Chapter 6

8088/8086 Microprocessor Programming 2
Introduction

6.1 Flag-Control Instructions—✓
6.2 Compare Instruction—✓
6.3 Jump Instructions—✓
6.4 Subroutines and Subroutine-Handling Instructions —✓
6.5 The Loop and Loop-Handling Instructions —✓
6.6 Strings and Sting-Handling Instructions
6.1 Flag Control Instructions—Loading, Storing, and Modifying Flags

- Variety of flag control instructions provide support for loading, saving, and modifying content of the flags register
  - LAHF/SAHF → Load/store control flags
  - CLC/STC/CMC → Modify carry flag
  - CLI/STI → Modify interrupt flag

- Modifying the carry flag—CLC/STC/CMC
  - Used to initialize the carry flag
  - Clear carry flag
    CLC
    \( 0 \rightarrow (CF) \)
  - Set carry flag
    STC
    \( 1 \rightarrow (CF) \)
  - Complement carry flag
    CMC
    \( (CF^*) \rightarrow (CF) \) * stands for overbar (NOT)

- Modifying the interrupt flag—CLI/STI
  - Used to turn on/off external hardware interrupts
  - Clear interrupt flag
    CLC
    \( 0 \rightarrow (CF) \) Disable interrupts
  - Set interrupt flag
    STC
6.1 Flag Control Instructions—Debug Example

- Debug flag notation
  - CF → CY = 1, NC = 0
- Example—Execution of carry flag modification instructions
  CY=1 → initial state
  CLC ; Clear carry flag
  STC ; Set carry flag
  CMC ; Complement carry flag
6.1 Flag Control Instructions - Loading and Storing the Flags Register

- Format of the flags in the AH register
  - All loads and stores of flags take place through the AH register
    - B0 = CF
    - B2 = PF
    - B4 = AF
    - B6 = ZF
    - B7 = SF
- Load the AH register with the content of the flags registers
  - LAHF
    (Flags) → (AH)
    Flags unchanged
- Store the content of AH into the flags register
  - SAHF
    (AH) → (Flags)
    SF,ZF,AF,PF,CF → updated
- Application—saving a copy of the flags in memory and initializing with new values from memory
  - LAHF ; Load of flags into AH
  - MOV [MEM1],AH ; Save old flags at address MEM1
  - MOV AH,[MEM2] ; Read new flags from MEM2 into AH
  - SAHF ; Store new flags in flags register
6.1 Flag Control Instructions - Debug Example

**Example**—Execution of the flag save and initialization sequence

**Other flag notation:**
- **Flag** = 1/0
- **SF** = NG/PL
- **ZF** = ZR/NZ
- **AF** = AC/NA
- **PF** = PE/PO
6.2 Compare Instruction - Instruction Format and Operation

- Compare instruction
- Used to compare two values of data and update the state of the flags to reflect their relationship
- General format:
  ```
  CMP D,S
  ```
- Operation: Compares the content of the source to the destination; updates flags based on result
  
  $$(D) - (S) \rightarrow \text{Flags updated to reflect relationship}$$

  - Source and destination contents unchanged
  - Allowed operand variations:
    - Values in two registers
    - Values in a memory location and a register
    - Immediate source operand and a value in a register or memory
  - Allows SW to perform conditional control flow—typically testing of a flag by jump instruction
    - $ZF = 1 \rightarrow D = S = \text{Equal}$
    - $ZF = 0, CF = 1 \rightarrow D < S = \text{Unequal, less than}$
    - $ZF = 0, CF = 0 \rightarrow D > S = \text{Unequal, greater than}$
6.2 Compare Instruction - Compare Example

- Example:
  
  ```
  MOV AX,1234H ; Initialize AX
  MOV BX,ABCDH ; Initialize BX
  CMP AX,BX ; Compare AX-BX
  ```

- Initialization of data registers AX and BX with immediate data:
  
  - IMM16 → (AX) = 1234H
  - IMM16 → (BX) = ABCDH

- Compare computation performed as:
  
  
  \[
  (AX) = 0001001000110100_2 \\
  (BX) = 1010101111001101_2 \\
  (AX) - (BX) = 0001001000110100_2 - 1010101111001101_2
  \]

  - ZF = 0 = NZ
  - SF = 0 = PL → treats operands as signed numbers
  - CF = 1 = CY
  - AF = 1 = AC
  - OF = 0 = NV
  - PF = 0 = PO

