

CPE 408330

Assembly Language and Microprocessors

Chapter 6: 8088/8086 Microprocessor Programming – Control Flow Instructions and Program Structures

[Computer Engineering Department,
Hashemite University]

Lecture Outline

- ▶ 6.1 Flag–Control Instructions
- ▶ 6.2 Compare Instructions
- ▶ 6.3 Control Flow and Jump Instructions
- ▶ 6.4 Subroutines and Subroutine–Handling Instructions
- ▶ 6.5 The Loop and the Loop–Handling Instructions
- ▶ 6.6 String and String–Handling Instructions

6.1 Flag-Control Instructions

- ▶ The flag-control instructions, when executed, directly affect the state of the flags. These instructions include:
 - ☐ LAHF (Load AH from flags)
 - ☐ SAHF (Store AH into flags)
 - ☐ CLC (Clear carry)
 - ☐ STC (Set carry)
 - ☐ CMC (Complement carry)
 - ☐ CLI (Clear interrupt)
 - ☐ STI (Set interrupt)

6.1 Flag-Control Instructions -

Loading, Storing, and Modifying Flags

Mnemonic	Meaning	Operation	Flags affected
LAHF	Load AH from flags	$(AH) \leftarrow (Flags)$	None
SAHF	Store AH into flags	$(Flags) \leftarrow (AH)$	SF,ZF,AF,PF,CF
CLC	Clear carry flag	$(CF) \leftarrow 0$	CF
STC	Set carry flag	$(CF) \leftarrow 1$	CF
CMC	Complement carry flag	$(CF) \leftarrow \overline{(CF)}$	CF
CLI	Clear interrupt flag	$(IF) \leftarrow 0$	IF
STI	Set interrupt flag	$(IF) \leftarrow 1$	IF

- 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 over bar (NOT)

- Modifying the interrupt flag—**CLI/STI**

- Used to turn on/off external hardware interrupts

- Clear interrupt flag

CLI

$0 \rightarrow (IF)$ Disable interrupts

- Set interrupt flag

STI

$1 \rightarrow (IF)$ Enable interrupts

6.1 Flag-Control Instructions -

Debug Example

- Debug flag notation
 - $CF \rightarrow CY = 1, NC = 0$
 - Example—Execution of carry flag modification instructions
- $CY = 1 \rightarrow$ initial state
- CLC** ; Clear carry flag
- STC** ; Set carry flag
- CMC** ; Complement carry flag

```
C:\DOS>DEBUG
```

```
-A
```

```
1342:0100 CLC
```

```
1342:0101 STC
```

```
1342:0102 CMC
```

```
1342:0103
```

```
-R F
```

```
NV UP EI PL NZ NA PO NC -CY
```

```
-R F
```

```
NV UP EI PL NZ NA PO CY -
```

```
-T
```

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
```

```
DS=1342 ES=1342 SS=1342 CS=1342 IP=0101 NV UP EI PL NZ NA PO NC
```

```
1342:0101 F9 STC
```

```
-T
```

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
```

```
DS=1342 ES=1342 SS=1342 CS=1342 IP=0102 NV UP EI PL NZ NA PO CY
```

```
1342:0102 F5 CMC
```

```
-T
```

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
```

```
DS=1342 ES=1342 SS=1342 CS=1342 IP=0103 NV UP EI PL NZ NA PO NC
```

```
1342:0103 8AFF MOV BH,BH
```

```
-Q
```

```
C:\DOS>
```

6.1 Flag-Control Instructions -

Loading and Storing the Flags Register



SF = Sign flag
ZF = Zero flag
AF = Auxiliary
PF = Parity flag
CF = Carry flag
— = Undefined (do not use)

- 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

6.1 Flag-Control Instructions - Loading and Storing the Flags Register

▶ EXAMPLE

Write an instruction sequence to save the current contents of the 8088's flags in the memory location at offset MEM1 of the current data segment and then reload the flags with the contents of the storage location at offset MEM2.

▶ Solution:

LAHF	; Load AH from flags
MOV [MEM1], AH	; Move content of AH to MEM1
MOV AH, [MEM2]	; Load AH from MEM2
SAHF	; Store content of AH into flags

6.1 Flag-Control Instructions - Loading and Storing the Flags Register

▶ EXAMPLE

```
CONSOLE MODE - DEBUG
-
-A 0:0110
0000:0110 LAHF
0000:0111 MOV [0150],AH
0000:0115 MOV AH, [0151]
0000:0119 SAHF
0000:011A
-E 0:0150 FF 01
-R CS
CS 0B37
:0
-R IP
IP 0100
:0110
-R DS
DS 0B37
:0
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0000 ES=0B37 SS=0B37 CS=0000 IP=0110 NV UP EI PL NZ NA PO NC
0000:0110 9F LAHF
-
```


6.1 Flag-Control Instructions - Loading and Storing the Flags Register

▶ EXAMPLE

```
C:\> CONSOLE MODE - DEBUG

AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0111  NV UP EI PL NZ NA PO NC
0000:0111 88265001      MOV      [0150],AH      DS:0150=FF
-T

AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0115  NV UP EI PL NZ NA PO NC
0000:0115 8A265101      MOV      AH,[0151]      DS:0151=01
-D 150 151
0000:0150  02 01
-T

AX=0100  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=0119  NV UP EI PL NZ NA PO NC
0000:0119 9E          SAHF
-T

AX=0100  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0000  ES=0B37  SS=0B37  CS=0000  IP=011A  NV UP EI PL NZ NA PO CY
0000:011A 00F0      ADD      AL,DH
-T
```

6.1 Flag-Control Instructions - Loading and Storing the Flags Register

▶ EXAMPLE

Of the three carry flag instructions CLC, STC, and CMC, only one is really independent instruction. That is, the operation that it provides cannot be performed by a series of the other two instructions. Determine which one of the carry instruction is the independent instruction.

▶ Solution:

CLC \Leftrightarrow STC followed by a CMC

STC \Leftrightarrow CLC followed by a CMC

Therefore, only CMC is the independent instruction.

6.1 Flag-Control Instructions -

Loading and Storing the Flags Register

▶ EXAMPLE

Verify the operation of the following instructions that affect the carry flag,

CLC

STC

CMC

by executing them with the DEBUG program. Start with CF flag set to 1 (CY).

6.1 Flag-Control Instructions - Loading and Storing the Flags Register

► Solution:

```
CONSOLE MODE - DEBUG
OB37:0100 CLC
OB37:0101 STC
OB37:0102 CMC
OB37:0103
-R F
NV UP EI PL NZ NA PO NC -CY
-T

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0101 NV UP EI PL NZ NA PO NC
OB37:0101 F9 STC
-T

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0102 NV UP EI PL NZ NA PO CY
OB37:0102 F5 CMC
-T

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B37 ES=0B37 SS=0B37 CS=0B37 IP=0103 NV UP EI PL NZ NA PO NC
OB37:0103 AC LODSB
-
```

6.2 Compare Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
CMP	Compare	CMP D,S	(D) – (S) is used in setting or resetting the flags	CF,AF,OF,PF,SF,ZF

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

- 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) → 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

▶ EXAMPLE

Describe what happens to the status flags as the sequence of instructions that follows is executed.

MOV AX, 1234H

MOV BX, ABCDH

CMP AX, BX

▶ Solution:

$(AX) = 1234_{16} = 0001001000110100_2$

$(BX) = ABCD_{16} = 1010101111001101_2$

$(AX) - (BX) = 0001001000110100_2 - 1010101111001101_2$
 $= 0110\ 0110\ 0110\ 0111_2$

Therefore, $ZF = 0$, $SF = 0$, $OF = 0$, $PF = 0$, $CF = 1$, $AF = 1$

6.2 Compare Instruction

▶ EXAMPLE

