- the assembly program contains the 8086 instructions and assembler directives, also called pseudo-instructions.
- the instructions result into machine codes whereas the directives do not. They simply direct the assembler for correct translation.
- SEGMENT and ENDS directives defines boundaries of logical segments, ASSUME binds the logical segments to segment register and END denotes the end of the program.
- DB, DW, DD, DQ and DT are the directive used to define the variables and allocate the storage.
- PTR is used as byte, word or double word pointer to resolve the conflicts in memory reference.
- SEG and OFFSET are used to get the segment and offset address of variables or procedures.

3.3 Writing and Executing a Program

We are now ready to write a simple assembly language program and see how we can convert it to machine language and execute it. Using a text editor, we have to write our program in a file with extension *.asm*, for example *add.asm*. Assume that we have written code of Program 3.1 in the file *add.asm* to add two 16-bit numbers.

Program 3.1

Write a program to add 16-bit numbers 1234h and 4321h.

Solution

The program is

```
code    SEGMENT
        ASSUME cs:code

start:  MOV AX, 1234h     ;load AX with 1234h
        MOV BX, 4321h     ;load BX with 4321h
        ADD AX, BX        ;add AX and BX

        MOV AX, 4C00h     ;terminate the program
        INT 21h

code    ENDS
        END start
```

The program consists of only one segment named as code which is bind to CS register to direct embler that it is code segment. The first two MOV instructions initialize the AX and BX registers, pectively. Third instruction adds the contents of AX and BX register and stores the result into AX regis- The label start to first MOV instruction indicates that the execution begins from this instruction.

Function : Display a character
Function number : AH = 2
Input : DL = ASCII of character to display
Output : None

Following code will display, capital alphabet 'A' on the screen as its ASCII is 41h.

```
MOV AH, 2              ;function number
MOV DL, 41h            ;load DL with 'A'
INT 21h                ;call the service
```

Program 3.3

Write a program to convert a given alphabet from lowercase to uppercase.

Solution

The program is

```
code    SEGMENT
        ASSUME cs:code
start:  MOV AH, 1          ;read a character
        INT 21h
        SUB AL, 20h        ;convert lower to upper
        MOV AH, 2          ;print a character
        MOV DL, AL
        INT 21h
        MOV AX, 4C00h      ;terminate the program
        INT 21h
code    ENDS
        END start
```

The ASCII of lowercase letter starts from 61h (97) and that of uppercase from 41h (65). Subtracting 20h (32) from the ASCII of lowercase letter converts it to uppercase. Assume that the above code is stored in the file *char_con.asm;* after assembling and linking, the *char_con.exe* is created. Running it on DOS prompt shows the result as

```
C:\MASM>char_con
aA
C:\MASM>
```

Once a program starts running, it waits for the user to enter a character. We have entered 'a' which is echoed on screen by read function and then it is converted to uppercase by subtract instruction and

management. Program 3.2 writes the listing of Program 3.1 using separate data and code segments. Assume that the code for Program 3.2 is stored in the file *add1.asm*.

Program 3.2

Rewrite Program 3.1 with separate data and code segments.

Solution

The program is

```
data    SEGMENT
        n1   DW 1234h        ;define n1 with value 1234h
        n2   DW 4321h        ;define n2 with value 4321h
        ans DW   ?           ;allocate space for answer
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data         ;initialize the data segment
        MOV DS,.AX
        MOV AX, n1           ;load AX with n1
        MOV BX, n2           ;load BX with n2
        ADD AX, BX           ;add AX and BX
        MOV ans, AX          ;store result in ans
        MOV AX, 4C00h        ;terminate the program
        INT 21h
code    ENDS
        END start
```

Note that only CS register is automatically initialized to base address of the physical segment co sponding to the logical segment `code`. The offset address for the CS register is always provided by Instruction Pointer (IP) register and the instructions are fetched from the current CS:IP address du execution starting from offset 0000h from the segment pointed by CS. Each time IP is increme by the length of the instruction to point to next instruction in sequence. This is how the progra executed from first instruction to last instruction.

During the execution, instructions refer to data from the data segment. To get the correct value DS register must point to physical segment created to store our data values defined in logical seg `data`. However, the initialization of the DS register is not done automatically. This is to be perfo by the user in beginning of the code segment before instructions refer any data from data segmen following statements in code segment performs this task:

```
        MOV AX, data
```

Function : Display a character
Function number : AH = 2
Input : DL = ASCII of character to display
Output : None

ollowing code will display, capital alphabet 'A' on the screen as its ASCII is 41h.

```
MOV AH, 2          ;function number
MOV DL, 41h        ;load DL with 'A'
INT 21h            ;call the service
```

Program 3.3

Write a program to convert a given alphabet from lowercase to uppercase.

Solution

The program is

```
code    SEGMENT
        ASSUME cs:code
start:  MOV AH, 1          ;read a character
        INT 21h
        SUB AL, 20h        ;convert lower to upper
        MOV AH, 2          ;print a character
        MOV DL, AL
        INT 21h
        MOV AX, 4C00h      ;terminate the program
        INT 21h
code    ENDS
        END start
```

The ASCII of lowercase letter starts from 61h (97) and that of uppercase from 41h (65). Su
20h (32) from the ASCII of lowercase letter converts it to uppercase. Assume that the abo
stored in the file *char_con.asm*; after assembling and linking, the *char_con.exe* is created. Ru
DOS prompt shows the result as

```
C:\MASM>char_con
aA
```

haracter. We have enter
bract i

es containing "Hello, World$"

```
mes     DB      "Hello, World$"
```

cal address of the string is defined by segment and offset address of string variable mes which we can get using the SEG and OFFSET directives. The following code displays the above string:

```
MOV AH, 9                   ;function number
MOV BX, SEG mes             ;get segment address
MOV DS, BX
MOV DX, OFFSET mes          ;get offset address
INT 21h
```

Observe that the segment address is copied into DS register using BX register because the immediate transfer to segment registers is not permitted . Keep in mind that all the register values including function number are to be preserved until service is called by INT 21h instruction. For example, by mistake if you use AX to transfer the segment address into DS, the function number loaded in AH gets overwritten and the service gets improper function number.

Program 3.4

Write a program to display message "Hello, World".

Solution

The program is

```
data    SEGMENT
mes     DB      "Hello, World$"
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data                ;initialize the data segment
        MOV DS, AX
        MOV AH, 9                   ;function number
        MOV BX, SEG mes             ;get segment address
        MOV DS, BX
        MOV DX, OFFSET mes          ;get offset address
        INT 21h
        MOV AX, 4C00h               ;terminate the program
```

```
        INT 21h
code    ENDS
        END start
```

Assume that the Program 3.4 is stored in *hello.asm*, the assembling and linking will produce the file *hello.exe*. The execution of it shows the output as follows:

```
C:\MASM>hello
Hello, World
C:\MASM>
```

Figure 7 shows the code and data segment layout of Program 3.4. Observe that directives SEG and OFFSET are replaced by the values. The layout of data segment at address 0B5E:0000h shows the storage of "Hello, World$" as ASCII values starting with 65h (ASCII of H) and ending with 24h (ASCII of $).

```
C:\MASM>debug hello.exe
-u 0 14
0B5F:0000 B85E0B          MOV     AX,0B5E
0B5F:0003 8ED8            MOV     DS,AX
0B5F:0005 B409            MOV     AH,09
0B5F:0007 BB5E0B          MOV     BX,0B5E
0B5F:000A 8EDB            MOV     DS,BX
0B5F:000C BA0000          MOV     DX,0000
0B5F:000F CD21            INT     21
0B5F:0011 B8004C          MOV     AX,4C00
0B5F:0014 CD21            INT     21
-
-e 0B5E:0000  48.  65.  6C.  6C.  6F.  2C.  20.  57.
0B5E:0000  48.  65.  6C.  6C.  6F.  2C.  20.  57.
0B5E:0008  6F.  72.  6C.  64.  24.
-
```

Figure 7 Code and data layout of Program 3.4.

If you examine the code shown in Fig. 7, it shows that first two instructions used for initial
well as the fourth and fifth instructions used to get the segment address of mes transfer the sam
value 0B5Eh in DS as both place we refer to same segment. In this case, we can remove the instr
get the segment address of mes to avoid duplication. We can simply write the code to display

```
MOV AH, 9                ;function number
MOV DX, OFFSET mes       ;get offset address
INT 21h
```

This will save the execution time. However, you must be careful that this is not always th
ecessary to write it exclusively, otherwise problems
intable ASCII characters, it also
ASCII character.

key (0Dh) is not counted in the length. The ASCII values of string characters starts from `str_buf+2` position in string buffer as first two bytes are occupied by maximum length and actual length.

The function 10 needs the address of string buffer defined to store an input string in the DS:DX register in same manner as function to display a string. The details of function to read a string are as follows:

Function	: Read a string
Function number	: AH = 10
Input	: DS = segment address of string buffer DX = offset address of string buffer
Output	: String is stored in string buffer

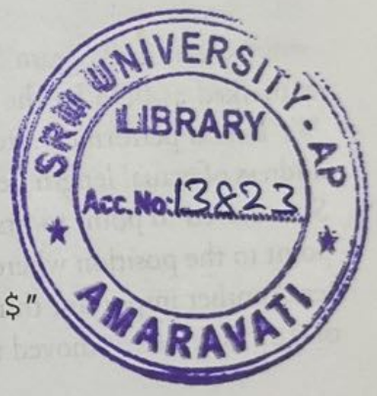
We can write the code to read the string as follows.

```
MOV AH, 10                          ;function number
MOV BX, SEG str_buf                 ;get segment address
MOV DS, BX
MOV DX, OFFSET str_buf              ;get offset address
INT 21h
```

This is same as displaying a string except the function number. Once the above code is executed, the program stops to read a string from keyboard. It accepts the characters (maximum 255) until **Enter** key is pressed, stores them in data part of the buffer and also updates the actual length field with length of the string entered.

Program 3.5

Write a program to read a string from the keyboard and display it on the screen.



Solution

The program is

```
data    SEGMENT
        mes1        DB   "Enter a string : $"
        str_buf     DB   255,256 dup(0)
        nl          DB   0Dh, 0Ah, '$'
        mes2        DB   "String entered is : $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX
        MOV AH, 9           ;print message
        MOV DX, OFFSET mes1
        INT 21h
        MOV AH, 10          ;read a string
        MOV DX, OFFSET str_buf
        INT 21h
```

```
        MOV SI, OFFSET str_buf+1 ;replace end of string
        MOV CX, 0                ;with '$'
        MOV CL, BYTE PTR [SI]
        INC SI
        ADD SI, CX
        MOV BYTE PTR [SI], '$'
        MOV AH, 9                ;print new line
        MOV DX, OFFSET nl
        INT 21h
        MOV AH, 9                ;print message
        MOV DX, OFFSET mes2
        INT 21h
        MOV AH, 9                ;print a string
        MOV DX, OFFSET str_buf+2
        INT 21h
        MOV AX, 4C00h            ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in Program 3.5, first the string is read from the user. We know that the end of the strin is marked as 0Dh by the function 10. In order to print it using function 9, we have to change it t '$'. This is performed after reading string through the block of six instructions. The first – the offse address of actual length field – is loaded into SI, the length is read from that offset into CL register, th SI is moved to point to first character in data field and then adding length in CX to SI moves the SI t point to the position where end of the string is stored. The last instruction replaces it by the '$' charac ter. Another important thing is that while finally printing an entered string using function 9, the offse of `str_buf+2` is moved to DX as the actual string characters start from that position in

Check Point

The byte stored in AL register is replaced by the byte stored at the location BX + AL in the lookup table where BX is the base of the lookup table and contains offset address of the lookup table stored in data segment. Program 3.7 clears the concept by converting lowercase string to uppercase using lookup table and XLAT instruction.

Program 3.7

Write a program to convert lowercase string to uppercase using XLAT instruction. Input string should not contain characters other than lowercase alphabets including spaces.

Solution

The program is

```
data    SEGMENT
        mes1    DB  "Enter a lower case string : $"
        str_buf DB  255,256 dup(0)
        nl      DB  0Dh, 0Ah, '$'
        mes2    DB  "Uppercase string is : $"
        table   DB  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data      ;initialize the data segment
        MOV DS, AX