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ZF</th>
<th>SF</th>
<th>CF</th>
<th>AF</th>
<th>OF</th>
<th>PF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial state</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOV AX,1234H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOV BX,ABCDH</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CMP AX,BX</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
6.2 Compare Instruction-Listing and Debug Execution

```
TITLE EXAMPLE 6.6
PAGE 132

STACK_SEG SEGMENT STACK 'STACK'
    DB 64 DUP(?)

STACK_SEG ENDS

CODE_SEG SEGMENT 'CODE'

ASSUME CS:CODE_SEG, SS:STACK_SEG

CODE_SEG ENDS

;To return to DEBUG program put return address on the stack

CODEB_SEG ENDS

Segments and groups:

Name     Size  align  combine  class

CODE_SEG  000E  PARA   NONE   'CODE'
STACK_SEG 0040  PARA   STACK  'STACK'

Symbols:

Name     Type    Value  Attr

EX66     F PROC  0000  CODE_SEG

Warning Errors  Severe Errors
9          0

C:\DOS>DEBUG A:EX66.EXE
-U 0, D
0F50:0000 1E  PUSH DS
0F50:0001 B80000 MOV AX, 0000
0F50:0004 50  PUSH AX
0F50:0005 B83412 MOV AX, 1234
0F50:0008 B8CDAB MOV BX, 1234
0F50:000B 3B3C CMP AX, BX
0F50:000D CB  RETF

;Following code implements Example 6.6

1E
0001 B8 0000
0004 50

;To return to DEBUG program put return address on the stack

0005 B8 1234
0008 B8 ABCD
000B 3B C3

;Return to DEBUG program

00DD CB
000E

C:\DOS>

Program terminated normally
-Q

The 8088 and 8086 Microprocessors, Triebel and Singh 9
6.3 Jump Instructions - Unconditional and Conditional Jump Control Flow

Jump operation alters the execution path of the instructions in the program—flow control

- **Unconditional Jump**
  - Always takes place
  - No status requirements are imposed
  - Example
    - JMP AA instructions in Part I executed
    - Control passed to next instruction identified by AA in Part III
    - Instructions in Part II skipped

- **Conditional jump**
  - May or may not take place
  - Status conditions must be satisfied
  - Example
    - Jcc AA instruction in Part I executed
    - Conditional relationship specified by cc is evaluated
    - If conditions met, jump takes place and control is passed to next instruction identified by AA in Part III
    - Otherwise, execution continues sequentially with first instruction in Part II
  - Condition cc specifies a relationship of status flags such as CF, PF, ZF, etc.
6.3 Jump Instructions - Unconditional Jump Instruction

Unconditional jump instruction
- Implements the unconditional jump operation needed by:
  - Branch program control flow structures
  - Loop program control flow structures

General format:
JMP Operand

Types of unconditional jumps
- Intrasegment—branch to address is located in the current code segment
  - Only IP changes value
  - short-label
    - 8-bit signed displacement coded into the instruction
    - Immediate addressing
    - Range equal –126 to +129
    - New address computed as:
      (Current IP) + short-label \rightarrow (IP)
      
      Jump to address = (Current CS) + (New IP)
  - near-label
    - 16-bit signed displacement coded in the instruction
    - Example
      JMP 1234H
6.3 Jump Instructions - regptr16

Unconditional Jump Example

- `regptr16`
  - 16-bit value of IP specified as the content of a register
  - Register addressing
  - Operation:
    
    
    \[(BX) \rightarrow (IP)\]

    
    Jump to address = (Current CS(0)) + (New IP)

- **Example**

  `1342:0100 JMP BX`

  Prior to execution

  \[(IP) = 0100H\]

  \[(BX) = 0010H\]

  After execution

  \[(IP) = 0010H\]

  Address of next instruction

  \[(CS:IP) = 1342:0010\]
6.3 Jump Instructions- memptr16

Unconditional Jump Example

- **memptr16**
  - 16-bit value of IP specified as the content of a storage location in memory
  - Register indirect addressing

- **Example**
  - 1342:0100 JMP [BX]

Prior to execution
- 
  - (IP) = 0100H
  - (DS) = 1342H
  - (BX) = 1000H
  - (DS:BX) = (1342H:1000H) = 0200H

After execution
- 
  - (IP) = 0200H

Next instruction
- 
  - (CS:IP) = 1342:0200H
6.3 Jump Instructions- Intersegment
Unconditional Jump Operation

- Intersegment—branch to address is located in another code segment
  - Both CS and IP change values
  - far-label
    - 32-bit immediate operand coded into the instruction
    - New address computed as:
      - 1st 16 bits $\rightarrow$ (IP)
      - 2nd 16 bits $\rightarrow$ (CS)
    Jump to address = (New CS):(New IP)
- memptr32
  - 32-bit value specified in memory
  - Memory indirect addressing
  - Example
    - JMP DWORD PTR [DI]
  - Operation:
    - (DS:DI) $\rightarrow$ new IP
    - (DS:DI +2) $\rightarrow$ new CS
    Jump to address = (New CS):(New IP)
6.3 Jump Instructions - Conditional Jump Instruction

- Condition jump instruction
- Implements the conditional jump operation
- General format:
  
  \[ \text{Jcc} \text{ Operand} \]
  
  - \( \text{cc} = \) one of the supported conditional relationships
  - Supports the same operand types as unconditional jump

- Operation: Flags tested for conditions defined by \( \text{cc} \) and:

  If \( \text{cc} \) test True:
  
  - Jump is taken
  - Execution resumes at jump to target address

  If \( \text{cc} \) test False:
  
  - Jump is not taken
  - Execution continues with the next sequential instruction

- Examples of conditional tests:
  
  \[ \text{JC} = \text{jump on carry} \rightarrow \text{CF} = 1 \]
  
  \[ \text{JPE/JP} = \text{jump on parity even} \rightarrow \text{PF} = 1 \]
  
  \[ \text{JE/JZ} = \text{jump on equal} \rightarrow \text{ZF} = 1 \]
6.3 Jump Instructions - Branch Program Structures

- Example—IF-THEN-ELSE comparing values
- One of the most widely used flow control program structure
- Implemented with CMP, JE, and JMP instructions

**Operation**
- AX compared to BX to update flags
- JE tests for 
  \[ ZF = 1 \]
- If \( (AX) \neq (BX) \); \( ZF = 0 \) \( \rightarrow \) THEN path—next sequential instruction is executed
- If \( (AX) = (BX) \); \( ZF = 1 \) \( \rightarrow \) ELSE path—instruction pointed to by EQUAL executes
- JMP instruction used in THEN path to bypass the ELSE path when
6.3 Jump Instructions - Branch Program Structures

- Example—IF-THEN-ELSE using a register bit test
- Conditional test is made with JNZ instruction and branch taken if
  \[ ZF = 0 \]
- Generation of test condition
  \[ (AL) = \text{xxxxxxx} \ \text{AND} \ 00000100 \]
  \[ = 00000x00 \]
  if bit 2 = 1 ZF = 0 (not zero)
  if bit 2 = 0 ZF = 1
  Therefore, jump to BIT2_ONE only takes place if bit 2 of AL equals 1
- Same operation can be performed by shifting bit 2 to the CF and then testing with JC
  CF = 1