```
CONSOLE MODE - DEBUG
-
-A
0B35:0100 MOV AX, 1234
0B35:0103 MOV BX, ABCD
0B35:0106 CMP AX, BX
0B35:0108
-T
AX=1234 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0103 NV UP EI PL NZ NA PO NC
0B35:0103 BBCDAB MOV BX,ABCD
-T
AX=1234 BX=ABCD CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0106 NV UP EI PL NZ NA PO NC
0B35:0106 39D8 CMP AX,BX
-T
AX=1234 BX=ABCD CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B35 ES=0B35 SS=0B35 CS=0B35 IP=0108 NV UP EI PL NZ AC PO CY
0B35:0108 41 INC CX
-
```

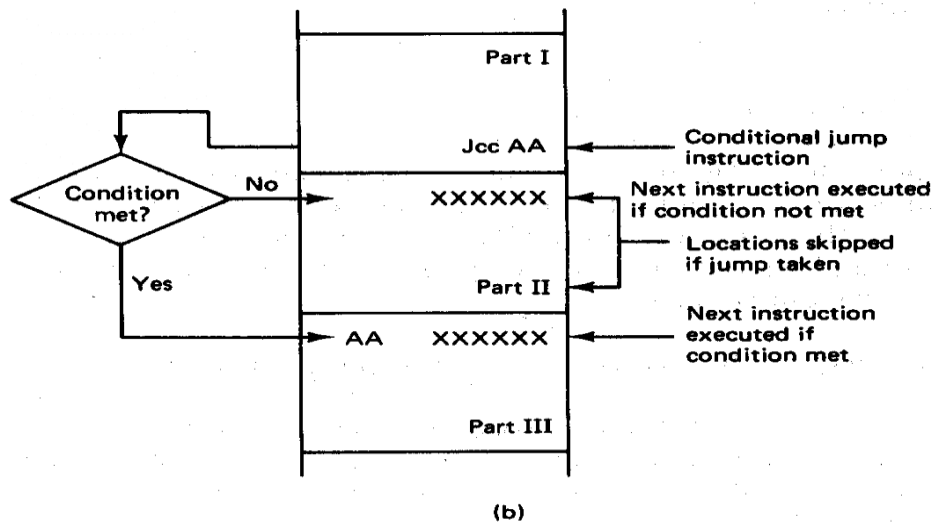
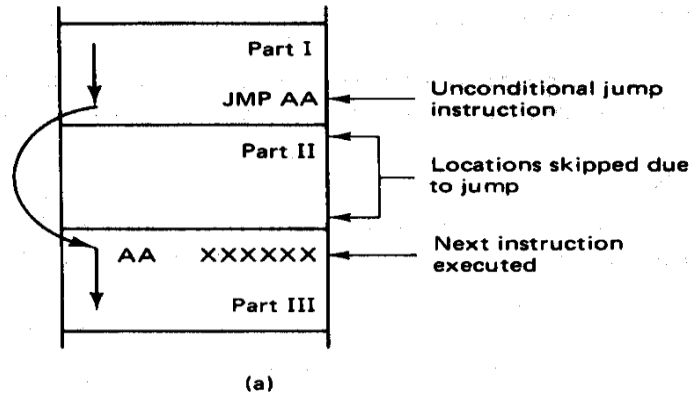
6.3 Control Flow and Jump Instructions

- Control flow: alternate the execution path of instructions in the program
 - What are the pointers that keep track of the instructions being executed ?
 - How it is possible to alternate the sequence of instructions being executed ?
- ☐ Unconditional jump instruction
 - ☐ Conditional jump instruction
 - ☐ Branching structure – IF–THEN
 - ☐ Loop program structure – REPEAT– UNTIL and WHILE–DO
 - ☐ Applications using the loop and branch software structures

Type of Jumps: How IP and CS are modified with Jumps

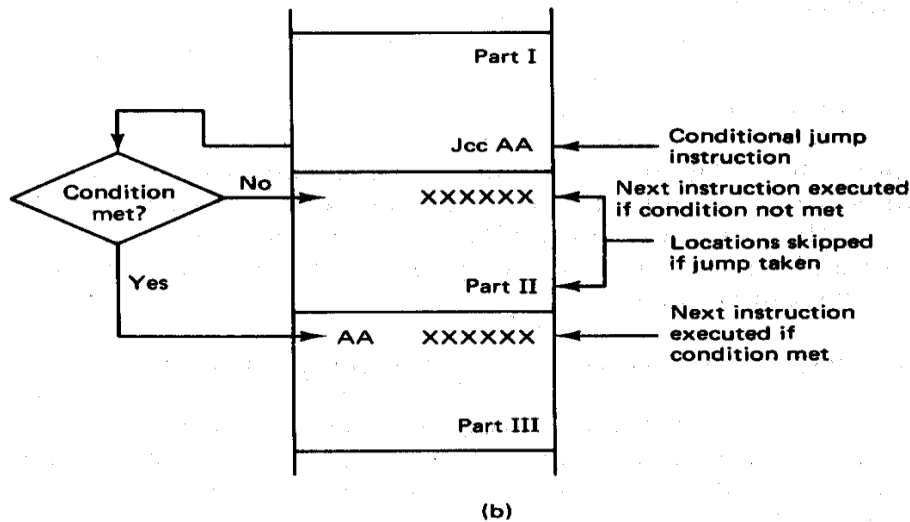
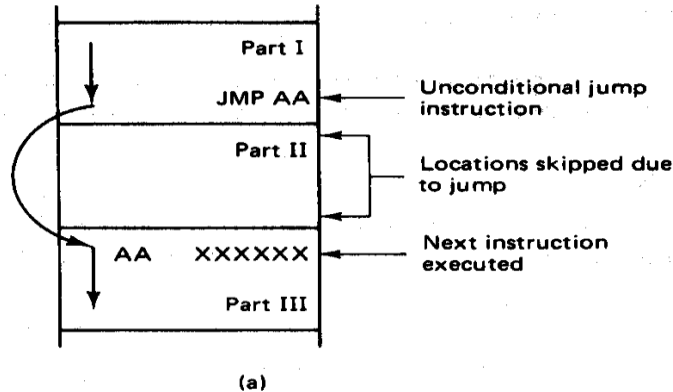
- ▶ **Intra-segment jump**: **modify IP**
 - **Short-label** specify 8-bit signed displacement (relative to the jump instruction)
 - **Near-label** specify IP with 16-bit immediate operand
 - **Memptr-16** and **Regptr-16** are same as Near-label but IP is specified as content of memory or register.
- ▶ **Inter-segment jump**: **modify IP and CS**
 - **Far-label**: uses 32-bit immediate operand to specify IP and CS
 - **Memptr-32**: uses 4 byte memory locations to specify IP and CS.

6.3 Control Flow and 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 (part a)
 - **JMP** AA instructions in Part I executed
