d. `NEG AL` where AL contains 7Fh

e. `XCHG AX,BX` where AX contains 1ABCh and BX contains 712Ah

f. `ADD AL,BL` where AL contains 80h and BL contains FFh

g. `SUB AX,BX` where AX contains 0000h and BX contains 8000h

h. `NEG AX` where AX contains 0001h

2. a. Suppose that AX and BX both contain positive numbers, and ADD AX,BX is executed. Show that there is a carry into the msb but no carry out of the msb if, and only if, signed overflow occurs.

   b. Suppose AX and BX both contain negative numbers, and ADD AX,BX is executed. Show that there is a carry out of the msb but no carry into the msb if, and only if, signed overflow occurs.

3. Suppose ADD AX,BX is executed. In each of the following parts, the first number being added is the contents of AX, and the second number is the contents of BX. Give the resulting value of AX and tell whether signed or unsigned overflow occurred.

   a. 512Ch
   + 4185h

   b. FE12h
   + 1ACBh

   c. E1E4h
   + DAB3h

   d. 7132h
   + 7000h

   e. 6389h
   + 1176h

4. Suppose SUB AX,BX is executed. In each of the following parts, the first number is the initial contents of AX and the second number is the contents of BX. Give the resulting value of AX and tell whether signed or unsigned overflow occurred.

   a. 2143h
   − 1986h

   b. 81FEh
   − 1986h

   c. 19BCh
   − 81FEh

   d. 0002h
   − FE0Fh

   e. 8BCDh
   − 71ABh

6

Flow Control Instructions

Overview

For assembly language programs to carry out useful tasks, there must be a way to make decisions and repeat sections of code. In this chapter we show how these things can be accomplished with the jump and loop instructions.

The jump and loop instructions transfer control to another part of the program. This transfer can be unconditional or can depend on a particular combination of status flag settings.

After introducing the jump instructions, we'll use them to implement high-level language decision and looping structures. This application will make it much easier to convert a pseudocode algorithm to assembly code.

6.1 An Example of a Jump

To get an idea of how the jump instructions work, we will write a program to display the entire IBM character set.

Program Listing PGM6_1.ASM

```
1:  TITLE   PGM6_1: IBM CHARACTER DISPLAY
2:  .MODEL          SMALL
3:  .STACK          100H
4:  .CODE
5:  MAIN            PROC
6:          MOV     AH,2        ;display char function
7:          MOV     CX,256      ;no. of chars to display
8:          MOV     DL,0        ;DL has ASCII code of null cha
9:  PRINT_LOOP:
```

```
10:           INT    21h           ;display a char
11:           INC    DL            ;increment ASCII code
12:           DEC    CX            ;decrement counter
13:           JNZ    PRINT_LOOP    ;keep going if CX not 0
14: ;DOS exit
15:           MOV    AH,4CH
16:           INT    21h
17: MAIN      ENDP
18:           END    MAIN
```

There are 256 characters in the IBM character set. Those with codes 32 to 127 are the standard ASCII display characters introduced in Chapter 2. IBM also provides a set of graphics characters with codes 0 to 31 and 128 to 255.

To display the characters, we use a loop (lines 9 to 13). Before entering the loop, AH is initialized to 2 (single-character display) and DL is set to 0, the initial ASCII code. CX is the loop counter; it is set to 256 before entering the loop and is decremented after each character is displayed.

The instruction that controls the loop is JNZ (Jump if Not Zero). If the result of the preceding instruction (DEC CX) is not zero, then the JNZ instruction transfers control to the instruction at label PRINT_LOOP. When CX finally contains 0, the program goes on to execute the DOS return instructions. Figure 6.1 shows the output of the program. Of course, the ASCII codes of backspace, carriage return, and so on cause a control function to be performed, rather than displaying a symbol.

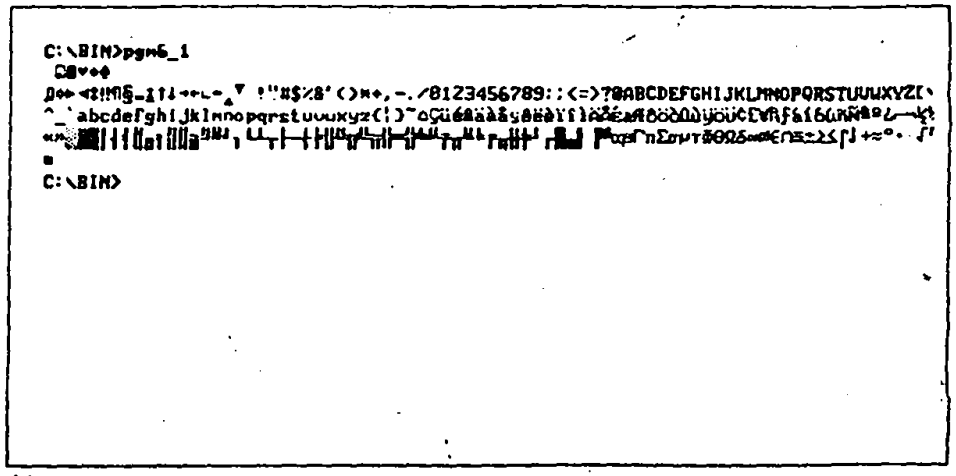
Note: PRINT_LOOP is the first statement label we've used in a program. Labels are needed in situations where one instruction refers to another, as is the case here. Labels end with a colon, and to make labels stand out, they are usually placed on a line by themselves. If so, they refer to the instruction that follows.