```
AND AL, 04H
JNZ BIT2_ONE
    ---  ---  ; Next instruction if B2 of AL = 0
    ---
.
JMP END

BIT2_ONE:  ---  ---  ; Next instruction if B2 of AL = 1
    ---

END:      ---  ---
```
6.3 Jump Instructions - Program Applying Branching

C:\DOCS>DEBUG A: EX610.EXE
-U 0 15
OD03:0000 1E PUSH DS
OD03:0001 B80000 MOV AX, 0000
OD03:0004 5C60 PUSH AX
OD03:0005 3BC3 CMP AX, BX
OD03:0007 7207 JB 0010
OD03:0009 6ED0 MOV DX, AX
OD03:000C 280D SUB DX, SX
OD03:000D EB05 JNP 0014
OD03:000F 5C0F NOP
OD03:0010 8B33 MOV DX, BX
OD03:0012 280D SUB DX, AX
OD03:0014 90 NOP
OD03:0015 CB RETF

AX=0200 BX=0000 CX=0016 DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0C0D ES=0C0D SS=0C08 CS=0D01 IP=0007 NV UP EI PL NZ NA PO NC
OD03:0005 3BC3 CMP AX, BX
-R AX AX 0000
:8 -R BX BX 0000
:3 -T

AX=0200 BX=0002 CX=0016 DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0C0D ES=0C0D SS=0C08 CS=0D03 IP=0007 NV UP EI PL NZ NA PO NC
OD03:0007 7207 JB 0010
-G 14

Program terminated normally
-R AX AX 0000
:8 -R BX BX 0002
:3 -T

AX=0200 BX=0006 CX=0016 DX=0004 SP=0038 BP=0000 SI=0000 DI=0000
DS=0C0D ES=0C0D SS=0C08 CS=0D03 IP=0007 NV UP EI PL NZ AC PE CY
OD03:0007 7207 JB 0010
-G 14

Program terminated normally
-Q
C:\DOCS>
6.3 Jump Instructions- Loop Program Structures

- Example—Repeat-Until program structure
  - Allows a part of a program to be conditionally repeated over and over
  - Employs post test—conditional test at end of sequence; always performs one iteration
- Important parameters
  - Initial count → count register
  - Terminal count → zero or other value
- Program flow of control:
  - Initialize count
    MOV CL,COUNT
  - Perform body of loop operation
    AGAIN: --- --- first of multiple instructions
  - Decrement count
    DEC CL
  - Conditional test for completion
    JNZ AGAIN

```
AGAIN:          ; Set loop repeat count
    MOV CL,COUNT
    ; 1st instruction of loop
    --- ---
    ; 2nd instruction of loop
    ---
    ; ...;p\^{\text{th}}\text{ instruction of loop}
    ---
    DEC CL ; Decrement repeat count by 1
    JNZ AGAIN ; Repeat from AGAIN if (CL) \neq 00H or (ZF) = 0
    --- --- ; First instruction executed after the loop is complete, (CL) = 00H, (ZF) = 1

(a)
```
6.3 Jump Instructions - Loop Program Structures

- Example — While-Do program structure
  - Allows a part of a program to be conditionally repeated over and over
  - Employs pre-test — at entry of loop; may perform no iterations
- Important parameters
  - Initial count \(\rightarrow\) count register
  - Terminal count \(\rightarrow\) zero or other value
- Program flow/control:
  - Initialize count
    \[ \text{MOV CL, COUNT} \]
  - Pre-test
    \[ \text{AGAIN: JZ NEXT} \]
  - Perform body of loop operation
    \[ \text{--- --- first of multiple instructions} \]
  - Decrement count
    \[ \text{DEC CL} \]
  - Unconditional return to start of loop
    \[ \text{JMP AGAIN} \]
6.3 Jump Instructions- Block Move Program

Start

Establish the data segment, source block, and destination block

Initialization

Set up a counter for the points to be moved

Move the next source point to the accumulator

Data movement

Move the accumulator to the next destination point

Update counter, source pointer, and destination pointer

Update

No All points moved?

Test

Yes

Stop

MOV AX, DATASEGADDR
MOV DS, AX
MOV SI, BLK1ADDR
MOV DI, BLK2ADDR
MOV CX, N

NXTPT:
MOV AH, [SI]
MOV [DI], AH
INC SI
INC DI
DEC CX
JNZ NXTPT
HLT

The 8088 and 8086 Microprocessors, Triebel and Singh 21
6.4 Subroutines and Subroutine-Handling Instructions

- **Subroutine**—special segment of program that can be called for execution from any point in a program
  - Program structure that implements HLL “functions” and “procedures”
  - Written to perform an operation (function/procedure) that must be performed at various points in a program
  - Written as a subroutine and only included once in the program
  - Example:
    - Instruction in Main part of program calls “Subroutine A”
    - Program flow of control transferred to first instruction of Subroutine A
    - Instructions of Subroutine A execute sequentially
    - Return initiated by last instruction of Subroutine A
    - Same sequence repeated when the subroutine is called again later in the program

- **Instructions**
  - Call instruction—initiates the subroutine from the main part of program
  - Return instruction—initiates return of control to the main program at completion of the subroutine
  - Push and pop instructions used to save register content and pass parameters
6.4 Subroutines and Subroutine-Handling Instructions- Call Instruction

- Call Instruction
  - Implements two types of calls
    - Intrasegment call
    - Intersegment call
  - Intrasegment call—starting address of subroutine is located in the current code segment
    - Only IP changes value
    - near-proc
      - 16-bit offset coded in the instruction
      - Example
        CALL 1234H
      - Operation:
        1. IP of next instruction saved on top of stack
        2. SP is decremented by 2
        3. New value from call instruction is loaded into IP
        4. Instruction fetch restarts with first instruction of subroutine
          Current CS:New IP
6.4 Subroutines and Subroutine-Handling Instructions - Intrasegment Call Operation (Continued)

- **regptr16**
  - 16-bit value of IP specified as the content of a register
  - Register addressing
  - Example:
    
    ```
    CALL BX
    ```
  - Operation:
    - Same as near-proc except
      - (BX) → New IP

- **memptr16**
  - 16-bit value of IP specified as the content of a storage location in memory
  - Memory addressing modes—register addressing
  - Example
    
    ```
    CALL [BX]
    ```
  - Same as near-proc except
    - (DS:BX) → New IP
6.4 Subroutines and Subroutine-Handling Instructions - Intersegment Call Operation

- **Intersegment**—start address of the subroutine points to another code segment
  - Both CS and IP change values
  - **far-proc**
    - 32-bit immediate operand coded into the instruction
    - New address computed as:
      - 1\textsuperscript{st} 16 bits $\rightarrow$ New IP
      - 2\textsuperscript{nd} 16 bits $\rightarrow$ New CS
    
    Subroutine starts at $=$ New CS:New IP

- **memptr32**
  - 32-bit value specified in memory
  - Memory addressing modes—register indirect addressing
  - Example
    
    CALL DWORD PTR [DI]
  
  - Operation:
    
    (DS:DI) $\rightarrow$ New IP
    (DS:DI +2) $\rightarrow$ New CS
    
    Starting address of subroutine $=$ New CS:New IP
6.4 Subroutines and Subroutine-Handling Instructions - Return Instruction

- Return instruction
  - Every subroutine must end with a return instruction
  - Initiates return of execution to the instruction in the main program following that which called the subroutine
- Example:

```
RET
```

- Causes the value of IP (inrasegment return) or both IP and CS (intersegment return) to be popped from the stack and put back into the IP and CS registers
- Increments SP by 2/4
6.4 Subroutines and Subroutine-Handling Instructions - Example of a Subroutine Call

```assembly
TITLE EXAMPLE 6.11
PAGE 132

STACK_SEG SEGMENT STACK 'STACK'
DB 64 DUP(?)
STACK_SEG ENDS

CODE_SEG SEGMENT 'CODE'
ASSUME CS:CODE_SEG, SS:STACK_SEG

;To return to DEBUG program put return address on the stack

0000 1E
0001 B0 0000
0002 50
;Following code implements Example 6.11

0003 89 0009 R
0004 CB
0005 CALL SUM
0006 CB
0007 PROC NEAR
0008 MOV DX, AX
0009 ADD DX, BX
000A  ; (DX) = (AX) + (BX)
000B  RET
000C  MOV DX, AX
000D  ADD DX, BX
000E  MOV DX, AX
000F  ADD DX, BX
0010  MOV AX, 0
0011  PUSH AX
0012  MOV AX, 0
0013  PUSH DS
0014  MOV AX, 0
0015  ADD DX, BX
0016  MOV AX, 0
0017  CALL 0009

C:\DOS>DEBUG A:EX611.EXE
-U 0 D
0003:0000 1E  PUSH DS
0003:0001 B0 0000  MOV AX, 0000
0003:0004 50  PUSH AX
0003:0005 B0 0100  CALL 0009
0003:0008 CB  RETF
0003:0009 B0 00  MOV DX, AX
0003:000B 03 03  ADD DX, BX
0003:000D C3  RET

-AX AX 0000
+2
-R BX
+BX 0000
+4
-T

AX=0000 BX=0000 CK=000E DX=0000 SP=003C BP=0000 SI=0000 DI=0000
DS=0F41 ES=0F41 SS=0F52 CS=0D03 IP=0005 NV UP EI PL NZ NA PO NC
0003:0005 B0 0100  CALL 0009

-AX AX 0000
+2
-R BX
+BX 0000
+4
-T

AX=0002 BX=0004 CK=000E DX=0000 SP=003A BP=0000 SI=0000 DI=0000
DS=0F41 ES=0F41 SS=0F52 CS=0D03 IP=0009 NV UP EI PL NZ NA PO NC
0003:0008 03 03 ADD DX, BX

-C:

The 8088 and 8086 Microprocessors, Triebel and Singh 27
### 6.4 Subroutines and Subroutine-Handling Instructions

#### Structure of a Subroutine

- **Elements of a subroutine**
  - Save of information to stack—**PUSH**
  - Main body of subroutine—Multiple instructions
  - Restore of information from stack—**POP**
  - Return to main program—**RET**

- **Save of information**
  - Must save content of registers/memory locations to be used or other program parameters (FLAGS)
  - **PUSH, PUSHF**

- **Main body**
  - Retrieve input parameters passed from main program via stack—stack pointer indirect address
  - Performs the algorithm/function/operation required of the subroutine
  - Prepare output parameters/results for return to main body via stack—stack pointer indirect addressing

- **Restore information**
  - Register/memory location contents saved on stack at entry of subroutine must be restored before return to main program—**POP, POPF**
6.4 Subroutines and Subroutine-Handling Instructions- Push and Pop Instruction

- **Push instruction**
  - **General format:**
    
    ```
PUSH S
    ```

    - Saves a value on the stack—content of:
      - Register/segment register
      - Memory

  - **Example:**
    
    ```
PUSH AX
    ```

    
    ```
    (AH) → ((SP)-1)
    (AL) → ((SP)-2)
    (SP)-2 → (SP) = New top of stack
    ```