 - Control passed to next instruction identified by AA in Part III
 - Instructions in Part II skipped

6.3 Control Flow and Jump Instructions– Unconditional and Conditional Jump Control Flow



No return linkage is saved when the JUMP is performed

- **Conditional** jump
 - May or may not take place
 - Status conditions must be satisfied
- Example (part b)
 - **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 Control Flow and Jump Instructions– Unconditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Affected flags
JMP	Unconditional jump	JMP Operand	Jump is initiated to the address specified by the operand	None

(a)

Operands
Short-label
Near-label
Far-label
Memptr16
Regptr16
Memptr32

(b)

CS:100	lab	ADD BX,1234
CS:104		INC AX
CS:106		JMP lab
CS:108		NEG BX

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 → (IP)

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

- **near-label**

- **16-bit** signed displacement coded in the instruction

- Example

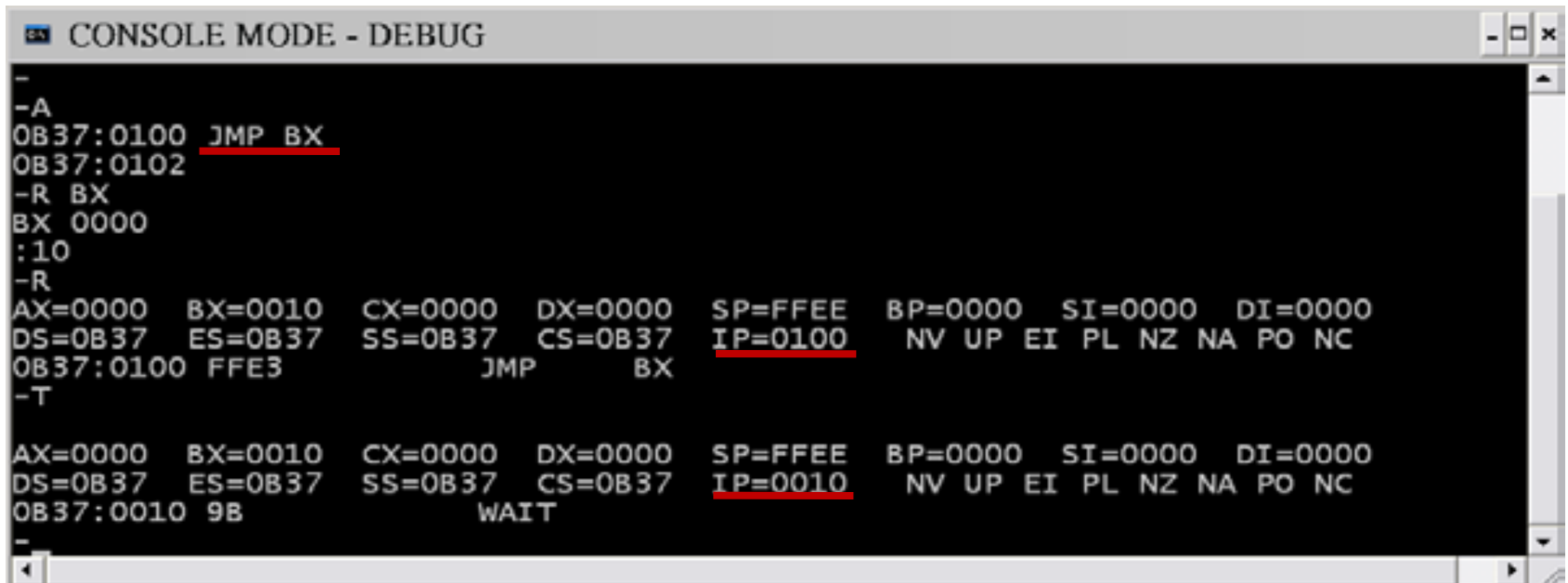
JMP 1234H

6.3 Control Flow and Jump Instructions – Unconditional Jump Instruction

▶ EXAMPLE

Verify the operation of the instruction `JMP BX` using the `DEBUG` program. Let the contents of `BX` be `001016`.

▶ Solution:



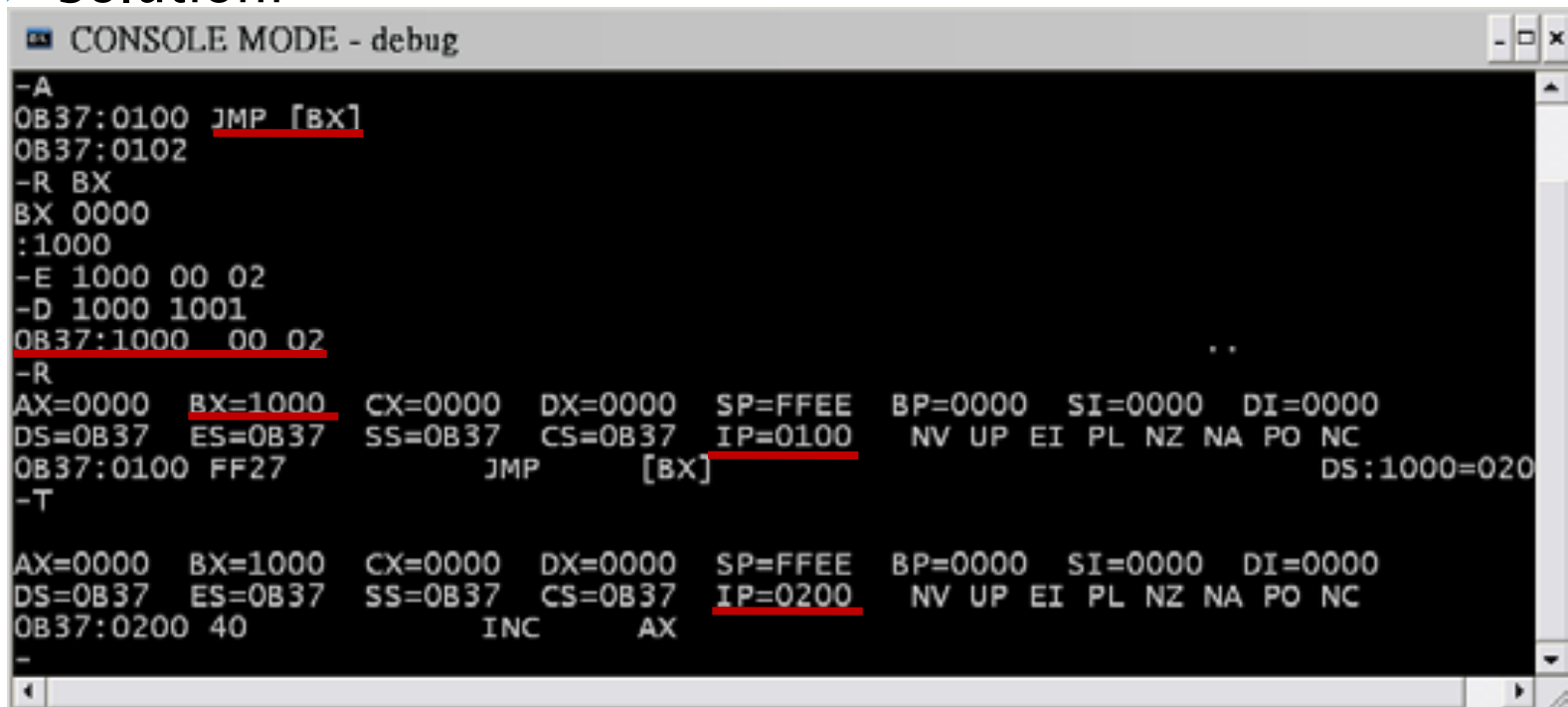
```
CONSOLE MODE - DEBUG
-
-A
0B37:0100 JMP BX
0B37:0102
-R BX
BX 0000
:10
-R
AX=0000  BX=0010  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0100  NV UP EI PL NZ NA PO NC
0B37:0100 FFE3             JMP     BX
-T
AX=0000  BX=0010  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0010  NV UP EI PL NZ NA PO NC
0B37:0010 9B             WAIT
-
```

6.3 Control Flow and Jump Instructions – Unconditional Jump Instruction

▶ EXAMPLE

Use the DEBUG program to observe the operation of the instruction `JMP [BX]`.

▶ Solution:



```

- A
0B37:0100  JMP [BX]
0B37:0102
- R BX
BX 0000
:1000
- E 1000 00 02
- D 1000 1001
0B37:1000 00 02
- R
AX=0000  BX=1000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0100  NV UP EI PL NZ NA PO NC
0B37:0100 FF27          JMP      [BX]          DS:1000=020
- T

AX=0000  BX=1000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B37  ES=0B37  SS=0B37  CS=0B37  IP=0200  NV UP EI PL NZ NA PO NC
0B37:0200 40          INC      AX
-

```

6.3 Control Flow and 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 → (**IP**)
 - **2nd** 16 bits → (**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) → new IP
 - (DS:DI + 2) → new CS
 - Jump to address = (New CS):(New IP)

6.3 Control Flow and Jump Instructions–

Conditional Jump Instruction

Mnemonic	Meaning	Format	Operation	Flags affected
Jcc	Conditional jump	Jcc Operand	If the specified condition cc is true the jump to the address specified by the operand is initiated; otherwise the next instruction is executed.	None

(a)

Mnemonic	Meaning	Condition
JA	above	CF = 0 and ZF = 0
JAE	above or equal	CF = 0
JB	below	CF = 1
JBE	below or equal	CF = 1 or ZF = 1
JC	carry	CF = 1
JCXZ	CX register is zero	(CF or ZF) = 0
JE	equal	ZF = 1
JG	greater	ZF = 0 and SF = OF
JGE	greater or equal	SF = OF
JL	less	(SF xor OF) = 1
JLE	less or equal	((SF xor OF) or ZF) = 1
JNA	not above	CF = 1 or ZF = 1
JNAE	not above nor equal	CF = 1
JNB	not below	CF = 0
JNBE	not below nor equal	CF = 0 and ZF = 0
JNC	not carry	CF = 0
JNE	not equal	ZF = 0
JNG	not greater	((SF xor OF) or ZF) = 1
JNGE	not greater nor equal	(SF xor OF) = 1
JNL	not less	SF = OF
JNLE	not less nor equal	ZF = 0 and SF = OF
JNO	not overflow	OF = 0
JNP	not parity	PF = 0
JNS	not sign	SF = 0
JNZ	not zero	ZF = 0
JO	overflow	OF = 1
JP	parity	PF = 1
JPE	parity even	PF = 1
JPO	parity odd	PF = 0
JS	sign	SF = 1
JZ	zero	ZF = 1

(b)

- Condition jump instruction
- Implements the conditional jump operation
- General format:

Jcc Operand

- cc = one of the supported conditional relationships
- Supports the same operand types as unconditional jump
- Operation: Flags tested for conditions defined by cc and:

If cc test **True**:

IP, or IP and CS are updated with new value

- Jump is taken
- Execution resumes at jump to target address

If cc test **False**:

IP, or IP and CS are unchanged

- Jump is not taken
- Execution continues with the next sequential instruction
- Examples of conditional tests:

JC = jump on carry → CF = 1

JPE/JIP = jump on parity even → PF = 1

JE/JZ = jump on equal → ZF = 1

These instructions are associated with the compare instruction usually

6.3 Control Flow and Jump Instructions— Branch Program Structures

```
CMP  AX, BX
JE   EQUAL
---  ---
      ; Next instruction if (AX) ≠ (BX)
      .
      .
      .
EQUAL: ---  ---
      ; Next instruction if (AX) = (BX)
      .
      .
      .
      ---  ---
```

- 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 $(AX) \neq (BX)$; $ZF = 0 \rightarrow$ **ELSE** path—next sequential instruction is executed
 - If $(AX) = (BX)$; $ZF = 1 \rightarrow$ **THEN** path—instruction pointed to by **EQUAL** executes
 - **JMP** instruction used in **ELSE** path to bypass the **THEN** path.

6.3 Control Flow and Jump Instructions– Branch Program Structures