        MOV AH, 9   ;print message
        MOV DX, OFFSET mes1
        INT 21h

        MOV AH, 10  ;read a string
        MOV DX, OFFSET str_buf
        INT 21h
        ;store the base of table to BX
        MOV BX, offset table
        ;get the length of string into CL
        MOV SI, OFFSET str_buf+1
        MOV CX, 0
        MOV CL, BYTE PTR [SI]
next:   INC SI                  ;point next character
        MOV AL, BYTE PTR [SI]   ;get the character
        SUB AL, 'a'             ;convert to index
        XLAT                    ;translate to uppercase
        MOV BYTE PTR [SI], AL   ;store back
        LOOP next               ;repeat if not end
```

3.7 Arithmetic Instructions

The arithmetic operations are most important operations in any programming. The 8086 processor provides a rich set of arithmetic instructions to perform add, subtract, multiply and divide operations on byte and word operands. It not only supports these operations on binary numbers, but also provides the instructions to handle the operands in the BCD form – both packed and unpacked. The arithmetic operations manipulate the operand values and hence they affect the conditional flags. This means that the contents of the flag register are updated based on the result of the operation. We will learn the simple arithmetic instructions handling binary operands in this section, while rest of the instructions dealing with BCD formats will be discussed in the next chapter. The arithmetic instructions are explained as follows.

Addition Instructions

The addition instructions cover the ADD (add), ADC (add with carry) and INC (increment) instructions. Let us learn them in detail with examples.

ADD Instruction

The ADD instruction is used to perform the arithmetic addition of the operands. The syntax for the ADD instruction is as follows:

```
ADD destination, source
(destination) ← (destination) + (source)
```

The contents of *source* and *destination* are added and the result is stored in *destination*. The *source* can be immediate value, register or memory location. The *destination* can be register or memory location. Both *source* and *destination* cannot be in memory. All the six conditional flags are updated based on the result stored in *destination*. Following are some examples:

```
ADD AX, BX                ;if AX = 5623h and BX = A236h
                          ;result in AX = F859h
                          ;OF=0, CF=0, PF=1, AF=0, ZF=0, SF=1
                          ;(AL) = (AL)+35h
ADD AL, 35h
ADD SI, WORD PTR [BX]     ;add SI with word at offset BX
ADD BYTE PTR [SI], 2      ;add 2 to the byte at offset SI
ADD AL, n                 ;add byte variable n to AL
```

Program 3.8

Write a program to add five words.

Solution

The program is

```
data    SEGMENT
block   DW  1234h, 5634h, 00D2h, 23A1h, 0AA45h
        DW  ?
```

(destination) ↔ (source)

The contents of *source* and *destination* operands are swapped (i.e., exchanged) with each other. The *source* can be register or memory location. The *destination* can be register or memory location. The *source* and *destination* both cannot represent the memory location. Operands must be of same type either bytes or words. Following are some examples:

```
XCHG AX, BX                 ;exchange AX and BX
XCHG CL, BYTE PTR [SI]      ;exchange CL and byte at [SI]
XCHG [SI+2], AX             ;exchange word at [SI+2] and AX
XCHG DX, n1                 ;exchange DX and word defined by
```

Program 3.6

Write a program to exchange the contents of two words stored in memory.

Solution

The program is

```
data    SEGMENT
        n1      DW      1234h
        n2      DW      4321h
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data
        MOV DS, AX          ;initialize the data segment

        MOV AX, n1          ;get n1 to AX
        XCHG AX, n2         ;exchange AX and n2
        MOV n1, AX          ;store AX(n2) in n1

        MOV AX, 4C00h
        INT 21h             ;terminate the program
code    ENDS
        END start
```

XLAT Instruction

It is known as translate instruction. It is mainly used to translate the codes from one s
using lookup table. The syntax for the XLAT instruction is as follows:

```
XLAT
(AL) ← (BX + AL)
```

```
        INC SI                          ;go to next position
        MOV BYTE PTR [SI], '$' ;end the string with '$'
        MOV AH, 9                       ;print new line
        MOV DX, OFFSET nl
        INT 21h

        MOV AH, 9                       ;print message
        MOV DX, OFFSET mes2
        INT 21h

        MOV AH, 9                       ;print a string
        MOV DX, OFFSET str_buf+2
        INT 21h

        MOV AX, 4C00h                   ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in Program 3.7, after reading a string, the code for conversion using lookup table and XLAT instruction is written and then the converted string is printed. For conversion, first the offset of lookup table defined as `table` is get into the BX register. The length of the string is stored in the CX register for controlling the number of iterations using LOOP instruction. LOOP instruction repeats the code from the instruction labelled with label `next` up to LOOP instruction CX number of times. Each time it reduces the CX by 1 and if CX ≠ 0, it jumps to `next` and repeats the operation. When CX = 0, the loop is terminated. Each time in a loop the current character pointed by SI is moved into the AL, the ASCII of 'a' is subtracted from it to convert it into the index (index of 'a' is 0, 'b' is 1, ..., 'z' is 25) and then it is converted to uppercase letter using XLAT instruction. For example, if character is 'd', then the index is 3 and the character at BX+3 (fourth character in `table`) is 'D'. The converted character is stored back in the original string at SI. After converting whole string in this manner, it is terminated with '$' so that we can print it properly. The execution of program produces the results as

```
Enter a lowercase string : ma
Uppercase string is
```

Run this program in the debug using
executed. When the LOOP inst
start of loop. You can
be executed. Wh

```
        ADD DI, 2                ;forward pointer
        LOOP next                ;go to next

        MOV AX, 4C00h            ;terminate the program
        INT 21h

code    ENDS
        END start
```

As seen in Program 3.9, each time in a loop, a byte value pointed by SI is transferred to AL, multiplied by the constant in BL and then stored at the next word position pointed by the DI. The pointer SI is incremented by 1 to point to next byte and the pointer DI is incremented by 2 to point to next word position.

DIV Instruction

It is known as unsigned division and used to divide an unsigned word by an unsigned byte or unsigned double word by an unsigned word. The syntax for the DIV instruction is as follows:

```
DIV source
(AH) = reminder, (AL) = quotient ← (AX) / (source byte)
(DX) = reminder, (AX) = quotient ← (DX:AX) / (source word)
```

The *source* can be register or memory location. If the divisor is an unsigned byte, then the dividend must be an unsigned word in the AX register. After performing division operation, the 8-bit remainder will be stored in the AH register and the 8-bit quotient will be stored in the AL register. If the divisor is an unsigned word, then the dividend must be unsigned double word in the DX:AX register. After performing division operation, the 16-bit remainder will be stored in the DX register and the 16-bit quotient will be stored in the AX register. The quotient will always be truncated. All the conditional flags are undefined for the DIV instruction. If we try to divide by 0 or if the quotient does not fit in the destination (greater than FFh or FFFFh) then the interrupt type 0 known as "divide by zero" will be called. We will learn the 8086 interrupts in Chapter 7.

For performing division of unsigned byte by unsigned byte or unsigned word by unsigned word, the dividend must be extended by putting 0s in the upper byte or upper word and then division by word/byte or double word/word can be used. Following are some examples.

```
DIV BL                   ;if AX=0033h (51 decimal) and BL=2,
                         ;the remainder AH=1 and
                         ;question AL =19h (25 decimal)

DIV WORD PTR  [SI]       ;divide DX:AX by word at offset SI
```

Program 3.10

Write a program to divide 32-bit unsigned number by an 16-bit unsigned number.

Solution

The program is

```
data    SEGMENT
        DD    11204534h
```

upper byte of a 16-bit result or upp... then CF = OF
otherwise CF = OF = 1. This means that if part of the result is stored in upper byte or upper word
both CF and OF are set. This information is useful to decide whether or not we can discard upper
or upper word from the result. The conditional flags AF, PF, ZF and SF are undefined for this ins
tion. Following are some examples:

```
MUL BL                      ;if BL = 5, AL = 10, then AX = 32h
                            ;(50 decimal)
MUL WORD PTR [BX+SI]        ;multiply word at offset BX + SI
                            ;with AX, result in DX:AX
```

If you need to multiply an unsigned byte with an unsigned word, then first you need to co... an unsigned byte to an unsigned word by putting 0s in upper 8-bits and then multiply two wo... Consider a following example:

```
MOV AL, 05h       ;load AL with unsigned value 05h
MOV BX, 2312h     ;load AX with unsigned value 2312h
MOV AH,00h        ;convert to unsigned word AX = 0005h
MUL BX            ;multiply
```

Program 3.9

Write a program to perform scalar multiplication of array of five unsigned bytes.

Solution

The program is

```
data    SEGMENT
        n    DB 03h
        val  DB 12h,0A5h,0FFh,34h,98h
        s_val DW 5 dup(0)
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data
        MOV DS, AX                  ;initialize the data segment

        MOV CX, 5
        MOV BL, n                   ;load the count
        LEA SI, val                 ;get scalar constant
        LEA DI, s_val               ;pointer to original values
next:   MOV AL, BYTE PTR [SI]       ;pointer to scaled values
        MUL BL                      ;read next value
        MOV WORD PTR [DI], AX       ;multiply by scalar constant
        INC SI                      ;store scaled value
                                    ;forward pointer
```

```
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data              ;initialize the data segment
        MOV DS, AX
        MOV CX, 5                 ;load the count
        MOV AX, 0                 ;intialize sum
        LEA BX, block             ;point to first number
next:   ADD AX, WORD PTR [BX]     ;add next number
        ADD BX, 2                 ;move to next number
        LOOP next                 ;go to next
        MOV WORD PTR [BX], AX     ;store the result
        MOV AX, 4C00h             ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in the program, CX is initialized to 5, AX with 0 and BX to point to first word. Each time in the loop, the word pointed by BX is added to AX and BX in incremented by 2 to point to next word. After completing the loop, the result in AX is stored to the next word position in memory.

ADC Instruction

The ADC is known as "add with carry". The syntax for the ADD instruction is as follows.

ADD *destination, source*

(*destination*) ← (*destination*) + (*source*) + (CF)

It adds the contents of the *source* and *destination* plus carry flag CF and stores the result into the *destination*. The rest of the things are same as the ADD instruction. Following are some examples:

```
ADC AX, BX          ;if AX = 5623h, BX = A236h and CF = 1
                    ;result in AX = F85Ah
ADC AL, 2           ;OF=0, CF=0, PF=1, AF=0, ZF=0, SF=
                    ;(AL) = (AL) + 2 +(CF)
```

This instruction is useful in performing the addition of 32-bit numbers (i.e., c ls) in steps. First the ADD will add the lower words of both the operand n ADC ds. Ir igher words of both the operands along with carry generated by adding th we can perform higher order additions such as 64-bit additions al

INC Instruction

The INC instruction is used increm
tion is as foll

Program 3.11

Write a program to perform ORing of two 16-bit numbers without using OR instruction. [Hint: De Morgan's rule, $A + B = (A' \cdot B')'$]

Solution

The program is

```
data    SEGMENT
        n1      DW      1234h
        n2      DW      4321h
        ans     DW        ?
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX

        MOV AX, n1          ;load AX with n1
        NOT AX              ;complement AX
        MOV BX, n2          ;load BX with n2
        NOT BX              ;complement BX
        AND AX, BX          ;AND AX with BX
        NOT AX              ;complement result
        MOV ans, AX         ;store result

        MOV AX, 4C00h       ;terminate the program
        INT 21h
code    ENDS
        END start
```

As seen in Program 3.11, the operands are complemented using NOT instruction, then the AN of complemented operands is performed and finally the result of AND operation is complemente get the ORing of operands.

XOR Instruction

The XOR instruction performs the bit-wise XOR operation. The syntax for the XOR instruction follows:

```
XOR destination, source
```

```
            divisor   DW   1250h
            rem       DW    ?
            quetient  DW    ?
data        ENDS
code        SEGMENT
            ASSUME cs:code, ds:data
start:      MOV AX, data       ;initialize the data segment
            MOV DS, AX
            LEA BX, dividend ;get the offset of dividend
            MOV AX, WORD PTR [BX]    ;load AX with lower word
            MOV DX, WORD PTR [BX+2] ;load DX with upper word
            DIV divisor              ;divide by divisor
            MOV rem, DX              ;store remainder
            MOV quetient, AX         ;store quetient
            MOV AX, 4C00h     ;terminate the program
            INT 21h
code        ENDS
            END start
```

Signed Multiplication and Division Instructions

The 8086 also provides instructions for multiplication and division for the signed operands, name IMUL and IDIV. They work in the same way as their unsigned versions. Before learning them, let first learn the instructions, converting signed byte to signed word CBW and converting signed word signed double word CWD useful to extend the signed operands. Remember that signed numbers a stored normally in sign-magnitude form using 2's complement.

CBW Instruction

This instruction converts the signed byte stored in the AL register to a signed word by copying the sig bit of AL into all the bits of AH. Consider the following examples:

```
CBW         ;if AL = 0000 0101 (
            ;then AX = 0000 000(
CBW         ;if AL = 1111 1011
            ;then AX = 1111 1(
```

101 (+5)

1011 (-

CWD Instruction

This instruction converts the signed word s
the sign bit of AX into

mented by 1 to point to next byte and the pointer DI is incremented by 2 to point to next word position pointed by the DI. The pointer SI is incremented by 2 to point to next word position.

DIV Instruction

It is known as unsigned division and used to divide an unsigned word by an unsigned byte or unsigned double word by an unsigned word. The syntax for the DIV instruction is as follows:

```
DIV source
(AH) = reminder, (AL) = quotient ← (AX) / (source byte)
(DX) = reminder, (AX) = quotient ← (DX:AX) / (source word)
```

The *source* can be register or memory location. If the divisor is an unsigned byte, then the dividend must be an unsigned word in the AX register. After performing division operation, the 8-bit remainder will be stored in the AH register and the 8-bit quotient will be stored in the AL register. If the divisor is an unsigned word, then the dividend must be unsigned double word in the DX:AX register. After performing division operation, the 16-bit remainder will be stored in the DX register and 16-bit quotient will be stored in the AX register. The quotient will always be truncated. All the conditional flags are undefined for the DIV instruction. If we try to divide by 0 or if the quotient does not in the destination (greater than FFh or FFFFh) then the interrupt type 0 known as "divide by zero" be called. We will learn the 8086 interrupts in Chapter 7.

For performing division of unsigned byte by unsigned byte or unsigned word by unsigned word the dividend must be extended by putting 0s in the upper byte or upper word and then division word/byte or double word/word can be used. Following are some examples.

```
DIV BL                      ;if AX=0033h (51 decimal) and BL=
                            ;the remainder AH=1 and
                            ;question AL =19h (25 decimal)
DIV WORD PTR  [SI]          ;divide DX:AX by word at offset SI
```

Program 3.10

Write a program to divide 32-bit unsigned number by an 16-bit unsigned number.

Solution

The program is

```
data        SEGMENT
            dividend  DD   11204534h
```

```
        n2 DW 0A245h
        min DW ?
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data
        MOV DS, AX                    ;initialize

        MOV AX, n1
        CMP AX, n2                    ;get n1 in AX
        JC skip                       ;compare AX with n2
        MOV AX, n2                    ;AX < n2
skip:   MOV min, AX                   ;AX > n2, store n2 in AX
                                      ;store AX to min
        MOV AX, 4C00h
        INT 21h                       ;terminate
code    ENDS
        END start
```

As can be seen from Program 4.2, both the numbers are compared and the message is printed based on the status of the carry flag (CF). After comparison, if CF = 1, the n1 is small, otherwise n2 is small. Whichever number is smaller, it is kept in the AX register and finally stored into the min.

Program 4.3

Write a program to check whether the given 16-bit number is odd or even. Print the appropriate message.

Solution

The program is

```
data    SEGMENT
        num  DW 3D2Bh
        mes1 DB "Number is even $"
data    mes2 DB "Number is odd $"
code    ENDS

        SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data
        MOV DS, AX                    ;initialize
        MOV AX, num
                                      ;get num in AX
```

```
        n2    DW 0A245h
        mes1  DB "Both the numbers are same $"
        mes2  DB "Both the numbers are different $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data
        MOV DS, AX                      ;initialize

        MOV AX, n1
        CMP AX, n2                      ;get n1 in AX
                                        ;compare AX with n2
        MOV AH, 9
        JNZ noteq                       ;print message
        LEA DX, mes1
        JMP over                        ;equal
noteq:  LEA DX, mes2                    ;not equal
over:   INT 21h

        MOV AX, 4C00h                   ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.1, both the numbers are compared and the message is printed based on th status of the zero flag (ZF). If both the numbers are equal then the compare instruction sets the ZF to otherwise 0. In the above example, the message printed is "Both the numbers are different as the numbers are not equal. You can test the program by giving the same value for both the numbers.

Program 4.2

Write a program to find the minimum of two 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        n1 DW 1234h
```

Program 4.1

Write a program to find whether two 16-bit numbers are equal. Print the appropriate message.

Solution

The program is

```
data    SEGMENT
        n1    DW 1234h
        n2    DW 0A245h
        mes1 DB "Both the numbers are same $"
        mes2 DB "Both the numbers are different $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data                ;initialize
        MOV DS, AX

        MOV AX, n1                  ;get n1 in AX
        CMP AX, n2                  ;compare AX with n2

        MOV AH, 9                   ;print message
        JNZ noteq
        LEA DX, mes1                ;equal
        JMP over
noteq:  LEA DX, mes2                ;not equal
over:   INT 21h

        MOV AX, 4C00h               ;terminate
        INT 21h

code    ENDS
        END start
```

As can be seen from Program 4.1, both the numbers are compared and the message is printed based on the status of the zero flag (ZF). If both the numbers are equal then the compare instruction sets the ZF to 1, otherwise 0. In the above example, the message printed is "Both the numbers are different" as the numbers are not equal. You can test the program by giving the same value for both the numbers.

Program 4.2

Write a program to find the minimum of two 16-bit numbers.

Solution

The program is

```
data    SEGMENT
```

```
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data
        MOV DS, AX              ;initialize

        MOV CX, N               ;get N into CX
        MOV AX, 0               ;sum in AX = 0
        MOV BX, 1               ;First num in BX = 1
next:   ADD AX, BX              ;ADD next number
        INC BX                  ;increment BX
        DEC CX                  ;decrement N
        JNZ next                ;if not 0, goto next
        MOV sum, AX             ;store AX into sum

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.4, the last number N is stored in CX, the sum is stored in AX and hence it is initialized to 0, and the BX contains the next number to be added starting from 1. Each time in the loop, BX is added to partial sum in AX, BX is incremented by 1 to get the next number in sequence and CX is reduced by 1. If the CX does not reach to 0, the process is repeated, otherwise the control moves to the next instruction which stores the sum value in the AX register to the data segment in variable sum. As the value of N in the data segment is 000Ah (decimal 10), the answer would be 0037h (decimal 55). You can check that by executing program in the debug mode.

Program 4.5

Write a program to add given N 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DW 0005h,5234h,4512h,1215h,213Dh,6D2Fh
              DW ?
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data            ;initialize
        MOV DS, AX
```

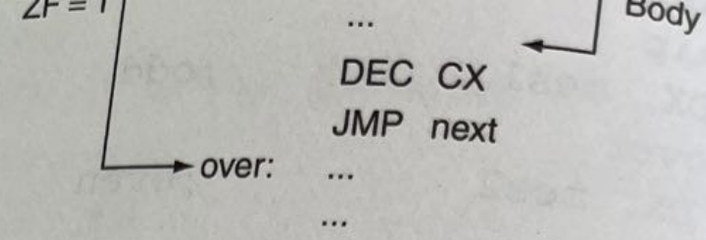


Figure 6 Implementing a loop in 8086 programming.

again with the updated value, and if it is true, the same process is repeated. It is repe
the condition is true. When the condition is false, the loop is completed and the contr
to a statement after the `Begin-End` block. Figure 6 shows the implementation of lo
programming.

As can be seen from Fig. 6, the CX register acts as looping variable which is initialize
value 5. Then the CX is compared with 0, and if it is not 0, then the body is executed. Th
tion in the body is DEC CX which reduces the CX by 1 and then the control is transferre
compare instruction to check the condition again. If the condition is true, the process is re
When the condition is false, the control is transferred to an instruction after the JMP ins
this case, the body of the loop is executed 5 times and hence CX acts as count indicating
times the loop is to be executed. The loop in Fig. 6 is more conveniently written as follows:

```
next:   MOV CX, 5          ;initialize count
        ...                ;body
        ...
        ...
        DEC CX
        JNZ next           ;reinitialization
                           ;goto next, if ZF ≠ 1
```

The following programs demonstrate the use of looping in the 8086 programs.

Program 4.4

Write a program to find the sum of first N positive integers.

Solution

The program is

```
data    SEGMENT
N       DW 000Ah
sum     DW ?
data    ENDS
```

```
        AND AX, 0001h          ;AND with 1
        CMP AX, 0001h          ;check LSB

        MOV AH, 9              ;print message
        JZ skip
        LEA DX, mes1           ;odd
        JMP over
        LEA DX, mes2           ;even
skip:
over:   INT 21h

        MOV AX, 4C00h          ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.3, the LSB of the numbers is checked to find whether the number is odd or even. If the LSB is 1, the number is odd, otherwise it is even. To check the LSB, the number is masked with `0001h` (only LSB = 1, rest of the bits are 0). If the number is odd, the result of the masking is `0001h`, otherwise it is `0000h`. Then the result is compared with the `0001h` and based on the status of zero flag (ZF), the message is printed. In the above case, the number is odd and hence the message printed is "`Number is odd`". You can test the program by giving the even value for the variable num. The above program can be easily implemented using the shift instructions. The LSB can be shifted into carry flag (CF), and based on the value of CF (either 0 or 1), the decision can be made. We will learn the shift instructions in Chapter 5.

The above examples implement the simple if-else structure provided in high-level programming languages. The high-level languages also provide the multiple if-else, also known as ladder if-else. This can be easily implemented in the 8086 programming by extending the concept used to implement simple if-else by using multiple conditions.

Looping is a technique to run the same block of instructions multiple times. In other words, looping allows us to perform the same operation many times. The general form of looping as provided by the most of the high-level languages is as follows:

```
Initialization;
While (Condition)
Begin
        Statement block;
        Reinitialization;

End
```

First, the initialization of looping variables is done. It is done only once. Then the condition involving looping variables is checked, and if is true, the body of the loop enclosed in the `Begin-End` executed. The body of the loop contains the statement block followed by reinitialization. After performing the statement block, the reinitialization part updates the looping variables. The condition is chec

Program 4.7

Write a program to reverse a given array of 16-bit numbers.

