruction pointer location, the destination can be reached with just an 8-bit displacement. The coding for this type of jump is shown on the second line of the coding template for the direct near JMP in Fig. 4.7. Only one byte is required for the displacement in this case. Again the 8086 produces the new instruction fetch address by adding the signed 8-bit displacement, contained in the instruction, to the contents of the instruction pointer register. Here are some examples of how you use these JMP instructions in programs.

DIRECT WITHIN-SEGMENT NEAR AND DIRECT WITHIN-SEGMENT SHORT JMP EXAMPLES

Suppose that we want an 8086 to execute the instructions in a program over and over. Figure 4.8 shows how the JMP instruction can be used to do this. In this program, the label BACK followed by a colon is used to give a name to the address we want to jump back to. When the assembler reads this label, it will make an entry in its symbol table indicating where it found the label. Then, when the assembler reads the JMP instruction and finds the name BACK in the instruction, it will be able to calculate the displacement from the jump instruction to the label. This displacement will be inserted as part of the code for the instruction. Even if you are not using an assembler, you should use labels to indicate jump destinations so that you can easily see them. The NOP instructions used in the program in Fig. 4.8 do nothing except fill space. We used them in this example to represent the instructions that we want to loop through over and over. Once the 8086 gets into the JMP-BACK loop, the only ways it can get out are if the power is turned off, an interrupt occurs, or the system is reset.

Now let's see how the binary code for the JMP instruc-tion in Fig. 4.8 is constructed. The jump is to a label in the same segment, so this narrows our choices down to the first three types of JMP instruction shown in Fig. 4.7. For several reasons, it is best to use the direct-type JMP instruction whenever possible. This narrows our choices down to the first two types in Fig. 4.7. The choice between these two is determined by whether you need a 1-byte or a 2-byte displacement to reach the JMP destination address. Since for our example program the destination address is within the range of −128 to +127 bytes from the instruction after the JMP instruction, we can use the direct within-segment short type of JMP. According to Fig. 4.7, the instruction template for this instruction is 11101011 (EBH) followed by a displacement. Here's how you calculate the displacement to put in the instruction.

NOTE: An assembler does this for you automatically, but you should still learn how it is done to help you in troubleshooting.

The numbers in the left column of Fig. 4.8 represent the offset of each code byte from the code segment base. These are the numbers that will be in the instruction pointer as the program executes. After the 8086 fetches an instruction byte, it automatically increments the instruction pointer to point to the next instruction byte. The displacement in the JMP instruction will then be added to the offset of the next in-line instruction after the JMP instruction. For the example program in Fig. 4.8, the displacement in the JMP instruction will be added to offset 0006H, which is in the instruction pointer

```
1                                       ; 8086 PROGRAM    F4-08.ASM
2                                ;ABSTRACT  : This program illustrates a "backwards" jump
3                                ;REGISTERS : Uses CS, AL
4                                ;PORTS     : None used
5
6  0000                          CODE   SEGMENT
7                                       ASSUME  CS:CODE
8  0000  04 03                   BACK:  ADD AL, 03H  ; Add 3 to total
9  0002  90                              NOP         ; Dummy instructions to represent those
10 0003  90                              NOP         ; instructions jumped back over
11 0004  EB FA                           JMP BACK    ; Jump back over instructions to BACK label
12 0006                          CODE   ENDS
13                                      END
```

Fig. 4.8 List file of program demonstrating "backward" JMP.

```
1                                       ; 8086 PROGRAM     F4-09.ASM
2                                ;ABSTRACT  : This program illustrates a "forwards" jump
3                                ;REGISTERS : Uses CS, AX
4                                ;PORTS     : None used
5
6  0000                          CODE   SEGMENT
7                                       ASSUME CS:CODE
8  0000  EB 03 90                       JMP THERE     ; Skip over a series of instructions
9  0003  90                             NOP           ; Dummy instructions to represent those
10 0004  90                             NOP           ; instructions skipped over
11 0005  B8 0000                THERE: MOV AX, 0000H ; Zero accumulator before addition instructions
12 0008  90                             NOP           ; Dummy instruction to represent continuation of execution
13 0009                          CODE   ENDS
14                                      END
```

Fig. 4.9 List file of program demonstrating "forward" JMP.

after the JMP instruction executes. What this means is that when you are counting the number of bytes of displacement, you always start counting from the address of the instruction immediately after the JMP instruction. For the example program, we want to jump from offset 0006H back to offset 0000H. This is a displacement of -6H.

You can't, however, write the displacement in the instruction as -6H. Negative displacements must be expressed in 2's complement, sign-and-magnitude form. First, write the number as an 8-bit positive binary number. In this case, that is 00000110. Then, invert each bit of this, including the sign bit, to give 11111001. Finally, add 1 to that result to give 11111010 binary or FAH, which is the correct 2's complement representation for -6H. As shown on line 11 in the assembler listing for the program in Fig. 4.8, the two code bytes for this JMP instruction then are EBH and FAH.

To summarize this example, then, a label is used to give a name to the destination address for the jump. This name is used to refer to the destination address in the JMP instruction. Since the destination in this example is within the range of -128 to +127 bytes from the address after the JMP instruction, the instruction can be coded as a direct within-segment short-type 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 Fig. 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 Fig. 4.9 demonstrate this. As shown in the instruction template in Fig. 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 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. 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 operation 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 Fig. 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

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	(CFor 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.

Fig. 4.10 8086 Conditional Jump instructions.

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 in the DH register with the byte in the BL register and sets flags according to the result. A previous section showed you how the carry and zero flags are affected by a Compare instruction. According to Fig. 4.10, the JAE instruction says, "Jump if above or equal" to the label HEATER_OFF. The question now is, will it jump if BL is above DH, or will it jump if DH is above BL? You could determine how the flags will be affected by the comparison and use Fig. 4.10 to answer the question, but an easier way is to mentally read parts of the Compare instruction between parts of the Jump instruction. If you read the example sequence as "Jump if BL is above or equal to DH," the meaning of the sequence is immediately clear. As you write your own programs, thinking of a conditional sequence in this way should help you to choose the right Conditional Jump instruction. The next sections show you how we use Conditional and Unconditional Jump instructions to implement some of the standard program structures and solve some common programming problems.

IF-THEN, IF-THEN-ELSE, AND MULTIPLE IF-THEN-ELSE PROGRAMS

IF-THEN Programs

The IF-THEN structure has the format

```
IF condition THEN
    action
    action
```

This structure says that IF the stated condition is found to be true, the series of actions following THEN will be executed. If the condition is false, execution will skip over the actions after the THEN and proceed with the next mainline instruction.