6.2 Conditional Jumps

JNZ is an example of a **conditional jump instruction**. The syntax is

```
Jxxx                destination_label
```

Figure 6.1 Output of PGM6_1



If the condition for the jump is true, the next instruction to be executed is the one at destination_label, which may precede or follow the jump instruction itself. If the condition is false, the instruction immediately following the jump is done next. For JNZ, the condition is that the result of the previous operation is not zero.

Range of a Conditional Jump

The structure of the machine code of a conditional jump requires that destination_label must precede the jump instruction by no more than 126 bytes, or follow it by no more than 127 bytes (we'll show how to get around this restriction later).

How the CPU Implements a Conditional Jump

To implement a conditional jump, the CPU looks at the FLAGS register. You already know it reflects the result of the last thing the processor did. If the conditions for the jump (expressed as a combination of status flag settings) are true; the CPU adjusts the IP to point to the destination label, so that the instruction at this label will be done next. If the jump condition is false, then IP is not altered; this means that the next instruction in line will be done.

In the preceding program, the CPU executes JNZ PRINT_LOOP by inspecting ZF. If ZF = 0, control transfers to PRINT_LOOP; if ZF = 1, the program goes on to execute MOV AH,4CH.

Table 6.1 shows the conditional jumps. There are three categories: (1) the **signed jumps** are used when a signed interpretation is being given to results, (2) the **unsigned jumps** are used for an unsigned interpretation, and (3) the **single-flag jumps**, which operate on settings of individual flags. *Note:* the jump instructions themselves do not affect the flags.

The first column of Table 6.1 gives the opcodes for the jumps. Many of the jumps have two opcodes; for example, JG and JNLE. Both opcodes produce the same machine code. Use of one opcode or its alternate is usually determined by the context in which the jump appears.

The CMP Instruction

The jump condition is often provided by the **CMP** *(compare)* instruction. It has the form

```
CMP  destination, source
```

This instruction compares destination and source by computing destination contents minus source contents. The result is not stored, but the flags are affected. The operands of CMP may not both be memory locations. Destination may not be a constant. *Note:* CMP is just like SUB, except that destination is not changed.

For example, suppose a program contains these lines:

```
CMP  AX,BX
JG   BELOW
```

where AX = 7FFFh, and BX = 0001. The result of CMP AX,BX is 7FFFh – 0001h = 7FFEh. Table 6.1 shows that the jump condition for JG is satisfied, because ZF = SF = OF = 0, so control transfers to label BELOW.

Table 6.1 Conditional Jumps

Signed Jumps

Symbol	*Description*	*Condition for Jumps*
JG/JNLE	jump if greater than jump if not less than or equal to	ZF = 0 and SF = OF
JGE/JNL	jump if greater than or equal to jump if not less than or equal to	SF = OF
JL/JNGE	jump if less than jump if not greater than or equal	SF <> OF
JLE/JNG	jump if less than or equal jump if not greater than	ZF = 1 or SF <> OF

Unsigned Conditional Jumps

Symbol	*Description*	*Condition for Jumps*
JA/JNBE	jump if above jump if not below or equal	CF = 0 and ZF = 0
JAE/JNB	jump if above or equal jump if not below	CF = 0
JB/JNAE	jump if below jump if not above or equal	CF = 1
JBE/JNA	jump if equal jump if not above	CF = 1 or ZF = 1

Single-Flag Jumps

Symbol	*Description*	*Condition for Jumps*
JE/JZ	jump if equal jump if equal to zero	ZF = 1
JNE/JNZ	jump if not equal jump if not zero	ZF = 0
JC	jump if carry	CF = 1
JNC	jump if no carry	CF = 0
JO	jump if overflow	OF = 1
JNO	jump if no overflow	OF = 0
JS	jump if sign negative	SF = 1
JNS	jump if nonnegative sign	SF = 0
JP/JPE	jump if parity even	PF = 1
JNP/JPO	jump if parity odd	PF = 0

Interpreting the Conditional Jumps

In the example just given, we determined by looking at the flags after CMP was executed that control transfers to label BELOW. This is how the CPU implements a conditional jump. But it's not necessary for a programmer to think about the flags; you can just use the name of the jump to decide if control transfers to the destination label. In the following,

```
CMP   AX,BX
JG    BELOW
```

if AX is greater than BX (in a signed sense), then JG (jump if greater than) transfers to BELOW.

Even though CMP is specifically designed to be used with the conditional jumps, they may be preceded by other instructions, as in PGM6_1. Another example is

```
DEC   AX
JL    THERE
```

Here, if the contents of AX, in a signed sense, is less than 0, control transfers to THERE.

Signed Versus Unsigned Jumps

Each of the signed jumps corresponds to an analogous unsigned jump; for example, the signed jump JG and the unsigned jump JA. Whether to use a signed or unsigned jump depends on the interpretation being given. In fact, Table 6.1 shows that these jumps operate on different flags: the signed jumps operate on ZF, SF, and OF, while the unsigned jumps operate on ZF and CF. Using the wrong kind of jump can lead to incorrect results.

For example, suppose we're giving a signed interpretation. If AX = 7FFFh, BX = 8000h, and we execute

```
CMP   AX,BX
JA    BELOW
```

then even though 7FFFh > 8000h in a signed sense, the program does not jump to BELOW. The reason is that 7FFFh < 8000h in an unsigned sense, and we are using the unsigned jump JA.

Working with Characters

In working with the standard ASCII character set, either signed or unsigned jumps may be used, because the sign bit of a byte containing a character code is always zero. However, unsigned jumps should be used when comparing extended ASCII characters (codes 80h to FFh).

Example 6.1 Suppose AX and BX contain signed numbers. Write some code to put the biggest one in CX.