Solution

The program is

```
data    SEGMENT
   block DW 0006h,1234h,5234h,4512h,1215h,0D213h,0AD2Fh
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data                    ;initialize
        MOV DS, AX

        LEA SI, block                   ;point to the size
        MOV CX, WORD PTR [SI]           ;get size into CX
        ADD SI, 2                       ;SI pointing to start
        MOV DX, CX                      ;multiply CX by 2
        MOV AL, DL
        MOV BL, 2
        MUL BL
        MOV DX, AX                      ;DX = CX x 2
        SUB DX, 2                       ;reduce DX by 2
        MOV DI, SI
        ADD DI, DX                      ;DI pointing to end
        MOV AX, CX                      ;divide CX by 2
        MOV BL, 2
        DIV BL
        MOV CX, 0
        MOV CL, AL
next:   MOV AX, WORD PTR [SI]           ;exchange the numbers
        MOV BX, WORD PTR [DI]           ;pointed by SI
        MOV WORD PTR [SI], BX           ;and DI pointers
        MOV WORD PTR [DI], AX
        ADD SI, 2                       ;SI = SI + 2
        SUB DI, 2                       ;DI = DI - 2
        DEC CX                          ;decrement CX
        JNZ next                        ;if not 0, goto next

        MOV AX, 4C00h
        INT 21h                         ;terminate
code    ENDS
        END start
```

```
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data                    ;initialize
        MOV DS, AX
        MOV SI, OFFSET block            ;intialize pointer
        MOV CX, WORD PTR [SI]           ;get count in CX
        ADD SI, 2                       ;point to first number
        MOV AX, WORD PTR [SI]           ;get first num in AX
        ADD SI, 2                       ;point to next number
next:   DEC CX                          ;decrement CX
        JZ over                         ;if 0, goto over
        CMP AX, WORD PTR [SI]           ;compare AX with next num
        JC skip                         ;AX is small
        MOV AX, WORD PTR [SI]           ;store next num in AX
        JMP next                        ;goto next
skip:   MOV WORD PTR [SI], AX           ;store smallest
over:   MOV AX, 4C00h                   ;terminate
        INT 21h
        ENDS
code    END start
```

As can be seen from Program 4.6, the data segment defines an array `block` storing the size of the array followed by array elements. In our case, the size is 5 and then five 16-bit numbers are stored. After array, the memory for storing a word is reserved to store the smallest number. In the code segment, the pointer SI is set to point to `block` and the size of array is stored in the CX register. Then the first number is stored in the AX register. Within the loop, the pointer is incremented to point to the next element in array and the CX register is decremented. If the CX is 0, the comparisons are completed, and the control moves out of the loop and stores the answer in the next free word position. If the CX is not 0, the smaller value so far in AX is compared with the next value and the smaller between them is kept in the AX, and the process is repeated. This program even works for one number, as we check the counter initially in the loop after reducing it by 1. If there is only one number in the array, first time in the loop the CX after decrementing becomes 0 and the control comes out and stores the only number as smallest in the next free word position. Test the program for different array sizes and verify the answer by executing the program in the debug mode. You will get the result in the next word position after your array elements. For example, in the above case for five words, the result will be available after the fifth element in the data segment.

```
        MOV SI, OFFSET block      ;intialize pointer
        MOV CX, WORD PTR [SI]     ;get count in CX
        ADD SI, 2                 ;point to first number
        MOV AX, WORD PTR [SI]     ;get first num in AX
        ADD SI, 2                 ;point to next number
next:   DEC CX                    ;decrement CX
        JZ over                   ;if 0, goto over
        ADD AX, WORD PTR [SI]     ;add next number
        JMP next                  ;goto next
skip:   MOV WORD PTR [SI], AX     ;store result
over:
        MOV AX, 4C00h             ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.5, the data segment defines an array `block` storing the size of the array followed by array elements. In our case, the size is 5 and then five 16-bit numbers are stored. After array, the memory for storing a word is reserved to store the result. In the code segment, the pointer SI is set to point to `block` and the size of array is stored in the CX register. Then the first number is stored in the AX register. Within the loop, the pointer is incremented to point to the next element in the array and the CX register is decremented. If CX is 0, all the numbers are added, and the control moves out of the loop and stores the result in the next free word position. If the CX is not 0, the next number is added to the partial addition value in the AX, and the process is repeated. This program works for the array of size 1 or more. The result is always stored at the end of the array after the last array element.

The complex real-life problems include both decision making and looping. The following programs demonstrate the use of decision making and looping together to write more complex programs to solve real-life problems.

Program 4.6

Write a program to find the minimum from a block of N 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DW 0005h,5234h,4512h,1215h,0D213h,0AD2Fh
              DW ?
data    ENDS
```

```
        MOV AX, 1                ;set AX = 1
        MOV BX, 1                ;set BX = 1
next:   MUL BX                   ;AX = AX x BX
        INC BX                   ;increment BL
        CMP BX, CX               ;is BX <= N?
        JBE next                 ;if yes, goto next
        MOV fact, AX             ;store result

        MOV AX, 4C00h            ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.9, the 8-bit number N whose factorial is to be found is stored in the CX register. The AX and BX registers are initialized to 1. Within the loop, the AX and BX are multiplied. The result of the multiplication is stored in the DX:AX register pair, but the DX remains zero and hence our partial multiplication is only in the AX part. Then the BX is incremented and compared with the CX register, that is, N. If it is less than or equal to N, the control is transferred to the beginning of the loop. Otherwise, the loop is completed and the $N!$ (i.e., $1 \times 2 \times 3 \times \cdots \times N$) is computed and stored in the data segment as variable `fact`.

Check Points

We now know that

- decision making and looping are important techniques to write complex programs.
- decision making is similar to the if-else statement in high-level languages and allows the user to select the block of statements for execution.
- looping is similar to the while-do loop in high-level languages and allows the user to execute block of codes multiple times.
- real-life problems involve different combinations of decisions and repetitions and hence decision-making and looping techniques are important to implement the programs for solving them.

4.5 Loop Instructions

The looping instructions, also known as *iteration control instructions*, are used to control the number of iterations to be performed on a specified block of instructions forming the body of a loop. These include the LOOP, LOOPE/LOOPZ and LOOPNE/LOOPNZ instructions as well as special instruction JCXZ to skip the block of instructions when CX is zero. Let us understand these instructions through their use in examples.

```
skip:   DEC DX                  ;decrement DX
        JNZ inner               ;if not 0, goto inner
        DEC CX                  ;decrement CX
        JNZ outer               ;if not 0, goto outer
        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.8, the data segment stores an array with name block with the ... of array as first number followed by elements. This program implements the sorting method in ... for N elements, total of $N-1$ passes are performed, and in each pass the smallest value comes to ... position equal to that pass number. This means that in the first pass the smallest value comes to the ... position, in the second pass the second smallest value comes to the second position and so on. In ... pass, SI points to the position equal to the pass and the value at that position is compared to the ... at positions SI + 1 to the last position pointed by the DI register. At any time if the value pointed ... SI is greater than the value pointed by DI, then they are exchanged. After completing all the pass ... the values are arranged in ascending order. In the above case, $N = 5$ and hence four passes are need... to sort the array. Testing the above program by executing it in the debug mode, the array elements ... look as follows:

```
12h 23h 66h 7Dh A2h
```

Program 4.9

Write a program to find factorial of a given 8-bit number. Give the number such that its factorial is ... than or equal to FFFFh.

Solution

The program is

```
data    SEGMENT
        N    DB    05h
        fact DW ?
data    ENDS

code    SEGMENT
        ASSUME cs:code , ds:data
start:  MOV AX, data            ;initialize
        MOV DS, AX

        MOV CX, 0               ;set CH = 0
        MOV CL, N               ;get N into CL
```

As can be seen from Program 4.7, the data segment stores an array with name `block` with the first number as the size of the array followed by the elements of the array. In the code segment, the SI register after getting the size into CX, points to the first element of the array, and the DI register is initialized to point to the last element in the array by computing offset of last element. To compute the offset of the last element, 2 × (size of array − 1) is added to the start position, that is, SI. The total number of exchanges required is the size of the array divided by 2. The CX register contains the number of exchanges needed before the start of the loop. Then the loop is executed CX times. Each time in the loop, the elements at the offset SI and DI are exchanged, the SI is incremented by 2 to point to the next element, and the DI is decremented by 2 to point to the previous element. At the end of the loop execution, the whole array is reversed. Test the program by executing it in the debug mode and then verifying the contents in the data segment.

Program 4.8

Write a program to sort the given block of 8-bit numbers.

Solution

The program is

```
data    SEGMENT
        block DB 05h, 23h, 0A2h, 66h, 12h, 7Dh
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data              ;initialize
        MOV DS, AX

        LEA SI, block             ;point to size
        MOV CX, 0                 ;get size
        MOV CL, BYTE PTR [SI]     ;into CL
        DEC CX                    ;set no. of passes
outer:  MOV DX, CX                ;set no. of comparisons
        INC SI                    ;set SI pointer
        MOV DI, SI                ;set DI = SI + 1
inner:  INC DI
        MOV AL, BYTE PTR [SI]     ;compare elements at
        CMP AL, BYTE PTR [DI]     ;position SI and DI
        JBE skip                  ;if (SI) < (DI), goto skip
        MOV BL, AL                ;otherwise exchange them
        MOV AL, BYTE PTR [DI]
        MOV BYTE PTR [DI], BL
        MOV BYTE PTR [SI], AL
```

```
start:  MOV AX, data                    ;intialize
        MOV DS, AX

        MOV CX, 5                       ;set count
        MOV SI, OFFSET array            ;set pointer
        MOV AX, WORD PTR [SI]           ;get first no. in AX
        DEC CX                          ;decrement count
        ADD SI, 2                       ;point to next number
next:   CMP AX, WORD PTR [SI]           ;compare with AX
        LOOPE next                      ;repeat if equal

        MOV AH, 9                       ;print message
        JNZ skip                        ;Not same
        LEA DX, mes1                    ;all are same
        JMP print
        LEA DX, mes2
skip:
print:  INT 21h

        MOV AX, 4C00h                   ;terminate
        INT 21h
        ENDS
code    END start
```

As can be seen from Program 4.11, the first number from the array is moved into the AX register and then the rest of the numbers are compared with the number in AX. The control comes out of the loop either when comparison fails, that is, ZF ≠ 1 or CX = 0. Then using the status of the ZF, the appropriate message is printed. In the above case, the message "`All the numbers are same`" is printed as all the numbers in the array `array` are same. If you test the program by changing one of the numbers in the array `array` in the above program, the message "`All the numbers are not same`" gets printed.

LOOPNE/LOOPNZ Instruction

This is known as loop if not equal and performs looping while CX ≠ 0 and ZF = 0. The syntax of LOOPNE/LOOPNZ instruction is as follows:

```
LOOPNE/LOOPNZ target_inst
CX ← CX - 1
;Jump to instruction labeled with target_inst if CX
;and ZF = 0
```

The LOOPNE and LOOPNZ are two different mnemonics for the same instruction. The ... into the CX register. This instruction loops while ... times or until ZF = 1. When it executes, ... = 0 then the control is trans... If CX = 0 or ZF

05h 08h 0Dh 15h

LOOPE/LOOPZ Instruction

This is known as loop if equal and performs looping while CX ≠ 0 and ZF = 1. The syntax of the LOOPE/LOOPZ instruction is as follows:

```
LOOPE/LOOPZ target_inst
CX ← CX - 1
;Jump to instruction labeled with target_inst if CX ≠ 0
;and ZF = 1
```

The LOOPE and LOOPZ are two different mnemonics for the same instruction. The count representing the iteration value is loaded into the CX register. This instruction loops while CX ≠ 0 and ZF = 1. This means that it loops CX number of times or until ZF = 0. When it executes, the CX register is automatically decremented by 1, and if CX ≠ 0 and ZF = 1 then the control is transferred to the instruction whose address is represented by the label *target_inst*. If CX = 0 or ZF = 0 then the control moves to the next instruction after the LOOPE/LOOPZ instruction. The displacement of the target instruction must be in the range of – 128 to + 127 bytes from the address of the instruction after the instruction LOOPE/LOOPZ. This instruction does not affect the flags. The function of LOOPE/LOOPZ instruction can also be represented as follows:

```
CX ← CX - 1
if(CX ≠ 0 and ZF = 1) then
    IP ← IP + displacement
if(CX = 0 or ZF = 0) then
    move to the next instruction
```

The LOOPE/LOOPZ instruction is normally used after the compare instruction. If comparison is equal (ZF = 1) and the CX ≠ 0 then the looping is performed. Let us see the use of the LOOPE/LOOPZ instruction in the following program to check whether all the numbers in an array are same or not.

Program 4.11

Write a program to check whether all the numbers in a given array of 16-bit numbers are same. Print the appropriate message.

Solution

The program is

```
data    SEGMENT
        array DW 1234h, 1234h, 1234h, 1234h, 1234h
        mes1  DB "All the numbers are same $"
        mes2  DB "All the numbers are not same $"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data
```

As can be seen from the above code snippet, for each iteration two instructions are executed, that is, DEC and JNZ, instead of one instruction, that is, LOOP.

Program 4.10

Write a program to generate the first 10 Fibonacci numbers starting from 0 and 1.