- **Pop instruction**
  - **General format:**
    
    ```
POP D
    ```

    - Restores a value on the stack—content to: register, segment register, memory

  - **Example:**
    
    ```
POP AX
    ```

    
    ```
    ((SP)) → AL
    ((SP)+1) → AH
    ((SP)+2) → SP = Old top of stack
    ```
6.4 Subroutines and Subroutine-Handling Instructions - Subroutine Call Involving PUSH and POP

```
TITLE EXAMPLE 6.13
PAGE 132

0000 0040 SEGMENT STACK 'STACK'
      DB 64 DUP(?)

0040 SEGMENT STACK_SEG ENDS

0000 0000 SEGMENT DATA_SEG
      TOTAL DN 1234H
      ENDS

0002 SEGMENT DATA_SEG ENDS

0000 SEGMENT CODE_SEG
      PROC FAR
      ASSUME CS:CODE_SEG, SS:STACK_SEG, DS:DATA_SEG

      ;To return to DEBUG program put return address on the stack
      PUSH DS
      MOV AX, 0
      PUSH AX

      ;Setup the data segment
      MOV AX, DATA_SEG
      MOV DS, AX

      ;Following code implements Example 6.13
      CALL SQUARE

      MOV BL, 12H
      ;BL contents = the number to be squared
      CALL SQUARE

      RET ;Return to DEBUG program

0010 0018 SEGMENT SQUARE
      PROC NEAR
      ;Save the register
      PUSH AX

      MOV AL, BL
      IMUL HL
      MOV BX, AX
      POP AX
      RET

0019 CODE_SEG ENDS

END EX613
```

Segments and Groups:

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Align</th>
<th>Combine</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE_SEG</td>
<td>0019</td>
<td>PARA</td>
<td>NONE</td>
<td>&quot;CODE&quot;</td>
</tr>
<tr>
<td>DATA_SEG</td>
<td>0002</td>
<td>PARA</td>
<td>NONE</td>
<td>&quot;STACK&quot;</td>
</tr>
<tr>
<td>STACK_SEG</td>
<td>0040</td>
<td>PARA</td>
<td>STACK</td>
<td>&quot;STACK&quot;</td>
</tr>
</tbody>
</table>

Symbols:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value Attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX613</td>
<td>F PROC</td>
<td>0000</td>
</tr>
<tr>
<td>SQUARE</td>
<td>N PROC</td>
<td>0010 CODE_SEG</td>
</tr>
<tr>
<td>TOTAL</td>
<td>L WORD</td>
<td>0000 DATA_SEG</td>
</tr>
</tbody>
</table>

| CPU     | TEXT   | 0101h     |
|-------------------------------|
| @FILENAME | TEXT | EX613     |
| @VERSION | TEXT | 613       |

48016 + 440523 Bytes symbol space free

0 Warning Errors
0 Severe Errors
6.4 Subroutines and Subroutine-Handling Instructions - Subroutine Call involving PUSH and POP (continued)

C:\DOS>DEBUG A:\EX613.EXE
-U 0 18
0010:0000 5E  push ds
0010:0001 BB 0000  mov ax,0000
0010:0004 50  push ax
0010:0005 BB 000D  mov ax,00ED
0010:0008 86D8  mov dx,ax
0010:000A 3112  mov bl,12
0010:000C EB 0100  call 0010
0010:000F CB  retf
0010:0010 50  push ax
0010:0011 B6C3  mov al,bl
0010:0013 F68B  imul bl
0010:0015 88D8  mov bx,ax
0010:0017 58  pop ax
0010:0018 C3  ret

AX=CDEB BX=0012 CX=0069 DX=0000 SP=003C
DS=CDEB ES=00D7 SS=CDE7 CS=CDEC IF=0000
BP=0000 SI=0000 DI=0000
0010:001A 0F 00

AX=CDEB BX=0012 CX=0069 DX=0000 SP=003A
DS=CDEB ES=00D7 SS=CDE7 CS=CDEC IF=0010
BP=0000 SI=0000 DI=0000
0010:001C 0F 00

AX=CDEB BX=0012 CX=0069 DX=0000 SP=003B
DS=CDEB ES=00D7 SS=CDE7 CS=CDEC IF=0011
BP=0000 SI=0000 DI=0000
0010:001E EB 0D

AX=0144 BX=0144 CX=0069 DX=0000 SP=003E
DS=00D7 SS=00D7 CS=0DEC IF=0017
BP=0000 SI=0000 DI=0000
0010:0020 POP AX

AX=CDEB BX=0144 CX=0069 DX=0000 SP=003A
DS=CDEB ES=00D7 SS=CDE7 CS=CDEC IF=0018
BP=0000 SI=0000 DI=0000
0010:0022 RET

AX=CDEB BX=0144 CX=0069 DX=0000 SP=003C
DS=CDEB ES=00D7 SS=CDE7 CS=CDEC IF=000F
BP=0000 SI=0000 DI=0000
0010:0024 RETF

Program terminated normally
-Q
6.4 Subroutines and Subroutine-Handling Instructions: Push Flags Instruction

- **Push flags instruction**
  - **General formats:**
    - **PUSHF**
      - Saves flags onto the stack
    - **Operation**
      - (FLAGS) → ((SP))
      - (SP)-2 → (SP) = New top of stack

- **Pop flags instruction**
  - **General formats:**
    - **POPF**
      - Restores flags from the stack
    - **Operation**
      - ((SP)) → FLAGS
      - (SP)+2 → (SP) = Old top of stack

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSHF</td>
<td>Push flags onto stack</td>
<td>((SP)) ← (Flags)</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SP) ← (SP)-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Flags) ← ((SP))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(SP) ← (SP)+2</td>
<td>OF, DF, IF, TF, SF, ZF, AF, PF, CF</td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags from stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 6.5 The Loop and Loop-Handling Instructions

#### Loop Instructions
- **Loop**—segment of program that is repeatedly executed
  - Can be implemented with compare, conditional jump, and decrement instructions
- **Loop instructions**
  - Special instructions that efficiently perform basic loop operations
  - Replace the multiple instructions with a single instruction
- **LOOP**—loop while not zero
  - CX ≠ 0 — repeat while count not zero
- **LOOPE/LOOPZ**—loop while equal
  - CX ≠ 0 — repeat while count not zero, and
  - ZF = 1—result of prior instruction was equal
- **LOOPNE/LOOPNZ**—loop while not equal
  - CX ≠ 0 — repeat while count not zero, and
  - ZF = 0—result from prior instruction was not equal