Solution:

```
        MOV   CX,AX          ;put AX in CX
        CMP   BX,CX          ;is BX bigger?
        JLE   NEXT           ;no, go on
        MOV   CX,BX          ;yes, put BX in CX
NEXT:
```

6.3 The JMP Instruction

The **JMP** *(jump)* instruction causes an unconditional transfer of control **(unconditional jump)**. The syntax is

```
JMP  destination
```

where destination is usually a label in the same segment as the JMP itself (see Appendix F for a more general description).

JMP can be used to get around the range restriction of a conditional jump. For example, suppose we want to implement the following loop:

```
TOP:

;body of the loop
DEC  CX                ;decrement counter
JNZ  TOP               ;keep looping if CX > 0
MOV  AX,BX
```

and the loop body contains so many instructions that label TOP is out of range for JNZ (more than 126 bytes before JMP TOP). We can do this:

```
TOP:

;body of the loop

                DEC  CX         ;decrement counter
                JNZ  BOTTOM     ;keep looping if CX > 0
                JMP  EXIT
BOTTOM:
                JMP  TOP
EXIT:
                MOV  AX,BX
```

6.4 High-Level Language Structures

We've shown that the jump instructions can be used to implement branches and loops. However, because the jumps are so primitive, it is difficult, especially for beginning programmers, to code an algorithm with them without some guidelines.

Because you have probably had some experience with high-level language constructs—such as the IF-THEN-ELSE decision structure or WHILE loops—we'll show how these structures can be simulated in assembly language. In each case, we will first express the structure in a high-level pseudocode.

6.4.1 Branching Structures

In high-level languages, branching structures enable a program to take different paths, depending on conditions. In this section, we'll look at three structures.

IF-THEN

The IF-THEN structure may be expressed in pseudocode as follows:

```
IF condition is true,
 THEN
  execute true-branch statements
END_IF
```

See Figure 6.2.

The *condition* is an expression that is true or false. If it is true, the true-branch statements are executed. If it is false, nothing is done, and the program goes on to whatever follows.

Example 6.2 Replace the number in AX by its absolute value.

Solution: A pseudocode algorithm is

```
IF AX < 0
 THEN
  replace AX by -AX
END_IF
```

It can be coded as follows:

```
;if AX < 0
                CMP   AX,0        ;AX < 0 ?
                JNL   END_IF      ;no, exit
;then
                NEG   AX          ;yes, change sign
END_IF:
```

The condition AX < 0 is expressed by CMP AX,0. If AX is not less than 0, there is nothing to do, so we use a JNL (jump if not less) to jump around the NEG AX. If condition AX < 0 is true, the program goes on to execute NEG AX.

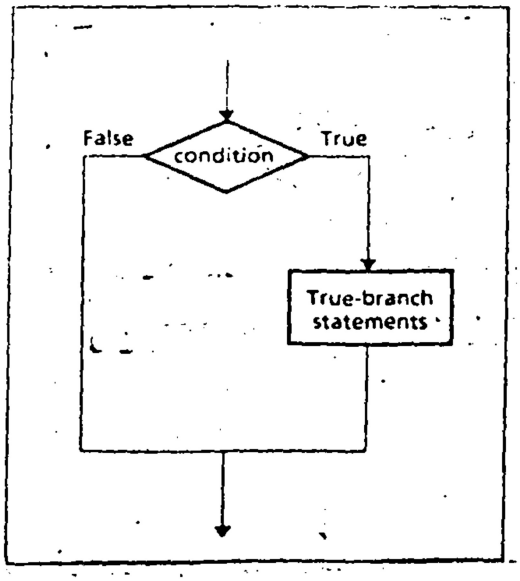


Figure 6.2 IF-THEN

IF-THEN-ELSE

```
IF condition is true
 THEN
  execute true-branch statements
 ELSE
  execute false-branch statements
END_IF
```

See Figure 6.3.

In this structure, if *condition* is true, the true-branch statements are executed. If *condition* is false, the false-branch statements are done.

Example 6.3 Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

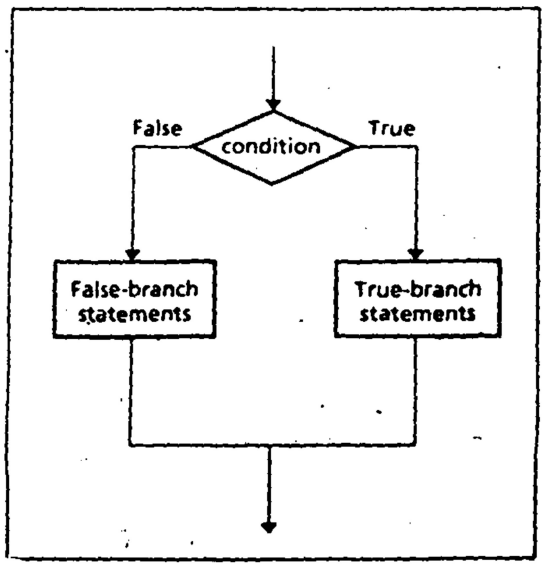
Solution:

```
IF AL <= BL
 THEN
  display the character in AL
 ELSE
  display the character in BL
END_IF
```

It can be coded like this:

```
                MOV  AH,2       ;prepare to display
;if AL <= BL
                CMP  AL,BL      ;AL <= BL?
                JNBE ELSE_      ;no, display char in BL
;then                           ;AL <= BL
                MOV DL,AL       ;move char to be displayed
                JMP DISPLAY     ;go to display
ELSE_:                          ;BL < AL
                MOV DL,BL
```

Figure 6.3 IF-THEN-ELSE



```
DISPLAY:
                INT 21h         ;display it
END_IF
```

Note: the label ELSE_ is used because ELSE is a reserved word.