Solution

The program is

```
data    SEGMENT
        fib DB 10 dup(0)
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data            ;intialize
        MOV DS, AX
        MOV CX, 10              ;set count
        MOV SI, OFFSET fib      ;set pointer
        MOV AL, 0               ;Fisrt number
        MOV BYTE PTR [SI], AL   ;store
        INC SI                  ;increment pointer
        MOV AL, 1               ;second number
        MOV BYTE PTR [SI], AL   ;store
        INC SI                  ;increment pointer
        SUB CX, 2               ;reduce counter
        MOV AL, 0               ;intialize
        MOV BL, 1
next:   ADD AL, BL              ;get next number
        MOV DL, AL
        MOV BYTE PTR [SI], AL   ;store result
        INC SI                  ;increment pointer
        MOV AL, BL              ;reinitialize
        MOV BL, DL
        LOOP next               ;goto next, if count is not 0
        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.10, the first two numbers, 0 and 1, are stored in the array fib. In the loop, every time the last two numbers in AL and BL registers are added, the result is stored in array fib at the next position and AL and BL are reinitialized. If you see the output in debug, it looks as follows:

```
                INC count          ;bit = 1, increment count
                LOOP next          ;perform next iteration
skip:
                MOV AX, 4C00h      ;terminate the program
                INT 21h
code            ENDS
                END start
```

As can be seen from Program 5.1, the shift left operation each time gets the next MS
If the current MSB in CF = 1, then the counter is incremented. This operation is rep
Finally, the value in the counter is the number of 1 bits in the operand.

Program 5.2

Write a program to count the number of similar bits in the two 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        n1      DW      54A2h
        n2      DW      0AF34h
        count   DB      ?
```

Program 5.1

Write a program to count the number of 1 bits in the binary representation of a given 16-bit operand.

Solution

The program is

```
data    SEGMENT
        num         DW   32AFh
        count       DB    ?
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX
        MOV AX, num         ;load AX with num
        MOV count, 0        ;initialize count = 0
        MOV CX, 16          ;load CX = 16
next:   SHL AX, 1           ;get next bit into CF
        JNC skip            ;if bit = 0, goto skip
        INC count           ;bit = 1, increment count
skip:   LOOP next           ;perform next iteration
        MOV AX, 4C00h       ;terminate the program
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 5.1, the shift left operation each time gets the next MSB into the CF. If the current MSB in CF = 1, then the counter is incremented. This operation is repeated 16 times. Finally, the value in the counter is the number of 1 bits in the operand.

Program 5.2

Write a program to count the number of similar bits in the two 16-bit numbers.

Solution

The program is

```
data    SEGMENT
        n1          DW      54A2h
                    DW      0AF34h
```

Program 4.12

Write a program to generate first 10 Fibonacci numbers in decimal form starting from 0 and 1.

Solution

The program is

```
data    SEGMENT
        fib DB 10 dup(0)
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data            ;intialize
        MOV DS, AX

        MOV CX, 10              ;set count
        MOV SI, OFFSET fib      ;set pointer
        MOV AL, 0               ;Fisrt number
        DAA                     ;convert to BCD
        MOV BYTE PTR [SI], AL ;store
        INC SI                  ;increment pointer
        MOV AL, 1               ;second number
        DAA                     ;convert to BCD
        MOV BYTE PTR [SI], AL ;store
        INC SI                  ;increment pointer
        SUB CX, 2               ;reduce counter
        MOV AL, 0               ;intialize
        MOV BL, 1
next:   ADD AL, BL              ;get next number
        DAA                     ;convert to BCD
        MOV DL, AL
        MOV BYTE PTR [SI], AL ;store result
        INC SI                  ;increment pointer
        MOV AL, BL              ;reinitialize
        MOV BL, DL
        LOOP next               ;goto next, if count is not 0

        MOV AX, 4C00h           ;terminate
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 4.12, the initial two Fibonacci numbers are converted to BCD and stored in the data segment into the array fib. Then in a loop, the addition of the last two Fibonacci numbers

```
        AND DX, 000Fh          ;to separate nibble
        MOV BYTE PTR [SI], DL  ;store the nibble
        INC SI                 ;increment pointer
        MOV CL,4               ;make CL = 4
        DEC BX                 ;decrement count
        JNZ next               ;goto next, if not zero

        MOV AX, 4C00h          ;terminate the program
        INT 21h

code    ENDS
        END start
```

As can be seen from Program 5.3, the nibble is separated from the operand by performing logical AND with `000Fh`. It is stored in the array pointed by SI. Then the operand is shifted four times right each time to get the next nibble at the lower 4-bit positions, ANDed with `000Fh` and the separated nibble is stored in the array at the next position. Run the program in trace mode and visualize the results.

SAR Instruction

The syntax of the shift arithmetic right (SAR) instruction is as follows:

```
SAR destination, count
(MSB) → (MSB), (B_{n+1}) → (B_n), (LSB) → (CF)
;Shift the destination logically right count times
 arithmetically
```

The *destination* can be either a register or a memory location. If the value of *count* is 1, then it is given as immediate value, otherwise it is specified in the CL register. The contents of *destination* are shifted right by *count* number of times with the MSB copied to the MSB to preserve the sign of the operand, and the LSBs are shifted right into the CF. The CF finally stores only the most recently shifted bit if shifting is done multiple times. This is shown in Fig. 3.

Assume that the *count* is 1, as seen in the figure, each bit of the operand is shifted right by 1 position, the MSB is copied into the MSB and the LSB is shifted into the CF. If the *count* is more than 1 then the same operation is repeated *count* times. All the conditional flags are affected by this instruction except the AF which is undefined for it. The PF is defined on the lower byte of the destination. Following are some examples:

1 position, 0 ... the operand is shifted right by
then the same operation is repeated count times. All the conditional flags are affected by this instruc-
tion except the AF which is undefined for it. The PF is defined on the lower byte of the destination.
Following are some examples:

```
SHR AX, 1                ;AX=0100 0011 0011 1011,CF=1 before
                         ;AX=0010 0001 1001 1101,CF=1 after

MOV CL, 2                ;load the count CL=2
SHR AL, CL               ;AL=0110 1010,CF=0, before
                         ;AL=0001 1010,CF=1 after

SHR WORD PTR [SI],CL             ;shift right the word at offset
                                 ;SI by CL times
```

Shifting right by 1 position is the same as division by 2. Similarly, if the shift is performed by 2 positions right, it is same as division by 4. This means that the shift instruction creates the effect of division by the power of 2. It performs the operation much faster than the DIV instruction. Hence if we need to divide the operand by power of 2, then it is better to use SHR to get improved performance. Remember that it will perform the truncated division.

Program 5.3

Write a program to separate the nibbles of a 16-bit number.

Solution

The program is

```
data    SEGMENT
        num         DW      54A2h
        nibbles     DB      4 dup(0)
data    ENDS

code    SEGMENT
        ASSUME  cs:code, ds:data

start:  MOV AX,  data           ;initialize the data segment
        MOV DS,  AX

        MOV AX,  num                    ;load AX with n1
        MOV CL,  0                      ;initialize CL = 0
        MOV BX,  4                      ;initialize BX = 4
        LEA SI,  nibbles                ;point to nibbles
next:   SHR AX,  CL                     ;shift 4 times right
        MOV DX,  AX                     ;AND with 000Fh
```

```
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data          ;initialize the data segment
        MOV DS, AX

        MOV AX, n1            ;load AX with n1
        MOV BX, n2            ;load BX with n2
        XOR AX, BX            ;perform ex-or

        MOV count, 0          ;initialize count = 0
        MOV CX, 16            ;load CX = 16
next:   SHL AX, 1             ;get next bit into CF
        JC skip               ;if bit = 1, goto skip
        INC count             ;incerement count
skip:   LOOP next             ;perform next iteration

        MOV AX, 4C00h         ;terminate the program
        INT 21h

code    ENDS
        END start
```

As can be seen from Program 5.2, first the XOR of both the operands is performed as EX-OR operation results into bit 0 if the bits at the same position in both the operands are equal. Then from the result of XOR, the 0 bits are counted to find the count of similar bits in both the operands. To count the number of 0 bits, we used the same logic as in Program 5.1.

SHR Instruction

The syntax of the shift right (SHR) instruction is as follows:

```
SHR destination, count
0 → (MSB), (B_{n+1}) → (B_n), (LSB) → (CF)
;Shift the destination logically right count times
```

The *destination* can be either a register or a memory location. If the value of *count* is 1, then it is given as immediate value, otherwise it is specified in the CL register. The contents of *destination* are shifted right by *count* number of times with 0s shifted in from the left end into the MSB position, and the LSBs are shifted out from the right end into the most recently shifted

```
        ;PART-1 Convert from hex string to binary
        MOV AH, 09h                ;print message
        LEA DX, mes1
        INT 21h
        MOV AH, 0ah                ;read hex string
        LEA DX, buf
        INT 21h
        LEA SI, buf+1              ;get length of string
        MOV BX, 0
        MOV BL, [si]               ;point to first char
        INC SI                     ;initialize
        MOV CX, 4
        MOV AX, 0
        MOV DX, 0
next:   MOV DL, [si]               ;get next ASCII digit
        CMP DL, '9'                ;convert it to binary
        JLE digit
        SUB DL, 7
digit:  SUB DL, '0'
        SHL AX, CL                 ;shift the partial conversion
        ADD AX, DX                 ;add current digit
        INC SI                     ;increment pointer
        DEC BX                     ;decrement length counter
        JNZ next                   ;if not zero, goto next
        PUSH AX                    ;push binary number on stack
        ;PART-2 Convert binary to decimal string
        MOV AH, 09h                ;print message
        LEA DX, mes2
        INT 21h
        POP AX                     ;get binary number back
        MOV CX, 0                  ;set counter to 0
        MOV BX, 10                 ;set divider
next1:  MOV DX, 0
        DIV BX                     ;divide number by divider
        PUSH DX                    ;push decimal digit on stack
        INC CX                     ;increment counter
        CMP AX, 0                  ;f number is not zero
        JA next1                   ;goto next1

        MOV AH, 2
```

We know that the data can be represented in many different formats. For example, characters are normally stored in the ASCII format. Strings are stored as sequence of ASCII characters. Numbers can be represented using the binary, decimal, octal or hexadecimal systems. Decimal numbers can also be represented in computers using the Binary Coded Decimal (BCD) form. Many a time, we need to convert the data represented in one format into the other. Let us take the example of how to convert a given hex number into a decimal number. This conversion follows the following steps:

1. Get the hex number as a string. We will accept the hex number with maximum 4 digits so that we can use the 16-bit register to store its binary conversion. For simplicity, we will accept 0 to 9 and only uppercase letters (A to F) for hex digits.
2. For each hex digit, perform the following operations:
   (a) Convert current hex digit into a binary by subtracting ASCII of 0 if it is "0" to "9". If it is "A" to "F," subtract ASCII of 0 plus 7.
   (b) Shift the partial answer left by 4 positions (shifting left 4 times is equal to multiplying by 16) and add the binary of the hex digit converted using step 2(a).
3. Store your binary number.
4. Repeat until the binary number is zero.
   (a) Divide it by 10 to separate the last digit, push it onto the stack.
5. For each digit onto the stack,
   (a) Pop it from the stack, convert it into the ASCII format by adding ASCII of 0 and display it.
6. Finally, the decimal number is displayed.

Steps 1–3 convert the given hex string into a binary number and Steps 4–6 convert the binary number into a decimal string and display it. Program 5.5 performs this conversion. Part 1 of this program implements Steps 1–3 and Part 2 implements Steps 4–6.

Program 5.5

Write a program to convert a given hex number into a decimal number.

Solution

The program is

```
data    SEGMENT
        mes1 DB "Enter hex number : $"
        buf  DB 5,6 dup(0)
        mes2 DB 0Dh,0Ah,"Decimal of given hex is : $"
data    ENDS

code    SEGMENT
        ASSUME cs:code,ds:data

start:  MOV AX, data                ;initialize data segment
        MOV DS, AX
```

Assume that the count is 1, ... count is more than ... rotated left ...
1 position; the MSB is rotated into the LSB and ...
then the same operation is repeated count times. This instruction affects only the CF and OF ...
CF contains the most recently rotated MSB. The OF will become 1 if the MSB is changed in a single ...
bit rotation, and it remains undefined for the multiple bit rotation. Following are some examples:

```
ROL AX, 1                ;AX=1011 1100 1010 0011, CF=0 before
                         ;AX=0111 1001 0100 0111, CF=1 after
ROL BX, CL               ;rotate left BX by CL times
ROL WORD PTR [BX], CL    ;rotate left word at BX
                         ;by CL times
```

Program 5.4

Write a program to exchange the nibbles of a given byte.