```

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

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

.
.
.
---    ---

```

- 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) = xxxxxxxx AND 00000100
= 00000x00

if **bit 2** = 1 ZF = 0 (not zero)

if bit 2 = 0 ZF = 1

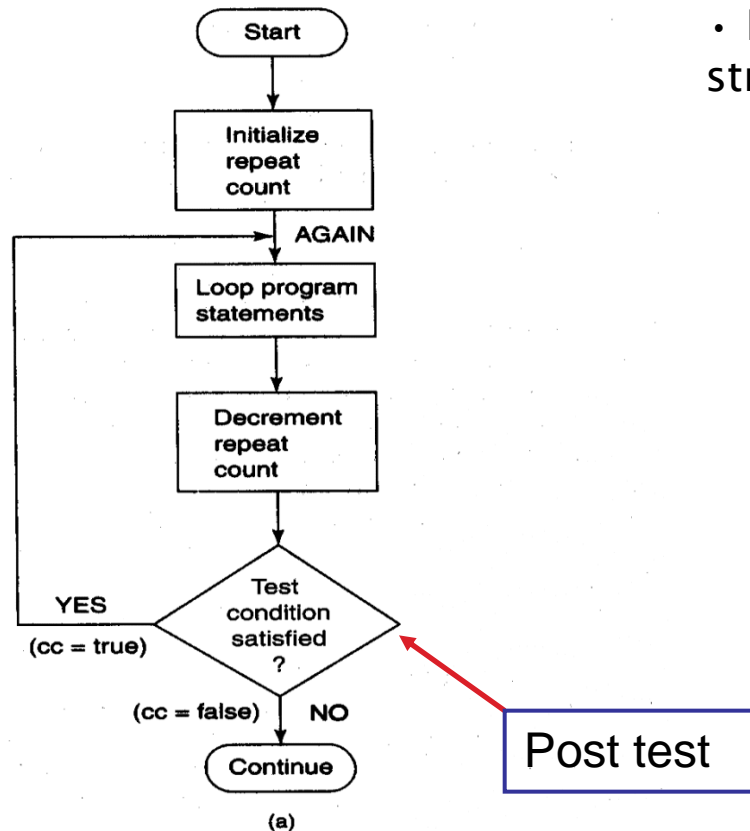
Therefore, jump to BIT2_ONE only takes place if bit 2 of AL equals 1

- Same operation can be performed by shifting bit 2 to the CF and then testing with **JC**

CF = 1

6.3 Control Flow and Jump Instructions– Loop Program Structures

Repeat Until structure



AGAIN:

MOV	CL,COUNT	;Set loop repeat count
---	---	;1st instruction of loop
---	---	;2nd instruction of loop
.	.	.
.	.	.
---	---	;nth instruction of loop
DEC	CL	;Decrement repeat count by 1
JNZ	AGAIN	;Repeat from AGAIN if (CL) ≠ 00H or (ZF) = 0
---	---	;First instruction executed after the loop is
---	---	;complete, (CL) = 00H, (ZF) = 1

(b)

- 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

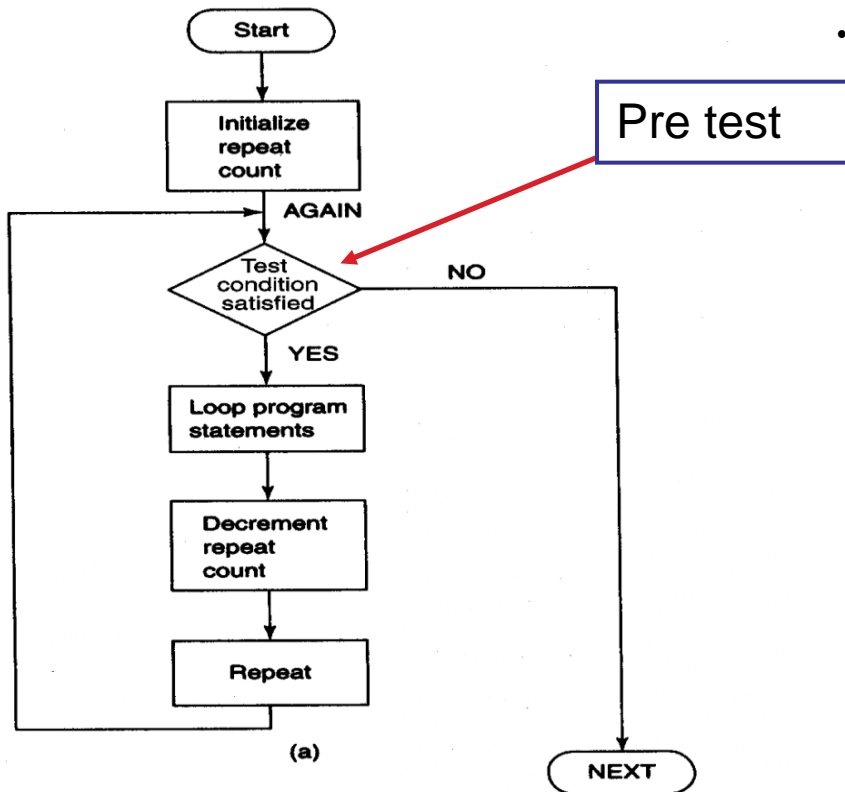
- Decrement count

DEC CL

- Conditional test for completion

JNZ AGAIN

6.3 Control Flow and Jump Instructions– Loop Program Structures



(a)

	MOV	CL,COUNT	;Set loop repeat count
AGAIN:	JZ	NEXT	;Loop is complete if CL = 00H (ZF = 1)
	---	---	;1st instruction of loop
	---	---	;2nd instruction of loop
	.	.	.
	.	.	.
	.	.	.
	---	---	;nth instruction of loop
	DEC	CL	;Decrement CL by 1
	JMP	AGAIN	;Repeat from AGAIN
NEXT:	---	---	;First instruction executed after loop is complete

(b)

- 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 → count register
 - Terminal count → zero or other value
 - Program flow/control:
 - Initialize count
`MOV CL,COUNT`
 - Pre-test
`AGAIN: JZ NEXT`
 - Perform body of loop operation
--- --- first of multiple instructions
 - Decrement count
`DEC CL`
 - Unconditional return to start of loop
`JMP AGAIN`

6.3 Control Flow and Jump Instructions – Loop Program Structures

▶ EXAMPLE

Implement an instruction sequence that calculates the absolute difference between the contents of AX and BX and places it in DX.

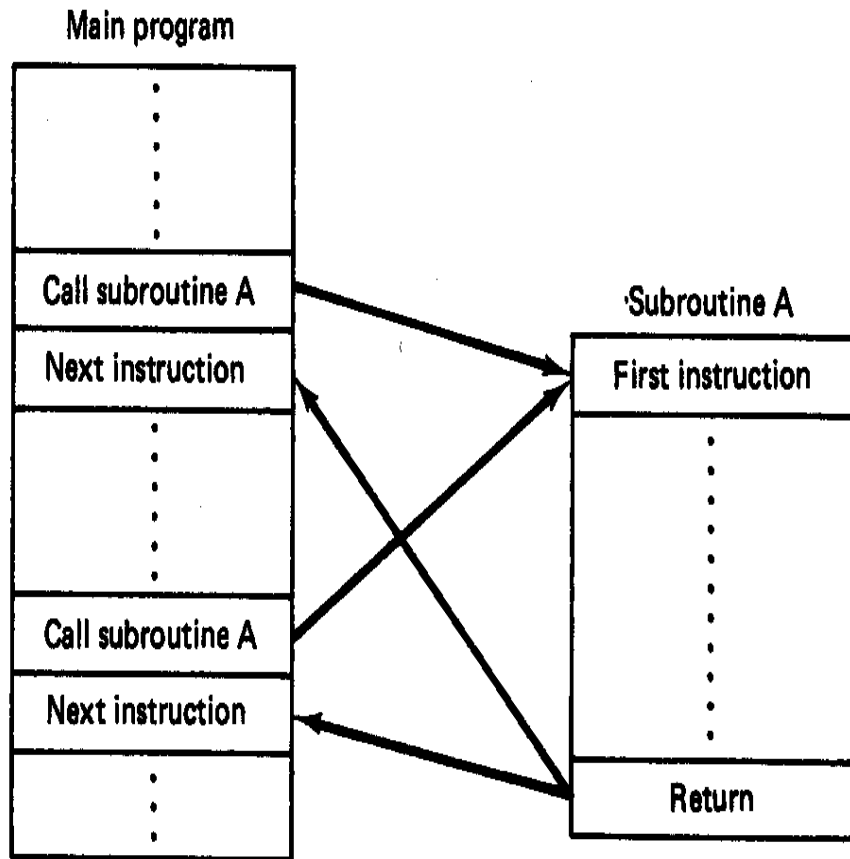
▶ Solution:

```
        CMP AX, BX
        JC DIFF2
DIFF1:  MOV DX, AX
        SUB DX, BX ; (DX)=(AX)-(BX)
        JMP DONE
DIFF2:  MOV DX, BX
        SUB DX, AX ; (DX)=(BX)-(AX)
DONE:  NOP
```

6.4 Subroutines and Subroutine-Handling Instructions

- **Subroutine**—special segment of program that can be called for execution from any point in a program (**like function**)
- A subroutine is also known as a procedure.
- 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
- A **return** instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.
- **CALL** and **RET** instructions
- **PUSH** and **POP** instructions

6.4 Subroutines and Subroutine-Handling Instructions



- 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

The subroutine may be called and executed more than one time, but it is written one time

6.4 Subroutines and Subroutine– Handling Instructions – Call Instruction

Mnemonic	Meaning	Format	Operation	Flags Affected
CALL	Subroutine call	CALL operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(b)

Operand
Near-proc
Far-proc
Memptr16
Regptr16
Memptr32

(c)

- 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 – Call Instruction

- **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 – Call Instruction

- **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:

- 1st 16 bits → New IP

- 2nd 16 bits → 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) → New IP

(DS:DI + 2) → 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 (intrasegment 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

Mnemonic	Meaning	Format	Operation	Flags Affected
RET	Return	RET or RET Operand	Return to the main program by restoring IP (and CS for fat-proc). If Operand is present, it is added to the contents of SP.	None

(a)

Operand
None Disp16

(b)

6.4 Subroutines and Subroutine– Handling Instructions – Return Instruction

▶ EXAMPLE