The condition AL <= BL is expressed by CMP AL,BL. If it's false, the program jumps around the true-branch statements to ELSE_. We use the unsigned jump JNBE (jump if not below or equal), because we're comparing extended characters.

If AL <= BL is true, the true-branch statements are done. Note that JMP DISPLAY is needed to skip the false branch. This is different from the high-level language IF-THEN-ELSE, in which the false-branch statements are automatically skipped if the true-branch statements are done.

CASE

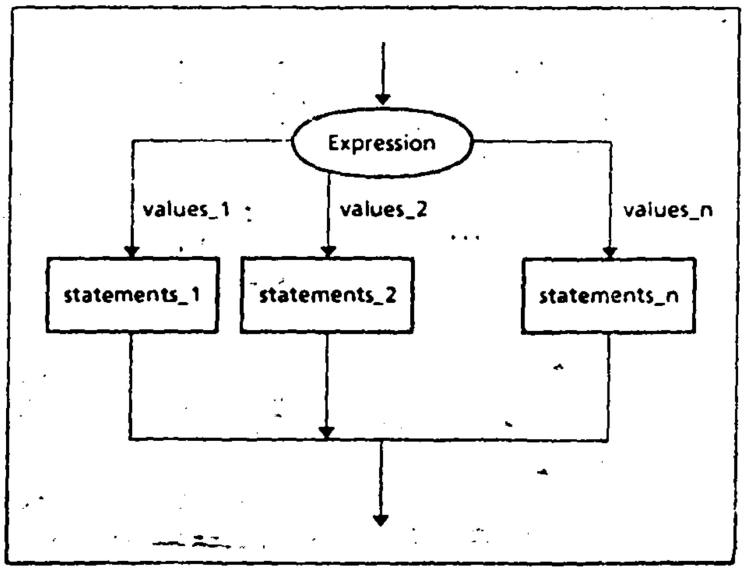
A CASE is a multiway branch structure that tests a register, variable, or expression for particular values or a range of values. The general form is as follows:

```
CASE expression
   values_1: statements_1
   values_2: statements_2
   .
   .
   .
   values_n: statements_n
END_CASE
```

See Figure 6.4.
In this structure, expression is tested; if its value is a member of the set values_i, then statements_i are executed. We assume that sets values_1,..,values_n are disjoint.

Example 6.4 If AX contains a negative number, put -1 in BX; if AX contains 0, put 0 in BX; if AX contains a positive number, put 1 in BX.

Figure 6.4 CASE



Solution:

```
CASE AX
   <0: put -1 in BX
   =0: put 0 in BX
   >0: put 1 in BX
END_CASE
```

It can be coded as follows:

```
;case AX
                CMP AX,0        ;test ax
                JL  NEGATIVE    ;AX < 0
                JE  ZERO        ;AX = 0
                JG  POSITIVE    ;AX > 0
NEGATIVE:
                MOV  BX,-1      ;put -1 in BX
                JMP  END_CASE   ;and exit
ZERO:
                MOV  BX,0       ;put 0 in BX
                JMP  END_CASE   ;and exit
POSITIVE:
                MOV  BX,1       ;put 1 in BX
END_CASE:
```

Note: only one CMP is needed, because jump instructions do not affect the flags.

Example 6.5 If AL contains 1 or 3, display "o"; if AL contains 2 or 4, display "e".

Solution:

```
CASE AL
   1,3: display 'o'
   2,4: display 'e'
END_CASE
```

The code is

```
;case AL
; 1,3:
                CMP  AL,1       ;AL = 1?
                JE   ODD        ;yes, display 'o'
                CMP  AL,3       ;AL = '3'
                JE   ODD        ;yes, display 'o'
; 2,4:
                CMP  AL,2       ;AL = 2?
                JE   EVEN       ;yes, display 'e'
                CMP  AL,4       ;AL = 4?
                JE   EVEN       ;yes, display 'e'
                JMP  END_CASE   ;not 1,3,4
ODD:                            ;display 'o'
                MOV  DL,'o'     ;get 'o'
                JMP  DISPLAY    ;go to display
EVEN:                           ;display 'e'
                MOV  DL,'e'     ;get 'e'
DISPLAY:
```

```
                MOV   AH,2
                INT   21H          ;display char
END_CASE:
```

Branches with Compound Conditions

Sometimes the branching condition in an IF or CASE takes the form

```
condition_1 AND condition_2
```

or

```
condition_1 OR condition_2
```

where condition_1 and condition_2 are either true or false. We will refer to the first of these as an **AND condition** and to the second as an **OR condition**.

AND Conditions

An AND condition is true if and only if condition_1 and condition_2 are both true. Likewise, if either condition is false, then the whole thing is false.

Example 6.6 Read a character, and if it's an uppercase letter, display it.

Solution:

```
Read a character (into AL)
IF ('A' <= character) and (character <= 'Z')
 THEN
  display character
END_IF
```

To code this, we first see if the character in AL follows "A" (or is "A") in the character sequence. If not, we can exit. If so, we still must see if the character precedes "Z" (or is "Z") before displaying it. Here is the code:

```
;read a character
                MOV   AH,1         ;prepare to read
                INT   21H          ;char in AL
;if ('A' <= char) and (char <= 'Z')
                CMP   AL,'A'       ;char >= 'A'?
                JNGE  END_IF       ;no, exit
                CMP   AL,'Z'       ;char <= 'Z'?
                JNLE  END_IF       ;no, exit
;then display char
                MOV   DL,AL        ;get char
                MOV   AH,2         ;prepare to display
                INT   21H          ;display char
END_IF:
```

OR Conditions

Condition_1 OR condition_2 is true if at least one of the conditions is true; it is only false when both conditions are false.