Solution

The program is

```
data    SEGMENT
        num DB 3Fh
        ans DB  ?
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data
start:  MOV AX, data        ;initialize the data segment
        MOV DS, AX
        MOV AL, num         ;get num into AL
        MOV CL, 4           ;load CL with 4
        ROL AL, CL          ;rotate left 4 times
        MOV ans, AL         ;store answer
        MOV AX, 4C00h       ;terminate the program
        INT 21h
code    ENDS
        END start
```

As can be seen from Program 5.4, the nibbles of given byte are exchanged simply by rotating the byte left four times.

ROR Instruction

The syntax of the rotate right (ROR) instruction is as follows:

the BIOS date ... with the "$" sign and then print it. ... to define the data segment to point to the physical segment with segment base FFFFh. We have to read ... 0005h, we have ... eight locations in that segment from the offset 0005h. Program 5.7 performs the task of printing the ... of BIOS. This program defines a data segment with the name "extra" to define the user space. The ... able `str1` is defined with 8 empty locations followed by "$" sign to end the string. The DS points to the logical ... segment with the base address FFFFh and SI points to offset 0005h. The ES points to the ... "extra" and the DI points to the offset of `str1`. Then the LODSB transfers a byte from the DS:SI to the ... AL and the STOSB stores byte in the AL to the ES:DI. The SI and DI both are auto-incremented by 1 ... DF is initialized to 0. These operations are repeated 8 times to copy the whole data. At the end, the ... tents of `str1`, that is BIOS date transferred from the system memory area to the user space is printed.

Program 5.7

Write a program to display the date of BIOS of your computer system.

Solution

The program is

```
extra   SEGMENT
        str1 DB 8 dup(0),'$'
extra   ENDS
code    SEGMENT
        ASSUME cs:code, es:extra

start:  MOV AX, extra      ;initialize the extra segment
        MOV ES, AX
        MOV DI, offset str1    ;set destination index

        MOV AX, 0FFFFh     ;initialize data segment
        MOV DS, AX         ;into system memory
        MOV SI, 0005h      ;set the source offset

        CLD                ;clear direction flag
        MOV CX, 8          ;set counter

next:   LODSB              ;load a byte from system memory
        STOSB              ;store a byte into extra segment
        LOOP next          ;repeat the operation

        MOV AH, 9          ;display the data from extra segment
        MOV BX, SEG str1
        MOV DS, BX
        LEA DX, str1
        INT 21h
```

Program 5.6

Write a program to copy a string defined in the data segment.

Solution

The program is

```
data    SEGMENT
        str1 DB "Hello, World$"
        str2 DB 12 dup(0),'$'
        ENDS
data
code    SEGMENT
        ASSUME cs:code, ds:data, es:data
start:  MOV AX, data                ;initialize the data segment
        MOV DS, AX
        MOV ES, AX                  ;initialize the extra segment
        MOV SI, offset str1         ;set source index
        MOV DI, offset str2         ;set destination index
        CLD                         ;clear direction flag
        MOV CX, 12                  ;count = 12
REP     MOVSB                       ;repeat move instruction
        MOV AH, 9                   ;print the copied string
        LEA DX, str2
        INT 21h
        MOV AX, 4C00h               ;terminate the program
        INT 21h
code    ENDS
        END start
```

We know that the string instruction move needs source string in the data segment and destination string in the extra segment. As can be seen from Program 5.6, we have used the logical segment `data` as both data and extra segments by initializing the DS and ES registers with the same segment base. Then the program initializes the SI to the offset of the `str1` and DI to the offset of `str2`. Thus, initially, the SI points to the first byte in the source string and the DI points to the first free location in the destination string. The direction flag (DF) is clear in order to auto-increment the SI and DI after the move operation to move the SI to the next character and the DI to the next free location. The repeat prefix repeats the move operation 12 times to copy the whole string as CX is initialized to 12. Finally, the copied version of the string is printed as follows:

```
Hello, World
```

```
next2:  POP DX
        ADD DX,'0'              ;pop the decimal digit
        INT 21h                 ;convert to ASCII
        LOOP next2              ;print on screen

        MOV AX,4C00h
        INT 21h                 ;terminate the program

code    ENDS
        END start
```

After assembling and linking the above program when executed, it produces the response as follows:

```
Enter hex number : FFFF
Decimal of given hex is : 65535
```

As can be seen from Program 5.5, although the binary number is 16-bit, we have used the 32-bit/16-bit division. The reason is that the remainder, that is the last decimal digit, comes out in the DX register which is then pushed onto the stack. Another reason is that the PUSH and POP instructions work with words only. We have used the PUSH and POP instructions without defining the stack segment. The default definition of stack segment is sufficient for this example. We will learn about the stack and stack-related instructions in Chapter 6.

A small change in Program 5.5 allows us to perform different conversions. Program 5.5 accepts the input in hex and produces the output in decimal. Keeping the input in hex only, the output can be changed to octal by changing the following statement

```
MOV BX, 10  ;set divider
```

in Part 2 to

```
MOV BX, 8  ;set divider
```

Changing the divider from 10 to 8 separates the octal digits from the binary number, and the final output string contains only octal digits. Thus, the small change now converts from hex to octal. To convert from hex to binary, simply change the divisor value in the above statement to 2.

Similarly, by making a small change in Part 1, we can change the input form, keeping the output in the decimal form. For example, to convert from octal to decimal, change the following statement:

```
MOV CX, 4  ;initialize
```

in Part 1 to

```
MOV CX, 3  ;initialize
```

For octal input, shifting by 3 is needed as it is equal to multiplication by 8. If we change shift count to 1, it is equal to multiplication by 2 and the conversion is performed from binary to hex. Remember the size of the input buffer is defined to accommodate maximum FOUR digits. For specific conversion, if the input string requires more digits, we have to change the buffer size to a large value. If we want the input to be decimal, merely changing the shift value in the above statement will not work because multiplication by 10 cannot be done by shifting. If we want to change both the input and output formats, we need to make appropriate changes in both the parts for the shift count and the divisor.

Let us take an example of printing the message "Hello, World" five times. Program 6.1 implements it using a procedure "prt_mes" and calling it five times in a loop.

Program 6.1

Write a procedure to print the message "Hello, World" and use it to print the message five times.

Solution

The program is

```
data        SEGMENT
            mes DB "Hello, World",0dh,0ah,'$'
data        ENDS

my_stack    SEGMENT STACK
            DW 10 dup(0)
stack_top   LABEL WORD
my_stack    ENDS

code        SEGMENT
            ASSUME cs:code, ds:data, ss:my_stack

            ;mainline program
start:      MOV AX, data              ;initialize data segment
            MOV DS, AX
            MOV AX, my_stack          ;initialize SS
            MOV SS, AX
            MOV SP, OFFSET stack_top      ;initialize SP

            MOV CX, 5                 ;set count
next:       CALL prt_mes              ;call procedure
            LOOP next

            MOV AX, 4C00h             ;terminate program
            INT 21h

            ;procedure
prt_mes     PROC NEAR
            MOV AH, 9                 ;print message
            LEA DX, mes
            INT 21h
            RET                       ;return
prt_mes     ENDP

code        ENDS
            END start
```

Similarly, for a word version we can use the CMPSW. By storing the count in the CX register and using an appropriate repeat prefix, we can use the compare string instruction to compare two strings. For example, REPE CMPSB means compare while CX is not 0 and ZF = 1.

Program 5.8

Write a program to compare two strings of equal length.

Solution

The program is

```
data    SEGMENT
        str1 DB "Computer"
        str2 DB "Computer"
        mes1 DB "Strings are same$"
        mes2 DB "Strings are different$"
data    ENDS
code    SEGMENT
        ASSUME cs:code, ds:data, es:data
start:  MOV ax, data                ;initialize
        MOV ds, ax
        MOV es, ax
        MOV SI, offset str1         ;set the source index
        MOV DI, offset str2         ;set the destination index
        CLD                         ;clear DF
        MOV CX, 8                   ;set the CX with length
REPE    CMPSB                       ;compare
        MOV AH, 9                   ;print message
        JZ skip
        LEA DX, mes2                ;different
        JMP over
skip:   LEA DX, mes1                ;same
over:   INT 21h
        MOV AX, 4C00h               ;terminate program
        INT 21h
code    ENDS
        END start
```

As can be seen in Program 5.8, after initializing the index registers and the clearing direction flag (D the CMPSB instruction is repeated by REPE as long as the characters in both the strings pointed by DS:SI and the ES:DI are same. Whenever the characters are not same, ZF becomes 0 and it comes of repetition, or when the CX is 0, that is, strings are ended then it comes out. If both the string same, then after coming out of comparison, ZF = 1, otherwise ZF = 0. Using the status of ZF, o the messages (mes1 or mes2) is printed. In our case, the output is

```
Strings are same
```

```
        MOV AX, 4C00h     ;terminate the program
        INT 21h

code    ENDS
        END start
```

CMPS/CMPSB/CMPSW Instruction

This is known as compare string instruction. It is used to compare a string byte or a string word in the source string with a string byte or a string word in the destination string. The syntax of the compare string instruction is as follows:

```
CMPS/CMPSB/CMPSW
;compare a byte or word in the source (DS:SI) with the destination
;(ES:DI)
if (DF = 0) then
    for byte
            SI = SI + 1 and DI = DI + 1

    for word
            SI = SI + 2 and DI = DI + 2

if (DF = 1) then
    for byte
            SI = SI - 1 and DI = DI - 1

    for word
            SI = SI - 2 and DI = DI - 2
```

The mnemonic CMPS represents the compare string instruction and compares a string byte or a string word stored at the offset address SI in the data segment with a string byte or a string word stored at the offset address DI in the extra segment. This means that the data segment acts as the source segment and the extra segment acts as the destination segment. The offset address in the source segment is specified by the source index SI register and the offset address in the destination in the extra segment is specified by the destination index DI register. The CMPS instruction affects the flags based on comparison in the same way as the CMP instruction, but does not change the source and destination bytes or words. After comparing a byte or a word, the source index and the destination index are automatically incremented by 1 for a byte and by 2 for a word if the direction flag (DF) is set to 0, otherwise decremented by 1 (for DF = 1). The CMPS does not automatically initialize anything and hence before we execute it, source and destination segments, the source and destination index registers and the DF must be initialized properly. The CMPS identifies the type of string (byte or word) from the definition of the using the directive DB or DW. For example, the following code illustrates the use of CMPS instru

```
MOV SI, offset str1     ;set source index
MOV DI, offset str2     ;set destination index
CLD                     ;set DF = 0
CMPS str2, str1         ;compare str1 with str2
```

If the source and destination strings are defined as byte strings, we can use the implicit version string comparison. In such a case, we can replace the last instruction with following instruc

```
CMPSB                   ;compare str1 with str2
```

flexibly from anywhere with
(1) using the registers and (2) using the stack.
with programming examples.

Parameters Passing Using Registers

In this method, the general-purpose registers are used to pass the parameters to the procedure and return the value from the procedure. That means before calling the procedure, the parameters are copied into the specific register(s) which then in the beginning of the procedure body are read from the registers and used. The procedure, before returning to the caller, copies the return value in a specific register which then is received from the register by the caller after getting back the control. The following programs demonstrate the working of this method:

Program 6.3

Write a program to count the number of 1's in the binary representation of a 16-bit number using procedure.