```
TITLE EXAMPLE 6.10
PAGE                                ,132
STACK_SEG S                        SEGMENT STACK 'STACK'
                                   DB 64 DUP(?)
STACK_SEG                          ENDS
CODE_SEG SEGMENT                  'CODE'
EX610 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 6.10
    CALL SUM
    RET
SUM PROC NEAR
    MOV DX, AX
    ADD DX, BX                ; (DX)=(AX)+(BX)
    RET
SUM ENDP
EX610 ENDP
CODE_SEG ENDS
EX610 END
```

6.4 Subroutines and Subroutine– Handling Instructions – Return Instruction

▶ EXAMPLE

```
C:\ CONSOLE MODE - DEBUG EX610.EXE
-
-U 0 D
0BB1:0000 1E          PUSH    DS
0BB1:0001 B80000      MOV     AX,0000
0BB1:0004 50          PUSH    AX
0BB1:0005 E80100      CALL    0009
0BB1:0008 CB          RETF
0BB1:0009 8BD0      MOV     DX,AX
0BB1:000B 03D3      ADD     DX,BX
0BB1:000D C3          RET
-G 5

AX=0000  BX=0000  CX=0314  DX=0000  SP=003C  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0005  NV UP EI PL NZ NA PO NC
0BB1:0005 E80100      CALL    0009
-R AX
AX 0000
:2
-R BX
BX 0000
:4
-
```

6.4 Subroutines and Subroutine– Handling Instructions – Return Instruction

▶ EXAMPLE

```

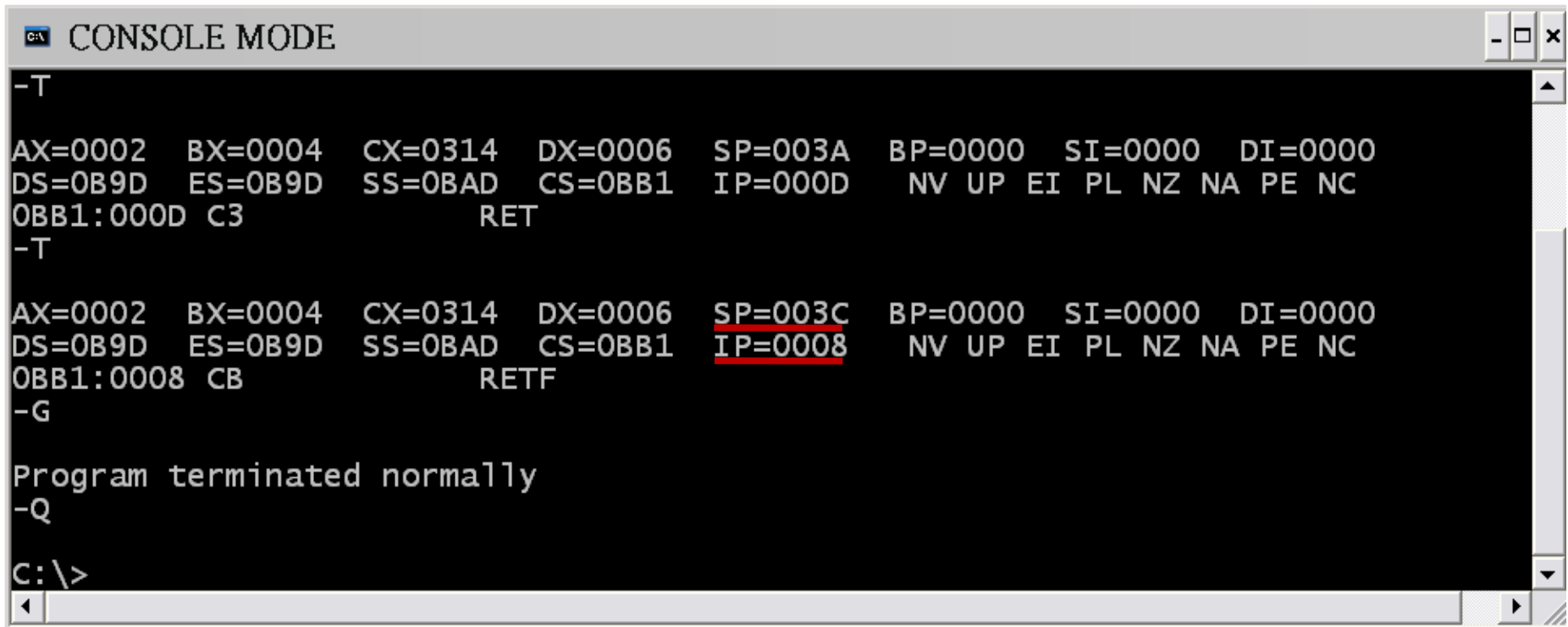
C:\> CONSOLE MODE - DEBUG EX610.EXE

-T
AX=0002  BX=0004  CX=0314  DX=0000  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0009  NV UP EI PL NZ NA PO NC
0BB1:0009 8BD0          MOV     DX,AX
-D SS:3A 3B
0BAD:0030          08 00          ..
-T
AX=0002  BX=0004  CX=0314  DX=0002  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000B  NV UP EI PL NZ NA PO NC
0BB1:000B 03D3          ADD     DX,BX
-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000D  NV UP EI PL NZ NA PE NC
0BB1:000D C3          RET
-

```

6.4 Subroutines and Subroutine– Handling Instructions – Return Instruction

▶ EXAMPLE



```
C:\> CONSOLE MODE

-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003A  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=000D  NV UP EI PL NZ NA PE NC
0BB1:000D C3          RET
-T
AX=0002  BX=0004  CX=0314  DX=0006  SP=003C  BP=0000  SI=0000  DI=0000
DS=0B9D  ES=0B9D  SS=0BAD  CS=0BB1  IP=0008  NV UP EI PL NZ NA PE NC
0BB1:0008 CB          RETF
-G

Program terminated normally
-Q

C:\>
```

6.4 Subroutines and Subroutine– Handling Instructions – Structure of a Subroutine

To save registers
and parameters
on the stack

{
PUSH XX
PUSH YY
PUSH ZZ

Main body of the
subroutine

{
:
:
:
:
:
:

To restore registers
and parameters
from the stack
Return to main
program

{
POP ZZ
POP YY
POP XX
RET

- **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

$(AH) \rightarrow ((SP)-1)$

$(AL) \rightarrow ((SP)-2)$

$(SP)-2 \rightarrow (SP) = \text{New top of stack}$

- Pop instruction

- General format:

POP D

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

- Example:

POP AX

$((SP)) \rightarrow AL$

$((SP)+1) \rightarrow AH$

$((SP)+2) \rightarrow SP = \text{Old top of stack}$

Mnemonic	Meaning	Format	Operation	Flags Affected
PUSH	Push word onto stack	PUSH S	$((SP)) \leftarrow (S)$ $(SP) \leftarrow (SP)-2$	None
POP	Pop word off stack	POP D	$(D) \leftarrow ((SP))$ $(SP) \leftarrow (SP)+2$	None

(a)

Operand (S or D)
Register
Seg-reg (CS illegal)
Memory

(b)

6.4 Subroutines and Subroutine– Handling Instructions – Push and Pop Instruction

► EXAMPLE

Write a procedure named SQUARE that squares the contents of BL and places the result in BX

► Solution:

```
;Subroutine:      SQUARE
;Description:     (BX)=square of (BL)
SQUARE  PROC  NEAR
    PUSH AX        ; Save the register to be used
    MOV AX, BX     ; Place the number in AL
    IMUL BL        ; Multiply with itself
    MOV BX, AX     ; Save the result
    POP AX         ; Restore the register used
    RET
SQUARE ENDP
```

6.4 Subroutines and Subroutine– Handling Instructions – Push Flag Instruction

- Push flags instruction

- General formats:

PUSHF

- Saves flags onto the stack

- Operation

$(\text{FLAGS}) \rightarrow ((\text{SP}))$

$(\text{SP}) - 2 \rightarrow (\text{SP}) = \text{New top of stack}$

- Pop flags instruction

- General formats:

POPF

- Restores flags from the stack

$((\text{SP})) \rightarrow \text{FLAGS}$

$(\text{SP}) + 2 \rightarrow (\text{SP}) = \text{Old top of stack}$

Mnemonic	Meaning	Operation	Flags Affected
PUSHF	Push flags onto stack	$((\text{SP})) \leftarrow (\text{Flags})$ $(\text{SP}) \leftarrow (\text{SP}) - 2$	None
POPF	Pop flags from stack	$(\text{Flags}) \leftarrow ((\text{SP}))$ $(\text{SP}) \leftarrow (\text{SP}) + 2$	OF, DF, IF, TF, SF, ZF, AF, PF, CF

6.5 The Loop and the Loop-Handling Instructions – Loop Instructions

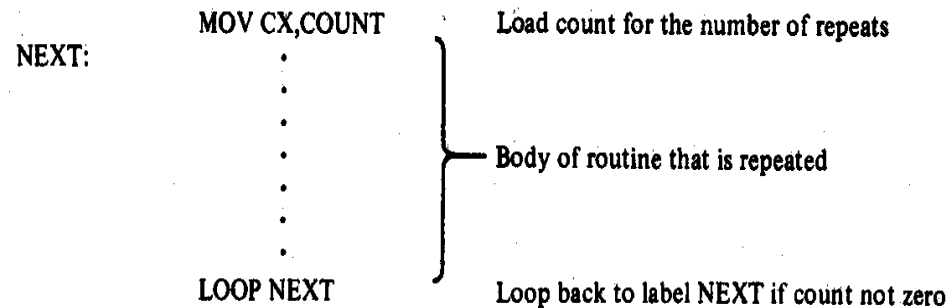
- **Loop**—segment of program that is repeatedly executed
 - Can be implemented with compare, conditional jump, and decrement instructions
- Loop instructions:
 - Special instructions that efficiently perform basic loop operations
 - Replace the multiple instructions with a single instruction
 - **LOOP**—loop while not zero (**while CX is not zero**)
 - $CX \neq 0$ — repeat while count not zero
 - **→Equivalent to Dec CX followed by JNZ**
 - **LOOPE/LOOPZ**— loop while equal
 - $CX \neq 0$ — repeat while count **not zero**, and
 - $ZF = 1$ —result of prior instruction was **equal**
 - **LOOPNE/LOOPNZ**—loop while not equal
 - $CX \neq 0$ — repeat while count **not zero**, and
 - $ZF = 0$ —result from prior instruction was **not equal**

NOTE All LOOPS instructions does not affect the flag register

6.5 The Loop and the Loop-Handling Instructions – Loop Instructions

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	$(CX) \leftarrow (CX) - 1$ Jump is initiated to location defined by short-label if $(CX) \neq 0$; otherwise, execute next sequential instruction
LOOPE/LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	$(CX) \leftarrow (CX) - 1$ Jump to location defined by short-label if $(CX) \neq 0$ and $(ZF) = 1$; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	$(CX) \leftarrow (CX) - 1$ Jump to location defined by short-label if $(CX) \neq 0$ and $(ZF) = 0$; otherwise, execute next sequential instruction

6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation



(a)

```

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
        LOOP    NXTPT
        HLT
    
```

(b)

- 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 $CX \neq 0$, loop of code is repeated by returning control to the instruction corresponding to the Short-Label (NXTPT:) operand

6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

➤ EXAMPLE

Given the following sequence of instructions, explain what happens as they are executed.

```
MOV DL, 05
MOV AX, 0A00H
MOV DS, AX
MOV SI, 0
MOV CX, 0FH
AGAIN: INC SI
      CMP [SI], DL
      LOOPNE AGAIN
```

6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

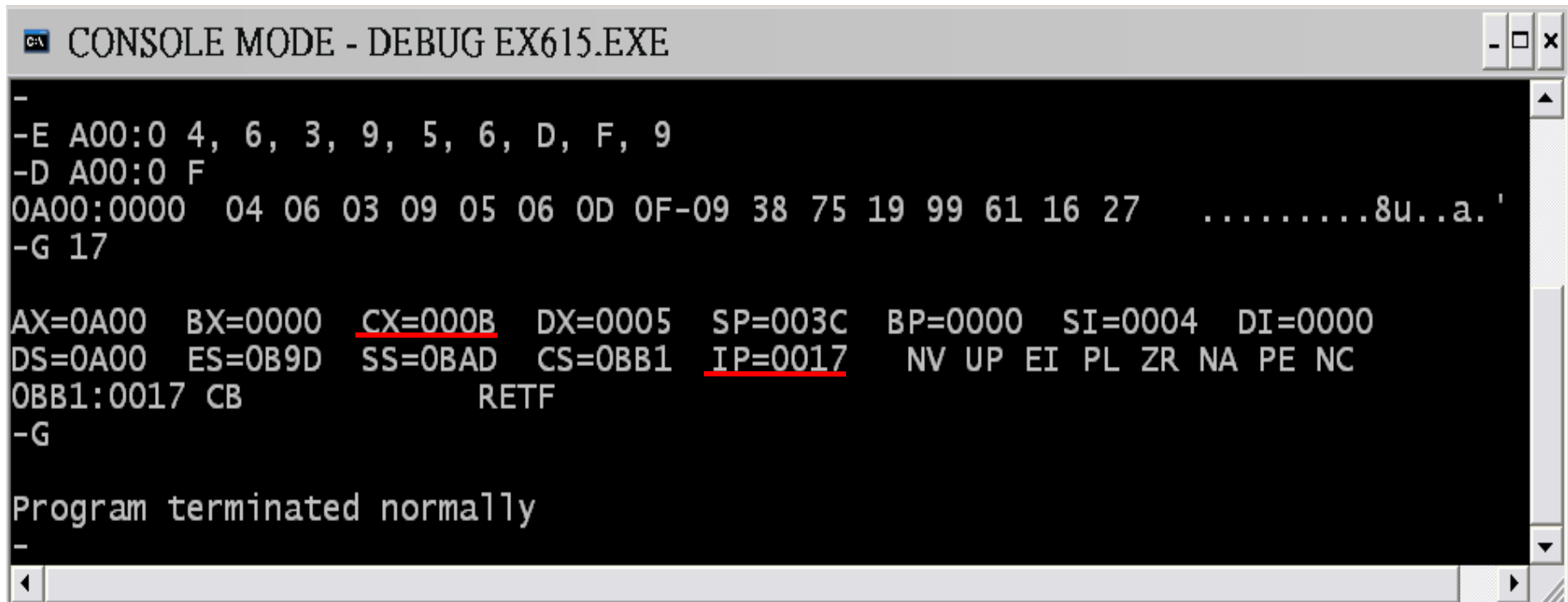
➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX615.EXE
C:\>DEBUG EX615.EXE
-U 0 17
0BB1:0000 1E      PUSH    DS
0BB1:0001 B80000    MOV     AX,0000
0BB1:0004 50      PUSH    AX
0BB1:0005 B205    MOV     DL,05
0BB1:0007 B8000A    MOV     AX,0A00
0BB1:000A 8ED8    MOV     DS,AX
0BB1:000C BE0000    MOV     SI,0000
0BB1:000F B90F00    MOV     CX,000F
0BB1:0012 46      INC     SI
0BB1:0013 3814    CMP     [SI],DL
0BB1:0015 E0FB    LOOPNZ 0012
0BB1:0017 CB      RETF
-G 12

AX=0A00  BX=0000  CX=000F  DX=0005  SP=003C  BP=0000  SI=0000  DI=0000
DS=0A00  ES=0B9D  SS=0BAD  CS=0BB1  IP=0012  NV UP EI PL NZ NA PO NC
0BB1:0012 46      INC     SI
-
```