Example 6.7 Read a character. If it's "y" or "Y", display it; otherwise, terminate the program.

Solution:

```
Read a character (into AL)
IF (character = 'y') OR (character = 'Y')
 THEN
  display it
 ELSE
  terminate the program
END_IF
```

To code this, we first see if character = "y". If so, the OR condition is true and we can execute the THEN statements. If not, there is still a chance the OR condition will be true. If character = "Y", it will be true, and we execute the THEN statements; if not, the OR condition is false and we do the ELSE statements. Here is the code:

```
;read a character
                MOV  AH,1       ;prepare to read
                INT  21H        ;char in AL
;if (character = 'y') or (character = 'Y')
                CMP  AL,'y'     ;char = 'y'?
                JE   THEN       ;yes, go to display it
                CMP  AL,'Y'     ;char = 'Y'?
                JE   THEN       ;yes, go to display it
                JMP  ELSE_      ;no, terminate
THEN:
                MOV  AH,2       ;prepare to display
                MOV  DL,AL      ;get char
                INT  21H        ;display it
                JMP  END_IF     ;and exit
ELSE_:
                MOV  AH,4CH
                INT  21H        ;DOS exit
END_IF:
```

6.4.2 Looping Structures

A **loop** is a sequence of instructions that is repeated. The number of times to repeat may be known in advance, or it may depend on conditions

FOR LOOP

This is a loop structure in which the loop statements are repeated a known number of times (a count-controlled loop). In pseudocode,

```
FOR  loop_count times DO
  statements
END_FOR
```

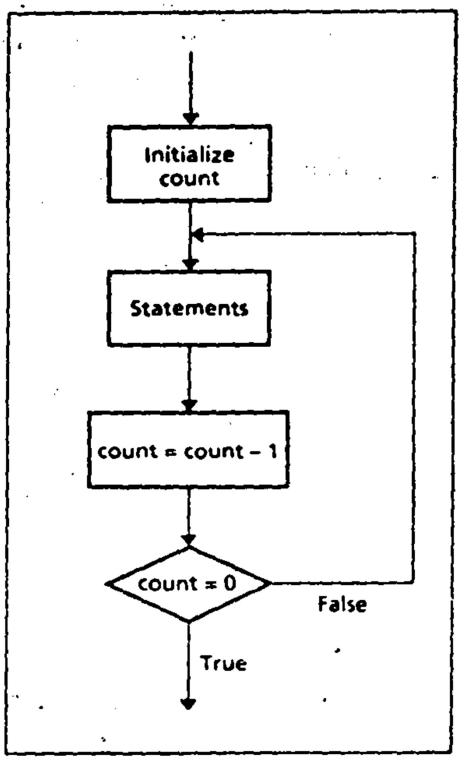
See Figure 6.5.

The **LOOP** instruction can be used to implement a FOR loop. It has the form

```
LOOP  destination_label
```

The counter for the loop is the register CX which is initialized to loop_count. Execution of the LOOP instruction causes CX to be decremented automatically,

Figure 6.5 FOR LOOP



and if CX is not 0, control transfers to destination_label. If CX = 0, the next instruction after LOOP is done. Destination_label must precede the LOOP instruction by no more than 126 bytes.

Using the instruction LOOP, a FOR loop can be implemented as follows:

```
                ;initialize CX to loop_count
TOP:
                ;body of the loop
                LOOP TOP
```

Example 6.8 Write a count-controlled loop to display a row of 80 stars.

Solution:

```
FOR 80 times DO
                display '*'
END_FOR
```

The code is

```
                MOV  CX,80      ;number of stars to display
                MOV  AH,2       ;display character function
                MOV  DL,'*'     ;character to display
TOP:
                INT  21h        ;display a star
                LOOP TOP        ;repeat 80 times
```

You may have noticed that a FOR loop, as implemented with a LOOP instruction, is executed at least once. Actually, if CX contains 0 when the loop is entered, the LOOP instruction causes CX to be decremented to FFFFh, and

the loop is then executed FFFFh = 65535 more times! To prevent this, the struction **JCXZ** *(jump if CX is zero)* may be used before the loop. Its syntax

```
JCXZ            destination_label
```

If CX contains 0, control transfers to the destination label. So a loop i plemented as follows is bypassed if CX is 0:

```
                JCXZ  SKIP
TOP:
                ;body of the loop
                LOOP  TOP
SKIP:
```

WHILE LOOP

This loop depends on a condition. In pseudocode,

```
WHILE condition DO
  statements
END_WHILE
```

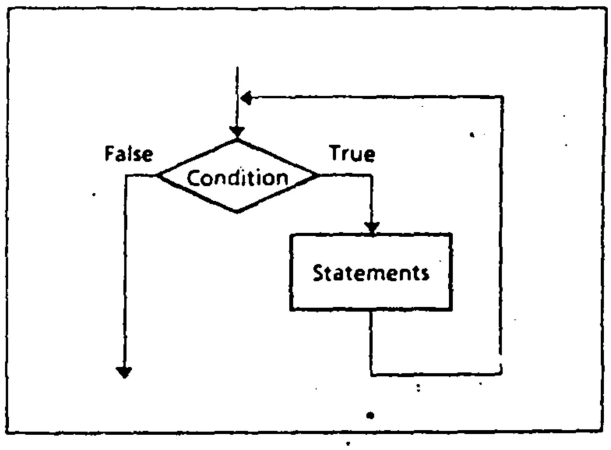
See Figure 6.6.
The *condition* is checked at the top of the loop. If true, the sta ments are executed; if false, the program goes on to whatever follows. It is p ssible that the *condition* will be false initially, in which case the loop body is n t execu e at all. The loop executes as long as the *condition* is true.