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>Loop</td>
<td>LOOP Short-label</td>
<td>(CX) ← (CX) - 1, Jump is initiated to location defined by short-label if (CX) ≠ 0, otherwise, execute next sequential instruction</td>
</tr>
<tr>
<td>LOOPE/LOOPZ</td>
<td>Loop while equal/loop while zero</td>
<td>LOOPE/LOOPZ Short-label</td>
<td>(CX) ← (CX) - 1, Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 1, otherwise, execute next sequential instruction</td>
</tr>
<tr>
<td>LOOPNE/LOOPNZ</td>
<td>Loop while not equal/loop while not zero</td>
<td>LOOPNE/LOOPNZ Short-label</td>
<td>(CX) ← (CX) - 1, Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 0, otherwise, execute next sequential instruction</td>
</tr>
</tbody>
</table>
6.5 The Loop and Loop-Handling Instructions—
Loop Program Structure and Operation

- Structure of a loop
  - Initialization of the count in CX
  - Body—instruction sequence that is to be repeated; short label identifying beginning
  - Loop instruction—determines if loop is complete or if the body is to repeat

Example
1. Initialize data segment, source and destination block pointers, and loop count
2. Body of program is executed—source element read, written to destination, and then both pointers incremented by 1
3. Loop test
   a. Contents of CX decremented by 1
   b. Contents of CX check for zero
   c. If CX = 0, loop is complete and next sequential instruction (HLT) is executed
   d. If CX ≠ 0, loop of code is repeated by returning control to the instruction corresponding to the Short-Label (NXTPT:) operand

```
MOV CX,COUNT

[Loop Body]

LOOP NEXT

NEXT:

MOV AX,DATASEGADDR
MOV DS,AX
MOV SI,BLK1ADDR
MOV DI,BLK2ADDR
MOV CX,N
MOV AH,[SI]
MOV [DI],AH
INC SI
INC DI
LOOP NXTPT

NXTPT:

MOV HLT
```
6.5 The Loop and Loop-Handling Instructions—Loop Example—Loop Count Operation

```
TITLE EXAMPLE 6.14
PAGE ,132

0000        STACK_SEG SEGMENT STACK 'STACK'
0004        DB 64 DUP(?)

0040        STACK_SEG ENDS

0000        CODE_SEG SEGMENT 'CODE'
0000        EX614 PROC FAR
0004        ASSUME CS:CODE_SEG, SS:STACK_SEG

/To return to DEBUG program put return address on the stack

0000 1E
0001 B6 0000
0004 50

;Following code implements Example 6.14

0005 B9 0005
0006 BA 0000
000B 90
000C 42
000D B2 FC

AGAIN: NOP
INC DX
LOOP AGAIN

000F CB
0010        EX614 ENDP

Segments and groups:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>align</th>
<th>combine</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE_SEG</td>
<td>0010</td>
<td>PARA</td>
<td>NONE</td>
<td>'CODE'</td>
</tr>
<tr>
<td>STACK_SEG</td>
<td>0040</td>
<td>PARA</td>
<td>STACK</td>
<td>'STACK'</td>
</tr>
</tbody>
</table>

Symbols:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Attr</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGAIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX614</td>
<td>L</td>
<td>NEAR</td>
<td></td>
</tr>
<tr>
<td>EX614</td>
<td>P</td>
<td>PROC</td>
<td></td>
</tr>
</tbody>
</table>

Warning  Severe  Errors  Errors
0         0

C:\DOS>DEBUG A:EX614.EXE
-U 0 0 F
0D03:0000 1E            PUSHD S
0D03:0001 B60000         MOV AX,0000
0D03:0004 50             PUSH AX
0D03:0005 B90500         MOV CX,0005
0D03:0008 BA0000         MOV DX,0000
0D03:000B 90             NOP
0D03:000C 42             INC DX
0D03:000D B2FC           LOOP 000B
0D03:000F CB             RETF
-G B

AX=0000 BX=0000 CX=0005 DX=0000 SP=002C BP=0000 SI=0000 DI=0000
DS=00D7 ES=0DD7 SS=0DE8 CS=0D03 IP=000B NV UP EI PL NZ NA PO NC
0D03:000B 90             NOP
-G D

AX=0000 BX=0000 CX=0005 DX=0001 SP=003C BP=0000 SI=0000 DI=0000
DS=00D7 ES=0DD7 SS=0DE8 CS=0D03 IP=000D NV UP EI PL NZ NA PO NC
0D03:000D B2FC           LOOP 000B
-T

AX=0000 BX=0000 CX=0004 DX=0001 SP=003C BP=0000 SI=0000 DI=0000
DS=00D7 ES=0DD7 SS=0DE8 CS=0D03 IP=000B NV UP EI PL NZ NA PO NC
0D03:000B 90             NOP
-G F

AX=0000 BX=0000 CX=0004 DX=0005 SP=003C BP=0000 SI=0000 DI=0000
DS=00D7 ES=0DD7 SS=0DE8 CS=0D03 IP=000F NV UP EI PL NZ NA PO NC
0D03:000F CB             RETF
-G

Program terminated normally
-Q

C:\DOS>
```
6.5 The Loop and Loop-Handling Instructions—Block Search Operation

TITLE EXAMPLE 6.16
PAGE .132

STACK_SEG    SEGMENT
    DB 64 DUP(?)