Solution

The program is

```
data        SEGMENT
            num    DW 0AAAAh
            count DB   ?
data        ENDS

my_stack    SEGMENT STACK
            DW 20 dup(0)
stack_top   LABEL WORD
my_stack    ENDS

code        SEGMENT
            ASSUME  cs:code, ds:data, ss:my_stack

            ;mainline program
start:      MOV  AX,  data      ;initialize
            MOV  DS,  AX
            MOV  AX,  my_stack
            MOV  SS,  AX
            MOV  SP,  OFFSET stack_top
```

```
                ;procedure
prt_mes         PROC NEAR
                MOV AH, 9          ;print message
                LEA DX, mes
                INT 21h
                RET                ;return
prt_mes         ENDP

code            ENDS
                END main
```

As can be seen from Program 6.2, the code segment contains two procedures: `main` and `prt_mes`. The `main` procedure is now the entry point of the program and should be called by the operating system which then calls the `prt_mes` procedure five times. The important point to be noted is that the entry point of the program is now `main` procedure and hence the END directive in last statement is followed by its name.

Stack and Procedure

We have used the stack in Program 6.1 through the CALL and RET instructions, not explicitly using instructions like PUSH and POP. The CALL instruction pushes the address of the next instruction on the stack in the same manner as the PUSH instruction, that is, it reduces the SP register by 2 and copies the current IP (the offset of the next instruction) at the new top position and then transfers the control to the procedure by copying the offset of the first instruction of the procedure into the IP register. At the end of the procedure, the RET instruction copies the word from the top of the stack to the IP register which is the offset of the next instruction after the CALL instruction. This ensures that after completion of the procedure, the execution resumes from the next instruction after the CALL instruction.

Both the mainline program and the procedure use the same set of registers, including flags, to carry out operations. It is obvious in that case the instructions in the body of the procedure may overwrite the values computed by the mainline program. This may cause the problem in the execution of the mainline program after the CALL instruction, as it resumes with different values of registers and flags than the values computed by the mainline program before the CALL instruction. In Program 6.1, it was not the issue as both the mainline program and the procedure used different registers. It is not possible always as there are limited numbers of registers. The solution is that before using the registers in the body of the procedure, they should be saved onto the stack, and before returning from the procedure all the register values should be restored. This ensures that the mainline program resumes the execution after calling and returning from the procedure with the same status. In general, whenever a procedure call occurs, the status of the caller is stored and it is restored back while returning from the procedure. The status includes the return address, all the registers and flags. The CALL and RET return address, so we need to worry about only registers inclu
edure body, we should save the flag register a
the task defined by the procedu

As can be seen from ... stack of 10 words. The stack is used by ... to save and retrieve ... return address. The code segment is internally divided into two parts: the mainline program and the ... cedure prt_mes. The procedure simply prints the message when called and returns back. The mainline ... program first initializes the data and stack segments, calls the procedure prt_mes five times in a ... and then terminates the program. Observe that the procedure is written at the bottom of the segment ... the mainline program. It is a good practice to write all the procedures after the mainline program. ... It is also possible to write the mainline program as a procedure which calls other procedures defined ... in the code segment. That procedure must be far as it is called by the operating system from outside ... the code segment. The details of the near and far procedures are discussed later. Program 6.2 written ... Program 6.1 with the mainline program as the far procedure.

Program 6.2

Rewrite Program 6.1 with the mainline program as the far procedure.

Solution

The program is

```
data        SEGMENT
            mes DB "Hello, World",0dh,0ah,'$'
data        ENDS

my_stack        SEGMENT STACK
                DW 10 dup(0)
stack_top       LABEL WORD
my_stack        ENDS

code        SEGMENT
            ASSUME cs:code, ds:data, ss:my_stack

            ;mainline program as procedure
main        PROC far
            MOV AX, data        ;initialize data segment
            MOV DS, AX
            MOV AX, my_stack    ;initialize SS
            MOV SS, AX
            MOV SP, OFFSET stack_top      ;initialize SP

            MOV CX, 5
next:       CALL prt_mes        ;set count
            LOOP next           ;call procedure

            MOV AX, 4C00h
            INT 21h             ;terminate program
main        ENDP
```

```
        CMP AX, BX                  ;compare and
        JC skip                     ;store minimum
        MOV AX, BX                  ;in AX register

skip:   POPF                        ;pop flags
        RET                         ;return
p_min   ENDP

code    ENDS
        END start
```

As can be seen from Program 6.4, the p_min procedure finds the minimum from the two numbers passed as parameters using the AX and BX registers and returns the minimum of those two numbers in the AX register. The mainline program calls the p_min procedure repeatedly to find the minimum from the array. Finally, the minimum from the array, available in the AX register, is stored in the data segment.

Both the above examples pass the data values as parameters through the registers. This is very similar to the call-by-value method of parameter passing used by most of the programming languages, including C language. The method is very simple and easy to understand. However, the limitation of this method is that there are only limited numbers of registers available to each processor and hence we have a limit on the numbers of parameters using this method.

Sometimes the number of values to be passed to the procedure is more and sometimes it is either difficult or impossible to pass the parameters through the registers directly just we did in above examples. For example, suppose we want to pass the whole array as a parameter. In this case, rather than passing the values using registers, we can pass the address of the array, that is, the pointer to an array as parameter. The procedure uses this pointer to access the whole array. This method also uses the registers but passes addresses rather than the values themselves. Hence it is also sometimes known as *parameter passing using pointers*. The following program demonstrates this concept:

Program 6.5

Write a procedure to find minimum from the array of N values and call it in the mainline program. Implement the procedure to use the array address as parameter.

Solution

The program is

```
data    SEGMENT
        block DW 0004, 4523h, 0A2D3h, 1345h, 78D2h
        min   DW  ?
data    ENDS

my_stack    SEGMENT STACK
            DW 20 dup(0)
stack_top   LABEL WORD
```

Program 6.4

Write a procedure to find minimum of two numbers using register parameter passing. Use it to find the minimum from the array of N numbers.

Solution

The program is

```
data      SEGMENT
          block DW 0004, 4523h, 0A2D3h, 1345h, 78D2h
          min   DW   ?
data      ENDS

my_stack  SEGMENT STACK
          DW 20 dup(0)
stack_top LABEL WORD
my_stack  ENDS

code SEGMENT
     ASSUME cs:code, ds:data, ss:my_stack

     ;mainline program
start: MOV AX, data                    ;initialize
     MOV DS, AX
     MOV AX, my_stack
     MOV SS, AX
     MOV SP, OFFSET stack_top

     LEA SI, block                     ;set pointer
     MOV CX, WORD PTR [SI]             ;get size
     DEC CX                            ;decrement CX
     ADD SI, 2                         ;point to first element
     MOV AX, WORD PTR [SI]             ;get first num in AX
     ADD SI, 2                         ;point to second element
next: MOV BX, WORD PTR [SI]            ;store next num in BX
     ADD SI, 2                         ;point to next num
     CALL p_min                        ;call procedure
     LOOP next                         ;repeat
     MOV min, AX                       ;store minimum

     MOV AX, 4C00h
     INT 21h                           ;terminate

     ;procedure to find minimum of two numbers
p_min PROC NEAR
     PUSHF
                                       ;push flags
```

```
        MOV AX,   num     ;copy parameter in AX
        CALL bin_cnt      ;call the procedure
        MOV count, AL     ;get the count

        MOV AX, 4C00h     ;terminate
        INT 21h

        ;procedure to count number of 1's
bin_cnt PROC NEAR
        PUSHF             ;push flags and
        PUSH BX           ;registers
        PUSH CX
        PUSH DX

        MOV BX, AX        ;copy parameter
        MOV CX, 16        ;set shift counter
        MOV DX, 0         ;set count in DL = 0
        SHR BX, 1         ;shift
next:   JNC skip          ;if 0, don't count
        INC DL            ;increment counter
        LOOP next         ;goto next
skip:   MOV AL, DL        ;return count in AL

        POP DX            ;pop registers
        POP CX            ;and flags
        POP BX
        POPF
        RET               ;return
bin_cnt ENDP

code    ENDS
        END start
```

As can be seen from Program 6.3, the data segment defines the variable num to store the 16-bit number from whose binary representation we want to count the 1s. The AX register is used as parameter and hence before calling the `bin_cnt` procedure in the mainline program, num is copied to the AX register. Then the call to the procedure `bin_cnt` passes control to the procedure. Within the procedure, after pushing flags and registers, the parameter is copied to the BX register from the AX register, the 1s are counted in the DL register and finally result in the DL register is copied into the AL register. Thus, AL acts as a return parameter. After returning from the procedure; the mainline program copies the result from the AL register to the variable `count` in the data segment.

As can be seen from Program 6.5, before the procedure is called the address of the array is copied into the SI register and the address of the variable `min` is copied into the DI register. Thus, we are passing the pointers to the array and a variable using registers rather than values. The `p_min` procedure accesses the array using the SI pointer, finds the minimum and stores the result into the variable `min` defined in data segment using the DI pointer.

The parameter passing using pointers is very useful especially when we need to pass complex data structures like arrays, lists, tables, etc. to the procedure.

Parameter Passing Using Stack

This is another widely used method of passing parameters between procedures. Most of the programming languages use this method of parameter passing including the C language. Before the procedure call, parameters are pushed onto the stack, which are then accessed by the called procedure from the stack. Similarly, the procedure stores the return value in the stack which caller reads from the stack after getting the control back. Let us rewrite Program 6.3 through the parameter passing using the stack method.

Program 6.6

Rewrite the procedure in Program 6.3 with the use of stack to pass the parameter and call it from mainline program.

Solution

The program is

```
data        SEGMENT
            num    DW 0AAAAh
            count DB ?
data        ENDS

my_stack    SEGMENT STACK
            DW 20 dup(0)
stack_top   LABEL WORD
my_stack    ENDS

code        SEGMENT
            ASSUME cs:code, ds:data, ss:my_stack

            ;mainline program
start:      MOV AX, data                    ;initialize
            MOV DS, AX
            MOV AX, my_stack
            MOV SS, AX
            MOV SP, OFFSET stack_top

            MOV AX, num                     ;copy parameter in AX
            PUSH AX                         ;push AX on stack
```

```
code SEGMENT
        ASSUME cs:code, ds:data, ss:my_stack

        ;mainline program              ;initialize
start:  MOV AX, data
        MOV DS, AX
        MOV AX, my_stack
        MOV SS, AX
        MOV SP, OFFSET stack_top

        LEA SI, block                  ;copy pointer to array
        LEA DI, min                    ;copy pointer to min
        CALL p_min                     ;call procedure

        MOV AX, 4C00h                  ;terminate
        INT 21h

        ;procedure to find minimum from array
p_min   PROC NEAR
        PUSHF                          ;push flags
        PUSH AX                        ;and registers
        PUSH CX
        PUSH SI

        MOV CX, WORD PTR [SI]   ;find minimum
        DEC CX
        ADD SI, 2
        MOV AX, WORD PTR [SI]
next:   ADD SI, 2
        CMP AX, WORD PTR [SI]
        JC skip
        MOV AX, WORD PTR [SI]
skip:   LOOP next
        MOV WORD PTR [DI], AX

        POP SI                         ;pop registers
        POP CX                         ;and flags
        POP AX
        POPF
        RET
p_min   ENDP                           ;return

code    ENDS
        END start
```

As can be seen from Program 6.4, the p_min procedure finds the minimum from the two numbers passed as parameters using the AX and BX registers and returns the minimum of those two numbers in the AX register. The mainline program calls the p_min procedure repeatedly to find the minimum from the array. Finally, the minimum from the array, available in the AX register, is stored in the data segment.

Both the above examples pass the data values as parameters through the registers. This is very similar to the call-by-value method of parameter passing used by most of the programming languages, including C language. The method is very simple and easy to understand. However, the limitation of this method is that there are only limited numbers of registers available to each processor and hence we have a limit on the numbers of parameters using this method.

Sometimes the number of values to be passed to the procedure is more and sometimes it is either difficult or impossible to pass the parameters through the registers directly just we did in above examples. For example, suppose we want to pass the whole array as a parameter. In this case, rather than passing the values using registers, we can pass the address of the array, that is, the pointer to an array as parameter. The procedure uses this pointer to access the whole array. This method also uses the registers but passes addresses rather than the values themselves. Hence it is also sometimes known as *parameter passing using pointers*. The following program demonstrates this concept:

Program 6.5

Write a procedure to find minimum from the array of N values and call it in the mainline program. Implement the procedure to use the array address as parameter.