Example 6.9 Write some code to count the number of chara ters in an input line.

Solution:

```
initialize count to 0
read a character
WHILE character <> carriage_return DO
 count = count + 1
 read a character
END_WHILE
```

Figure 6.6 WHILE LOOP



The code is

```
                MOV   DX,0       ;DX counts characters
                MOV   AH,1       ;prepare to read
                INT   21H        ;character in AL
WHILE_:
                CMP   AL,0DH     ;CR?
                JE    END_WHILE  ;yes, exit
                INC   DX         ;not CR, increment count
                INT   21H        ;read a character
                JMP   WHILE_     ;loop back
END_WHILE:
```

Note that because a WHILE loop checks the terminating condition at the top of the loop, you must make sure that any variables involved in the condition are initialized before the loop is entered. So you read a character before entering the loop, and read another one at the bottom. The label WHILE_: is used because WHILE is a reserved word.

REPEAT LOOP

Another conditional loop is the REPEAT LOOP. In pseudocode,

```
REPEAT
statements
UNTIL condition
```

See Figure 6.7.

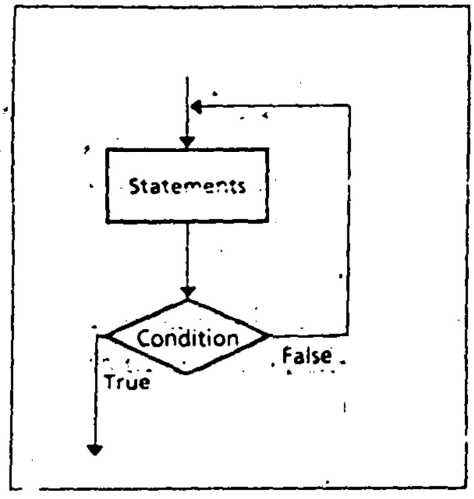
In a REPEAT . . . UNTIL loop, the statements are executed, and then the *condition* is checked. If true, the loop terminates; if false, control branches to the top of the loop.

Example 6.10 Write some code to read characters until a blank is read.

Solution:

```
REPEAT
 read a character
UNTIL character is a blank
```

Figure 6.7 REPEAT-LOOP



The code is

```
                MOV   AH,1        ;prepare to read
REPEAT:
                INT   21H         ;char in AL
;until
                CMP   AL,' '      ;a blank?
                JNE   REPEAT      ;no, keep reading
```

WHILE Versus REPEAT

In many situations where a conditional loop is needed, use of a WHILE loop or a REPEAT loop is a matter of personal preference. The advantage of a WHILE is that the loop can be bypassed if the terminating condition is initially false, whereas the statements in a REPEAT must be done at least once. However, the code for a REPEAT loop is likely to be a little shorter because there is only a conditional jump at the end, but a WHILE loop has two jumps: a conditional jump at the top and a JMP at the bottom.

6.5 Programming with High-Level Structures

To show how a program may be developed from high-level pseudocode to assembly code, let's solve the following problem.

Problem

Prompt the user to enter a line of text. On the next line, display the capital letter entered that comes first alphabetically and the one that comes last. If no capital letters are entered, display "No capital letters". The execution should look like this:

```
Type a line of text:
THE QUICK BROWN FOX JUMPED.
First capital = B Last capital = X
```

To solve this problem, we will use the method of **top-down program design** that you may have encountered in high-level language programming. In this method, the original problem is solved by solving a series of subproblems, each of which is easier to solve than the original problem. Each subproblem is in turn broken down further until we reach a level of subproblems that can be coded directly. The use of procedures (Chapter 8) may enhance this method.

First refinement

1. Display the opening message.
2. Read and process a line of text.
3. Display the results.

Step 1. Display the opening message.

This step can be coded immediately

```
MOV  AH,9          ;display string function
LEA  DX,PROMPT     ;get opening message
INT  21H           ;display it
```

The message will be stored in the data segment as

```
PROMPT  DB      'Type a line of text:',0DH,0AH,'$'
```

We include a carriage return and line feed to move the cursor to the next line so the user can type a full line of text.

Step 2. Read and process a line of text.

This step does most of the work in the program. It takes input from the keyboard, and returns the first and last capital letters read (it should also indicate if no capitals were read). Here is a breakdown:

```
Read a character
WHILE character is not a carriage return DO
IF character is a capital letter (*)
 THEN
  IF character precedes first capital
   THEN
    first capital = character
  END_IF
  IF character follows last capital
   THEN
    last capital = character
   END_IF
END_IF
Read a character
END_WHILE
```

Line (*) is actually an AND condition:

```
IF ('A' <= character) AND (character <= 'Z')
```

Step 2 can be coded as follows:

```
              MOV  AH,1       ;read char function
              INT  21H        ;char in AL
WHILE_:
;while character is not a carriage return do
              CMP  AL,0DH     ;CR?
              JE   END_WHILE  ;yes, exit
;if character is a capital letter
              CMP  AL,'A'     ;char >= 'A'?
              JNGE END_IF     ;not a capital letter
              CMP  AL,'Z'     ;char <= 'Z'?
              JNLE END_IF     ;not a capital letter
;then
; if character precedes first capital
              CMP  AL,FIRST   ;char < FIRST?
              JNL  CHECK_LAST ;no, >=
;then first capital = character
              MOV  FIRST,AL   ;FIRST = char
; end_if
CHECK_LAST:
```

```
;'if character follows last capital
                CMP   AL,LAST    ;char > LAST?
                JNG   END_IF     ;no, <=
;then last capital = character
                MOV   LAST,AL    ;LAST = char
; end_if
END_IF:
;read a character
                INT   21H        ;char in AL
                JMP   WHILE_     ;repeat loop
END_WHILE:
```