STACK_SEG    ENDS

CODE_SEG    SEGMENT 'CODE'
    PROC    FAR
    ASSUME    CS:CODE_SEG, SS:STACK_SEG

;To return to DEBUG program put return address on the stack
    PUSH    DS
    MOV    AX, 0
    PUSH    AX

;Following code implements Example 6.16
    B2 05
    MOV    DL, $H
    MOV    AX, $A000
    MOV    DS, AX
    MOV    SI, $0H
    MOV    CX, $0FH
    CMP    [SI], DL
    LOOPNE    AGAIN

    CB
    RET

    CB
    RET

CODE_SEG    ENDS

END EX616

Segments and groups:

Name     Size  align  combine  class
CODE_SEG  0018  PARA   NONE 'CODE'
STACK_SEG 0040  PARA   STACK 'STACK'

Symbols:

Name     Type     Value   Attr
AGAIN     L NEAR   0012 CODE_SEG
EX616     F PROC   0000 CODE_SEG   Length =0018

Warning    Severe  Errors  Errors
0        0

C:\DOS>DEBUG A:EX616.EXE
-D 0 17
0001 B0 0000
0004 50

C:\DOS>

The 8088 and 8086 Microprocessors, Triebel and Singh 36
6.6 Strings and String-Handling Instructions-
String Instructions

- **String**—series of bytes or words of data that reside at consecutive memory addresses
- **String instructions**
  - Special instructions that efficiently perform basic string operations
  - Replaces multiple instructions with a single instruction
- **Examples**
  - Move string
  - Compare string
  - Scan string
  - Load string
  - Store string
  - Repeated string

**Typical string operations**
- Move a string of data elements from one part of memory to another—block move
- Scan through a string of data elements in memory looking for a specific value
- Compare the elements of two strings of data elements in memory to determine if they are the same or different
- Initialize a group of consecutive storage locations in memory

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVS</td>
<td>Move string</td>
<td>MOVS/B/MOVSW</td>
<td>((ES)0 + (DI)) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2</td>
<td>None</td>
</tr>
<tr>
<td>CMPS</td>
<td>Compare string</td>
<td>CMPSB/CMPSW</td>
<td>Set flags as per ((DS)0 + (SI)) ← ((ES)0 + (DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2</td>
<td>CF, PF, AF, ZF, SF, OF</td>
</tr>
<tr>
<td>SCAS</td>
<td>Scan string</td>
<td>SCASB/SCASW</td>
<td>Set flags as per (AL or AX) ← ((ES)0 + (DI)) (DI) ← (DI) ± 1 or 2</td>
<td>CF, PF, AF, ZF, SF, OF</td>
</tr>
<tr>
<td>LODS</td>
<td>Load string</td>
<td>LODSB/LODSW</td>
<td>(AL or AX) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2</td>
<td>None</td>
</tr>
<tr>
<td>STOS</td>
<td>Store string</td>
<td>STOSB/STOSW</td>
<td>((ES)0 + (DI)) ← (AL or AX) ± 1 or 2 (DI) ← (DI) ± 1 or 2</td>
<td>None</td>
</tr>
</tbody>
</table>
6.6 Strings and String-Handling Instructions—Autoindexing

- Autoindexing—name given to the process of automatically incrementing or decrementing the source and destination addresses by the string instructions
- Direction (DF) control flag of the status register determines mode of operation
  - \(DF=0\) \(\rightarrow\) autoincrement
  - \(DF=1\) \(\rightarrow\) autodecrement
- Increment or decrement is by 1 or 2 depending on size data specified in the instruction
- Direction flag instructions permit the DF bit to be cleared or set as part of a string routine
  - CLD—clear direction flag
    - \(0 \rightarrow (DF) =\) autoincrement
  - STD—set direction flag
    - \(1 \rightarrow (DF) =\) autodecrement

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Format</th>
<th>Operation</th>
<th>Affected flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLD</td>
<td>Clear DF</td>
<td>CLD</td>
<td>((DF) \leftarrow 0)</td>
<td>DF</td>
</tr>
<tr>
<td>STD</td>
<td>Set DF</td>
<td>STD</td>
<td>((DF) \leftarrow 1)</td>
<td>DF</td>
</tr>
</tbody>
</table>

Moved from later
6.6 Strings and String-Handling Instructions—Move String Instruction

- Move string instruction
  - Used to move an element of data between a source and destination location in memory:
  - General format:
    - MOVSB—move string byte
    - MOVSW—move string word
  - Operation: Copies the content of the source to the destination; autoincrements/decrements both the source and destination addresses
    - \(((DS)0+(SI)) \rightarrow ((ES)0+(DI))\)
    - \((SI) \pm 1\) or 2 \(\rightarrow (SI)\)
    - \((DI) \pm 1\) or 2 \(\rightarrow (DI)\)
  - Direction flag determines increment/decrement
    - DF = 0 \(\rightarrow\) autoincrement
    - DF = 1 \(\rightarrow\) autodecrement
  - Application example—block move
    1. Initialize DS & ES to same value
    2. Load SI and DI with block starting addresses
    3. Load CX with the count of elements in the string
    4. Set DF for autoincrement
    5. Loop on string move to copy N elements
  - MOVSB and LOOP replaces multiple move and increment/decrement instructions

```
MOV  AX, DATASEGADDR
MOV  DS, AX
MOV  ES, AX
MOV  SI, BLK1ADDR
MOV  DI, BLK2ADDR
MOV  CX, N

CLD
MOVSB
LOOP NXTPT

NXTPT:  HLT
```
6.6 Strings and String-Handling Instructions—Compare/Scan String Instructions

- Compare string instruction
  - Used to compare the destination element of data in memory to the source element in memory and reflect the result of the comparison in the flags
  - General format:
    CMPSB,SW—compare string byte, word
  - Operation: Compares the content of the destination to the source; updates the flags; autoincrements/decrements both the source and destination addresses
    \[(DS)0+(SI)) - ((ES)0+(DI))\]
    update status flags
    \[(SI) \pm 1\text{ or } 2 \rightarrow (SI)\]
    \[(DI) \pm 1\text{ or } 2 \rightarrow (DI)\]