6.5 The Loop and the Loop-Handling Instructions – Loop Program Structure and Operation

➤ EXAMPLE



```
CONSOLE MODE - DEBUG EX615.EXE
-
-E A00:0 4, 6, 3, 9, 5, 6, D, F, 9
-D A00:0 F
0A00:0000 04 06 03 09 05 06 0D 0F-09 38 75 19 99 61 16 27 .....8u..a.'
-G 17
AX=0A00 BX=0000 CX=000B DX=0005 SP=003C BP=0000 SI=0004 DI=0000
DS=0A00 ES=0B9D SS=0BAD CS=0BB1 IP=0017 NV UP EI PL ZR NA PE NC
0BB1:0017 CB RETF
-G
Program terminated normally
-
```

6.6 String 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

6.6 String and String-Handling Instructions – String Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
MOVS	Move string	MOVSB/MOVSW	$((ES)0 + (DI)) \leftarrow ((DS)0 + (SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None
CMPS	Compare string	CMPSB/CMPSW	Set flags as per $((DS)0 + (SI)) - ((ES)0 + (DI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/SCASW	Set flags as per $(AL \text{ or } AX) - ((ES)0 + (DI))$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/LODSW	$(AL \text{ or } AX) \leftarrow ((DS)0 + (SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$	None
STOS	Store string	STOSB/STOSW	$((ES)0 + (DI)) \leftarrow (AL \text{ or } AX) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None

6.6 String 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:
 - MOVS****B**—move string byte
 - MOVS****W**—move string word
 - Operation: Copies the content of the source to the destination; autoincrements/decrements both the source and destination addresses
 $((DS)0 + (SI)) \rightarrow ((ES)0 + (DI))$
 $(SI) \pm 1 \text{ or } 2 \rightarrow (SI)$
 $(DI) \pm 1 \text{ or } 2 \rightarrow (DI)$
 - **Direction flag** determines increment/decrement
 $DF = 0 \rightarrow \text{autoincrement}$
 $DF = 1 \rightarrow \text{autodecrement}$

6.6 String and String-Handling Instructions – Move String Instruction

```
MOV     AX,DATASEGADDR
MOV     DS,AX
MOV     ES,AX
MOV     SI,BLK1ADDR
MOV     DI,BLK2ADDR
MOV     CX,N
CLD
NXTPT:  MOVSB
        LOOP  NXTPT
        HLT
```

Reset



- Application example— The block-move program using the move-string instruction:
 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

6.6 String 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
 $(AL,AX) - ((ES)0 + (DI))$

6.6 String and String-Handling Instructions – Compare/Scan String Instructions

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


- 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

6.6 String 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
 $((DS)0 + (SI)) \rightarrow (AL \text{ or } AX)$
 update status flags
 $(SI) \pm 1 \text{ or } 2 \rightarrow (SI)$
- **Store** string instruction—**STOS**
- Same operation as **LODS** except value in accumulator is stored in destination is memory
 $(AL, AX) \rightarrow ((ES)0 + (DI))$

6.6 String and String-Handling Instructions – Load/Store String Instructions

```
MOV     AX,0
MOV     DS,AX
MOV     ES,AX
MOV     AL,05
MOV     DI,0A000H
MOV     CX,0FH
CLD
AGAIN:  STOSB
        LOOP    AGAIN
```



- Application example—
initializing a block of memory
with a store string instruction:
 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

How many times will this loop execute ?

6.6 String 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 string 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**

6.6 String and String-Handling Instructions – **Repeat String Instructions**

Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string $CX \neq 0$
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal $CX \neq 0$ and $ZF = 1$
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal $CX \neq 0$ and $ZF = 0$

6.6 String and String-Handling Instructions – Repeat String Instructions

- General format:

REPXXXX

Where: XXXX = one of string instructions

- Examples:

REPMOVB

REPESCAS

REPNECASCAS

- 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
4. Clear the direction flag for autoincrement mode
4. Repeat store string until all elements of the string are initialized to 05H

```
MOV     AX,0
MOV     DS,AX
MOV     ES,AX
MOV     AL,05
MOV     DI,0A000H
MOV     CX,0FH
CLD
REPSTOSB
```

6.6 String and String-Handling Instructions – Repeat String Instructions

➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX617.EXE

-U 0 18
0BB5:0000 1E          PUSH     DS
0BB5:0001 B80000        MOV      AX,0000
0BB5:0004 50          PUSH     AX
0BB5:0005 B8B10B        MOV      AX,0BB1
0BB5:0008 8ED8        MOV      DS,AX
0BB5:000A 8EC0        MOV      ES,AX
0BB5:000C FC          CLD
0BB5:000D B92000        MOV      CX,0020
0BB5:0010 BE0000        MOV      SI,0000
0BB5:0013 BF2000        MOV      DI,0020
0BB5:0016 F3          REPZ
0BB5:0017 A4          MOVSB
0BB5:0018 CB          RETF
-G 16

AX=0BB1  BX=0000  CX=0020  DX=0000  SP=003C  BP=0000  SI=0000  DI=0020
DS=0BB1  ES=0BB1  SS=0BAD  CS=0BB5  IP=0016  NV UP EI PL NZ NA PO NC
0BB5:0016 F3          REPZ
0BB5:0017 A4          MOVSB
```

6.6 String and String-Handling Instructions – Repeat String Instructions

➤ EXAMPLE

```
CONSOLE MODE - DEBUG EX617.EXE

-F DS:0 1F FF
-F DS:20 3F 00
-D DS:0 3F
OBB1:0000  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0010  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
OBB1:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
-G 18

AX=0BB1  BX=0000  CX=0000  DX=0000  SP=003C  BP=0000  SI=0020  DI=0040
DS=0BB1  ES=0BB1  SS=0BAD  CS=0BB5  IP=0018  NV UP EI PL NZ NA PO NC
OBB5:0018  CB                RETF
-D DS:0 3F
OBB1:0000  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0010  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0020  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
OBB1:0030  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF .....
-G

Program terminated normally
-
```

H.W. #6

- ❑ Solve the following problems from Chapter 6 from the course textbook:

1, 8, 14, 28, 39, 43