Variables FIRST and LAST must have values before the WHILE loop is executed the first time. They can be initialized in the data segment as follows:

```
FIRST           DB              ']'
LAST            DB              '@'
```

The initial values "]" and "@" were chosen because "]" follows "Z" in the ASCII sequence, and "@" precedes "A". Thus the first capital entered will replace both of these values.

With step 2 coded, we can proceed to the final step.

Step 3. Display the results.

```
IF no capitals were typed,
 THEN
  display "No capitals"
 ELSE
  display first capital and last capital
END_IF
```

This step will display one of two possible messages; NOCAP_MSG if no capitals are entered, and CAP_MSG if there are capitals. We can declare them in the data segment as follows:

```
NOCAP_MSG       DB              'No capitals $'
CAP_MSG         DB              'First capital = '
FIRST           DB              ']'
                DB              ' Last capital = '
LAST            DB              '@ $'
```

When CAP_MSG is displayed, it will display "First capital =", then the value of FIRST, then "Last capital =", then the value of LAST. We used this same technique in the last program of Chapter 4.

The program decides, by inspecting FIRST, whether any capitals were read. If FIRST contains its initial value "]", then no capitals were read.

Step 3 may be coded as follows:

```
                MOV   AH,9       ;display string function
;if no capitals were typed
                CMP   FIRST,']'  ;FIRST = ']'?
                JNE   CAPS       ;no, display results
;then
                LEA   DX,NOCAP_MSG
                JMP   DISPLAY
CAPS:
                LEA   DX,CAP_MSG
DISPLAY:
                INT   21H        ;display message
;end_if
```

Here is the complete program:

Program Listing PGM6_2.ASM

```
TITLE PGM6_2: FIRST AND LAST CAPITALS
.MODEL SMALL
.STACK 100H
.DATA
PROMPT DB     'Type a line of text',0DH,0AH,'$'
NOCAP_MSG DB 0DH,0AH,'No capitals $'
CAP_MSG       DB      0DH,0AH,'First capital = '
FIRST DB      ']'
      DB      ' Last capital = '
LAST  DB      '@ $'

.CODE
MAIN  PROC
;initialize DS
      MOV  AX,@DATA
      MOV  DS,AX
;display opening message
      MOV  AH,9              ;display string function
      LEA  DX,PROMPT         ;get opening message
      INT  21H               ;display it
;read and process a line of text
      MOV  AH,1              ;read char function
      INT  21H               ;char in AL
WHILE_:
;while character is not a carriage return do
      CMP  AL,0DH            ;CR?
      JE   END_WHILE         ;yes, exit
;if character is a capital letter
      CMP  AL,'A'            ;char >= 'A'?
      JNGE END_IF            ;not a capital letter
      CMP  AL,'Z'            ;char <= 'Z'?
      JNLE END_IF            ;not a capital letter
;then
;if character precedes first capital
      CMP  AL,FIRST          ;char < first capital?
      JNL  CHECK_LAST        ;no, >=
;     then first capital = character
      MOV  FIRST,AL          ;FIRST = char
;end_if
CHECK_LAST:
;if character follows last capital
      CMP  AL,LAST           ;char > last capital?
      JNG  END_IF            ;no, <=
;then last capital = character
      MOV  LAST,AL           ;LAST = char
;end_if
END_IF:
;read a character
      INT  21H               ;char in AL
      JMP  WHILE_            ;repeat loop
END_WHILE:
;display results
```

```
        MOV  AH,9           ;display string function
;if no capitals were typed
        CMP  FIRST,']'      ;first = ']'
        JNE  CAPS           ;no, display results
;then
        LEA  DX,NOCAP_MSG   ;no capitals
        JMP  DISPLAY
CAPS:
        LEA  DX,CAP_MSG     ;capitals
DISPLAY:
        INT  21H            ;display message
;end_if
;dos exit
        MOV  AH,4CH
        INT  21H
MAIN    ENDP
        END  MAIN
```

Summary

- The jump instructions may be divided into unconditional and conditional jumps. The conditional jumps may be classified as signed, unsigned, and single-flag jumps.
- The conditional jumps operate on the settings of the status flags. The CMP (compare) instruction is often used to set the flags just before a jump instruction.
- The destination label of a conditional jump must be less than 126 bytes before or 127 bytes after the jump. A JMP can often be used to get around this restriction.
- In an IF-THEN decision structure, if the test condition is true, then the true-branch statements are done; otherwise, the next statement in line is done.
- In an IF-THEN-ELSE decision structure, if the test condition is true, then the true-branch statements are done; otherwise the false-branch statements are done. A JMP must follow the true-branch statements so that the false-branch will be bypassed.
- In a CASE structure, branching is controlled by an expression; the branches correspond to the possible values of the expression.
- A FOR loop is executed a known number of times. It may be implemented by the LOOP instruction. Before entering the loop, CX is initialized to the number of times to repeat the loop statements.
- In a WHILE loop, the loop condition is checked at the top of the loop. The loop statements are repeated as long as the condition is true. If the condition is initially false, the loop statements are not done at all.
- In a REPEAT loop, the loop condition is checked at the bottom of the loop. The statements are repeated until the condition is true. Because the condition is checked at the bottom of the loop, the statements are done at least once.