Solution

The program is

```
data        SEGMENT
            block DW 0004, 4523h, 0A2D3h, 1345h, 78D2h
            min   DW  ?
data        ENDS

my_stack    SEGMENT STACK
             DW 20 dup(0)
stack_top   LABEL WORD
my_stack ENDS
```

```
;code segment containing mainline program
code    SEGMENT PUBLIC
        ASSUME cs:code, ds:data, ss:my_stack

        ;mainline program
start:  MOV AX, data
        MOV DS, AX                      ;initialize
        MOV AX, my_stack
        MOV SS, AX
        MOV SP, OFFSET stack_top

        CALL bin_cnt                    ;call the procedure
        MOV count, AL                   ;get the count

        MOV AX, 4C00h                   ;terminate
        INT 21h

code    ENDS
        END start
```

The second module written in the *proc.asm* file is as follows:

```
;procedure module - proc.asm

;variable num is defined in other module
data    SEGMENT PUBLIC
        EXTRN num:word

data    ENDS

PUBLIC bin_cnt ;procedure bin_cnt declared public

;code segment defining far procedure
code_proc SEGMENT PUBLIC

        ;procedure to count number of 1's
bin_cnt PROC FAR
        ASSUME cs:code_proc, ds:data
        PUSHF                   ;push flag and
        PUSH BX                 ;registers
        PUSH CX
        PUSH DX

        MOV BX, num             ;copy parameter
        MOV CX, 16              ;set shift counter
        MOV DX, 0               ;set count in DL = 0
```

Let us rewrite Program 6.3 using the far procedure which computes the number of 1s in a 16-b value as shown in Program 6.7. This program demonstrates the use of the above directives by writi program in two assembly modules. The first module defines variables, stack space and contains mainline program. It is written in the *main.asm* file. The second module contains the far procedure is written in the *proc.asm* file. The mainline program in the first module calls the far procedure from second module. The necessary PUBLIC and EXTRN declarations are made in respective modules.

Program 6.7

Develop a far procedure to compute the number of 1s in a given 16-bit value in a module the file name proc.asm. Call it from the module written with the file name main.proc. Asse both the modules independently and link them together to generate an executable progra main.exe.

Solution

The program is

The first module written in the *main.asm* file is as follows:

```
;main module - main.asm

data      SEGMENT PUBLIC
          num     DW 0AAAAh
          count DB ?
data      ENDS

my_stack      SEGMENT STACK
               DW 20 dup(0)
stack_top     LABEL WORD
my_stack      ENDS

PUBLIC num  ;num is accessible in other modules

;procedure bin_cnt is defined in other module
code_proc SEGMENT PUBLIC
        EXTRN bin_cnt:FAR
code_proc ENDS
```

```
                CALL bin_cnt                ;call the procedure
                POP AX                      ;pop the result
                MOV count, AL               ;get the count

                MOV AX, 4C00h               ;terminate
                INT 21h

                ;procedure to count number of 1's
bin_cnt         PROC NEAR                   ;push flags
                PUSHF                       ;and registers
                PUSH AX
                PUSH CX
                PUSH DX
                PUSH BP

                MOV BP, SP                  ;copy SP to BP
                MOV AX, WORD PTR [BP+12]    ;get the parameter

                MOV CX, 16                  ;set shift counter
                MOV DX, 0                   ;set count in DL = 0
                SHR AX, 1                   ;shift
next:           JNC skip                    ;if 0, don't count
                INC DL                      ;increment counter
skip:           LOOP next                   ;goto next
                MOV AX, DX                  ;return count in AL
                MOV WORD PTR [BP+12], AX    ;store result

                POP DX                      ;pop registers
                POP CX                      ;and flags
                POP AX
                POP BP
                POPF
                RET
bin_cnt         ENDP                        ;return

code            ENDS
                END start
```

As can be seen from Program 6.6, within the mainline program, after initializing the data segment, the stack segment and the stack pointer, the num parameter is copied to the AX register and then pushed onto the stack. Figure 6(a) shows the position of the stack after pushing the num parameter. Then the CALL bin_cnt instruction pushes the IP address of the next instruction onto the stack and the control is transferred to the bin_cnt procedure.

parameter as it is replaced by the message defined in the data segment with name mes1, the macro call is as follows:

```
PRINT   mes1
```

The mes1 is an actual parameter and it replaces the dummy parameter mes during the macro expansion. The above macro call is replaced by the following sequence of instructions:

```
MOV AH, 9
LEA DX,  mes1
INT 21h
```

Program 6.9 demonstrates the use of above macro by calling it two times to print the messages defined as mes1 and mes2.

Program 6.9

Write a macro to print a given message. Demonstrate its use in the mainline program.

Solution

The program is

```
;macro to print the given message
PRINT   MACRO mes
        MOV AH, 9
        LEA DX, mes
        INT 21h
        ENDM

data    SEGMENT
        mes1 DB "Hello, World!",0dh,0ah,'$'
        mes2 DB "I am fine.$"
data    ENDS

code    SEGMENT
        ASSUME cs:code, ds:data

Start:  MOV AX, data          ;initialize
        MOV DS, AX

        PRINT mes1            ;print mes1
        PRINT mes2            ;print mes2

        MOV AX, 4C00h         ;terminate
        INT 21h

code    ENDS
        END start
```

Program 6.8

Write a macro to exchange the contents of the AX and BX registers. Use it to exchange the contents of the two numbers defined in the data segment.

Solution

The program is

```
;macro to exchange contents of AX and BX
EXCH    MACRO
        MOV CX, AX
        MOV AX, BX
        MOV BX, CX
        ENDM

data    SEGMENT
        num1 DW 1234h
        num2 DW 8756h
data    ENDS

code SEGMENT
        ASSUME cs:code, ds:data

start:  MOV AX, data          ;initialize
        MOV DS, AX

        MOV AX, num1          ;load AX and BX
        MOV BX, num2

        EXCH                  ;call macro

        MOV num1, AX          ;store AX and BX
        MOV num2, BX

        MOV AX, 4C00h         ;terminate
        INT 21h

code    ENDS
        END start
```

As can be seen from Program 6.8, the EXCH macro is defined in the beginning and then the data segme defines two numbers num1 and num2. Within the code segment, after initialization of the d segment, the numbers num1 and num2 are copied into the AX and BX registers, respectively. Then macro is called using its name EXCH which exchanges the contents of the AX and BX registers. Fin

```
next:       SHR BX, 1          ;shift
            JNC skip           ;if 0, don't count
            INC DL             ;increment counter
skip:       LOOP next          ;goto next
            MOV AL, DL         ;return count in AL

            POP DX             ;pop registers
            POP CX             ;and flags
            POP BX
            POPF
            RET                ;return
bin_cnt     ENDP

code_proc   ENDS
            END
```

As can be seen from Program 6.7, the variable num defined in the *main.asm* module is declared public to allow its access in the *proc.asm* module. Similarly, the `bin_cnt` procedure, defined in the *proc.asm* module, is declared public to call it from the *main.asm* module. The variable num and the `bin_cnt` procedure are declared external in *proc.asm* and *main.asm* modules, respectively. These external declarations inform the assembler that they are defined in other modules in the segments having the same names under which the `EXTRN` declarations are made. The type specifier `PUBLIC` in the segment starting after directive `SEGMENT` indicates that the segments with the same names are combined together.

To generate the executable program *main.exe*, first assemble both of them independently by giving the following commands:

```
C:\MASM\masm main;
C:\MASM\masm proc;
```

The above commands generate *main.obj* and *proc.obj* files, respectively. Link them together using the following command:

```
C:\MASM\link main.obj proc.obj
```

Accept the executable file name *main.exe* in response to the above command. The *main.exe* generated this way is an executable program which we wanted. Figure 7 shows the machine code of Program 6.7 using debug. Observe that the mainline program and the procedure are located in different code segments and the CALL instruction in the mainline program is coded as the direct intersegment CALL instruction.

```
        MOV AX, n1
        ADD AX, n2
        MOV ans, AX

        MOV AX, 4C00h
        INT 21h

code    ENDS
        END start
```

As can be seen from Program 6.11, the statement

```
small_gr GROUP data, code
```

informs the assembler that the segments with name `data` and `code` are combined in a logical group segment `small_gr`. Observe that the ASSUME statement binds both CS and DS to `small_gr`. If you observe the above program after assembling and linking using debug, you will find that both CS and DS contain the same base address (combined to single segment). The code segment uses the type specifier WORD in the first line, indicating that the segment should be built from the next even address. As can be seen from the data segment, it defines three words occupying space from offset 0000h to 0005h in a single segment. This means that the next available offset is 0006h from where the code segment is built. It is shown in Figure 9.

```
C:\MASM>debug p6_11.exe
-u
0B49:0006 B8490B      MOV     AX,0B49
0B49:0009 8ED8        MOV     DS,AX
0B49:000B A10000      MOV     AX,[0000]
0B49:000E 03060200    ADD     AX,[0002]
0B49:0012 A30400      MOV     [0004],AX
0B49:0015 B8004C      MOV     AX,4C00
0B49:0018 CD21        INT     21
-g cs:0015

AX=576A BX=0000 CX=001A DX=0000 SP=0000 BP=0000 SI=0000 DI=00
DS=0B49 ES=0B39 SS=0B49 CS=0B49 IP=0015 NV UP EI PL NZ NA PE
0B49:0015 B8004C      MOV     AX,4C00
-e ds:0000                              6A.   57.
0B49:0000  34.   12.   36.   45.
-
```

Figure 9 Machine code of Program 6.11.

Assume that Prog... the data and code in the same ... with value 100h. Following command gener...

```
C:\MASM>masm add;
```

To convert *add.obj* file to *add.exe*, the following command is used:

```
C:\MASM>masm add;
```

The above command generates the *add.exe* file which is not a compact binary file. As we have taken care of all the needs of the *.com* file while writing the *.asm* file, we can convert *add.exe* to *add.com* (compact binary file) using the following command:

```
C:\MASM>exe2bin add.exe add.com
```

The *add.com* file can then be executed like any DOS command. The major difference between the *.com* and *.exe* files is that the *.com* file is much faster than the *.exe* file, but it has limitation of size of 64 KB.

GROUP Directive

The GROUP directive is used to combine all the logical segments into a logical group segment. The linker links all the segments in a logical group segment into a single 64 KB physical segment having the same segment base. The following program demonstrates the use of the GROUP directive.

Program 6.11

Write a program to add two 16-bit numbers. Use the GROUP directive to combine the data and code segments.

Solution

The program is

```
small_gr GROUP data, code

data     SEGMENT
         n1  DW 1234h
         n2  DW 4536h
         ans DW ?
data     ENDS

code     SEGMENT WORD
         ASSUME cs:small_gr, ds:small_gr

start:   MOV AX, small_gr
         MOV DS, AX
```

ORG Directive

The ORG directive is another alignment directive and is known as originate. It is used to set the location counter to the value specified in the directive. Consider the following example:

```
ORG   $+5
```

The above directive sets the location counter to the current value of the location counter plus 5. This means that it reserves five memory bytes for future use.

The DOS supports two types of executable files: *.exe* and *.com*. The *.com* files are compact files and pack all the logical segments in the same 64 KB physical segment. The structure of the *.com* file requires the first instruction to start from the offset 100h (256 decimal) in its single segment as it uses the first 256 bytes (0 to 255) as header to store the important details. We can use the following directive to initialize the location counter to 100h:

```
ORG 100h
```

Let us see the use of the ORG directive to generate the DOS compact, that is, *.com* file.

Program 6.10

Write a program to add two 16-bit numbers. Use single segment and generate the compact binary file.

Solution

The program is

```
code SEGMENT
       ASSUME cs:code

       ORG 100h                ;set location counter
start: JMP skip                ;jump to first instruction

       n1 DW 1234.             ;store data
       n2 DW 4536h
       ans DW ?

skip:  MOV AX, n1              ;get n1 to AX
       ADD AX, n2              ;add AX to n2
       MOV ans, AX             ;store ans

       MOV AX, 4C00h           ;terminate
       INT 21h

code   ENDS
       END start
```