- Scan string instruction—SCAS
  - Same operation as CMPS except destination is compared to a value in the accumulator (A) register
    \[(AL,AX) - ((ES)0+(DI))\]
  - Application example—block scan
    1. Initialize DS & ES to same value
    2. Load AL with search value; DI with block starting address; and CX with the count of elements in the string; clear DF
    3. Loop on scan string until the first element equal to 05H is found

### Assembly Code

```asm
MOV AX,0
MOV DS,AX
MOV ES,AX
MOV AL,05
MOV DI,0A000H
MOV CX,0FH
CLD
AGAIN:
    SCASB
    LOOPNE AGAIN
```

The 8088 and 8086 Microprocessors, Triebel and Singh
6.6 Strings and String-Handling Instructions-
Load/Store String Instructions

- **Load string instruction**
  - Used to load a source element of data from memory into the accumulator register.
  - **General format:**
    
    ```
    LODSB, SW—load string byte, word
    ```
  - **Operation:** Loads the content of the source element in the accumulator; autoincrements/decrements the source addresses
    
    ```
    ((DS)0+(SI)) \rightarrow (AL or AX)
    ```
    
    update status flags
    
    ```
    (SI) \pm 1 or 2 \rightarrow (SI)
    ```

- **Store string instruction**—STOS
  - Same operation as LODS except value in accumulator is stored in destination is memory
    
    ```
    (AL, AX) \rightarrow ((ES)0+(DI))
    ```

- **Application example**—initializing a block of memory
  1. Initialize DS & ES to same value
  2. Load AL with initialization value; DI with block starting address, CX with the count of elements in the string; and clear DF
  3. Loop on store string until all element of the string are initialized to 05H

<table>
<thead>
<tr>
<th>MOV</th>
<th>AX,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>DS,AX</td>
</tr>
<tr>
<td>MOV</td>
<td>ES,AX</td>
</tr>
<tr>
<td>MOV</td>
<td>AL,05</td>
</tr>
<tr>
<td>MOV</td>
<td>DI,0A000H</td>
</tr>
<tr>
<td>MOV</td>
<td>CX,0FH</td>
</tr>
<tr>
<td>CLD</td>
<td></td>
</tr>
</tbody>
</table>

AGAIN: STOSB

LOOP AGAIN
6.6 Strings and String-Handling Instructions-
Repeat String Instructions

- Repeat string—in most applications the basic string operations are repeated
  - Requires addition of loop or compare & conditional jump instructions
  - Repeat prefix provided to make coding of repeated sting more efficient

- Repeat prefixes
  - **REP**
    - \( CX \neq 0 \) — repeat while not end of string
    - Used with: MOVS and STOS
  - **REPE/REPZ**
    - \( CX \neq 0 \) — repeat while not end of string, and
    - \( ZF = 1 \)—strings are equal
    - Used with: CMPS and SCAS
  - **REPNE/REPNZ**—Used with: CMPS and SCAS
    - \( CX \neq 0 \) — repeat while not end of string, and
    - \( ZF = 0 \)—strings are not equal
    - Used with: CMPS and SCAS

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Used with:</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP</td>
<td>MOVS, STOS</td>
<td>Repeat while not end of string</td>
</tr>
<tr>
<td>REPE/REPZ</td>
<td>CMPS, SCAS</td>
<td>Repeat while not end of string and strings are equal</td>
</tr>
<tr>
<td>REPNE/REPNZ</td>
<td>CMPS, SCAS</td>
<td>Repeat while not end of string and strings are not equal</td>
</tr>
</tbody>
</table>
6.6 Strings and String-Handling Instructions-
Repeat String Examples and Application

- General format:
  \[ \text{REPXXXX} \]
  Where: \( XXXX \) = one of string instructions

- Examples:
  - \( \text{REPMOV} \)
  - \( \text{REPESCAS} \)
  - \( \text{REPESCAS} \)

- Application example—initializing a block of memory
  1. Initialize DS & ES to same value
  2. Load AL with initialization value; DI with block starting address, and CX with the count of elements in the string
  3. Clear the direction flag for autoincrement mode
  4. Repeat store string until all elements of the string are initialized to \( 05H \)
6.6 Strings and String-Handling Instructions—
Example String Application

```
C:\DOS>DEBUG A:EX618.EXE
-U 0 18
0DE7:0000 1E  PUSH  DS
0DE7:0001  B80000  MOV  AX,0000
0DE7:0004  50  PUSH  AX
0DE7:0005  89E90D  MOV  AX,0DE9
0DE7:0008  88D8  MOV  DS,AX
0DE7:000A  8EC0  MOV  ES,AX
0DE7:000C  0F  CLD
0DE7:000D  B92000  MOV  CX,0020
0DE7:0010  BE0000  MOV  SI,0000
0DE7:0013  BF2000  MOV  DI,0020
0DE7:0016  F3  REPZ
0DE7:0017  A4  MOVSB
0DE7:0018  CB  RETP
-G 16

AX=0DE9 BX=0000 CX=0020 DX=0000 SP=003C BP=0000 SI=0000 DI=0020
DS=0DE9 ES=0DE9 SS=0DED CS=0DE7 IP=0016 NV UP EI PL NZ NA PO NC
0DE7:0016  F3  REPZ
0DE7:0017  A4  MOVSB
-F DS:0 1F FF
-F DS:20 3F 00
-F DS:0 3F

0DE9:0000  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0DE9:0010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0DE9:0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0DE9:0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
-G 16

AX=0DE9 BX=0000 CX=0000 DX=0000 SP=003C BP=0000 SI=0020 DI=0040
DS=0DE9 ES=0DE9 SS=0DED CS=0DE7 IP=0018 NV UP EI PL NZ NA PO NC
0DE7:0018  CB  RETP
-D DS:0 3F

0DE9:0000  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0DE9:0010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0DE9:0020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0DE9:0030  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
-G

Program terminated normally
-Q
```

8086 Microprocessors, Triebel and Singh 44