Glossary

AND condition	A logical AND of two conditions
conditional jump instruction	A jump instruction whose execution depends on status flag settings
loop	A sequence of instructions that is repeated
OR condition	A logical OR of two conditions
signed jump	A conditional jump instruction used with signed numbers
single-flag jump	A conditional jump that operates on the setting of an individual status flag
top-down program design	Program development by breaking a large problem into a series of smaller problems
unconditional jump	An unconditional transfer of control
unsigned jump	A conditional jump instruction used with unsigned numbers

New Instructions

CMP	JCXZ	JLE/JNG
JA/JNBE	JE/JZ	JMP
JAE/JNB	JG/JNLE	JNC
JB/JNAE	JGE/JNL	JNE/JNZ
JBE/JNA	JL/JNLE	LOOP
JC		

Exercises

1. Write assembly code for each of the following decision structures.

   a.
   ```
   IF AX < 0
    THEN
     PUT -1 IN BX
   END_IF
   ```

   b.
   ```
   IF AL < 0
    THEN
     put FFh in AH
    ELSE
     put 0 in AH
   END_IF
   ```

   c. Suppose DL contains the ASCII code of a character.
   ```
   (IF DL >= "A") AND (DL <= 'Z')
    THEN
     display DL
   END_IF
   ```

   d.
   ```
   IF AX < BX
    THEN
     IF BX < CX
      THEN
   ```

```
      put 0 in AX
     ELSE
      put 0 in BX
    END_IF
  END_IF
```

e.
```
IF (AX < BX) OR (BX < CX)
 THEN
  put 0 in DX
 ELSE
  put 1 in DX
END_IF
```

f.
```
IF AX < BX
 THEN
  put 0 in AX
 ELSE
  IF BX < CX
   THEN
    put 0 in BX
   ELSE
    put 0 in CX
  END_IF
END_IF
```

2. Use a CASE structure to code the following:

   Read a character.

   If it's "A", then execute carriage return.

   If it's "B", then execute line feed.

   If it's any other character, then return to DOS.

   Write a sequence of instructions to do each of the following:

   a. Put the sum $1 + 4 + 7 + \ldots + 148$ in AX.

   b. Put the sum $100 + 95 + 90 + \ldots + 5$ in AX.

   Employ LOOP instructions to do the following:

   a. put the sum of the first 50 terms of the arithmetic sequence 1, 5, 9, 13, . . . in DX.

   b. Read a character and display it 80 times on the next line.

   c. Read a five-character password and overprint it by executing a carriage return and displaying five X's. You need not store the input characters anywhere.

   The following algorithm may be used to carry out division of two nonnegative numbers by repeated subtraction:

```
initialize quotient to 0
WHILE dividend >= divisor DO
increment quotient
subtract divisor from dividend
END_WHILE
```

   Write a sequence of instructions to divide AX by BX, and put the quotient in CX.

6. The following algorithm may be used to carry out multiplication of two positive numbers M and N by repeated addition:

```
initialize product to 0
REPEAT
 add M to product
 decrement N
UNTIL N = 0
```

Write a sequence of instructions to multiply AX by BX, and put the product in CX. You may ignore the possibility of overflow.

7. It is possible to set up a count-controlled loop that will continue to execute as long as some condition is satisfied. The instructions

```
LOOPE label          ;loop while equal
```

and

```
LOOPZ label          ;loop while zero
```

cause CX to be decremented, then if CX <> 0 and ZF = 1, control transfers to the instruction at the destination label; if either CX = 0 or ZF = 0, the instruction following the loop is done. Similarly, the instructions

```
LOOPNE label         ;loop while not equal
```

and

```
LOOPNZ label         ;loop while not zero
```

cause CX to be decremented, then if CX <> 0 and ZF = 0, control transfers to the instruction at the destination label; if either CX = 0 or ZF = 1, the instruction following the loop is done.

   a. Write instructions to read characters until either a nonblank character is typed, or 80 characters have been typed. Use LOOPE.
   b. Write instructions to read characters until either a carriage return is typed or 80 characters have been typed. Use LOOPNE.

Programming Exercises

8. Write a program to display a "?", read two capital letters, and display them on the next line in alphabetical order.
9. Write a program to display the extended ASCII characters (ASCII codes 80h to FFh). Display 10 characters per line, separated by blanks. Stop after the extended characters have been displayed once.
10. Write a program that will prompt the user to enter a hex digit character ("0" . . . "9" or "A" . . . "F"), display it on the next line in decimal, and ask the user if he or she wants to do it again. If the user types "y" or "Y", the program repeats; if the user types anything else, the program terminates. If the user enters an illegal character, prompt the user to try again.

Sample execution:

```
ENTER A HEX DIGIT: 9
IN DECIMAL IS IT 9
DO YOU WANT TO DO IT AGAIN? y
ENTER A HEX DIGIT: c
ILLEGAL CHARACTER - ENTER 0..9 OR A..F: C
IN DECIMAL IT IS 12
DO YOU WANT TO DO IT AGAIN? N
```

11. Do programming exercise 10, except that if the user fails to enter a hex-digit character in three tries, display a message and terminate the program.

12. (hard) Write a program that reads a string of capital letters, ending with a carriage return, and displays the longest sequence of consecutive alphabetically increasing capital letters read.

Sample execution:

```
ENTER A STRING OF CAPITAL LETTERS:
FGHADEFGHC
THE LONGEST CONSECUTIVELY INCREASING STRING IS:
DEFGH
```