CSE213

Microcontroller Programming

Data Movement Instructions

Introduction

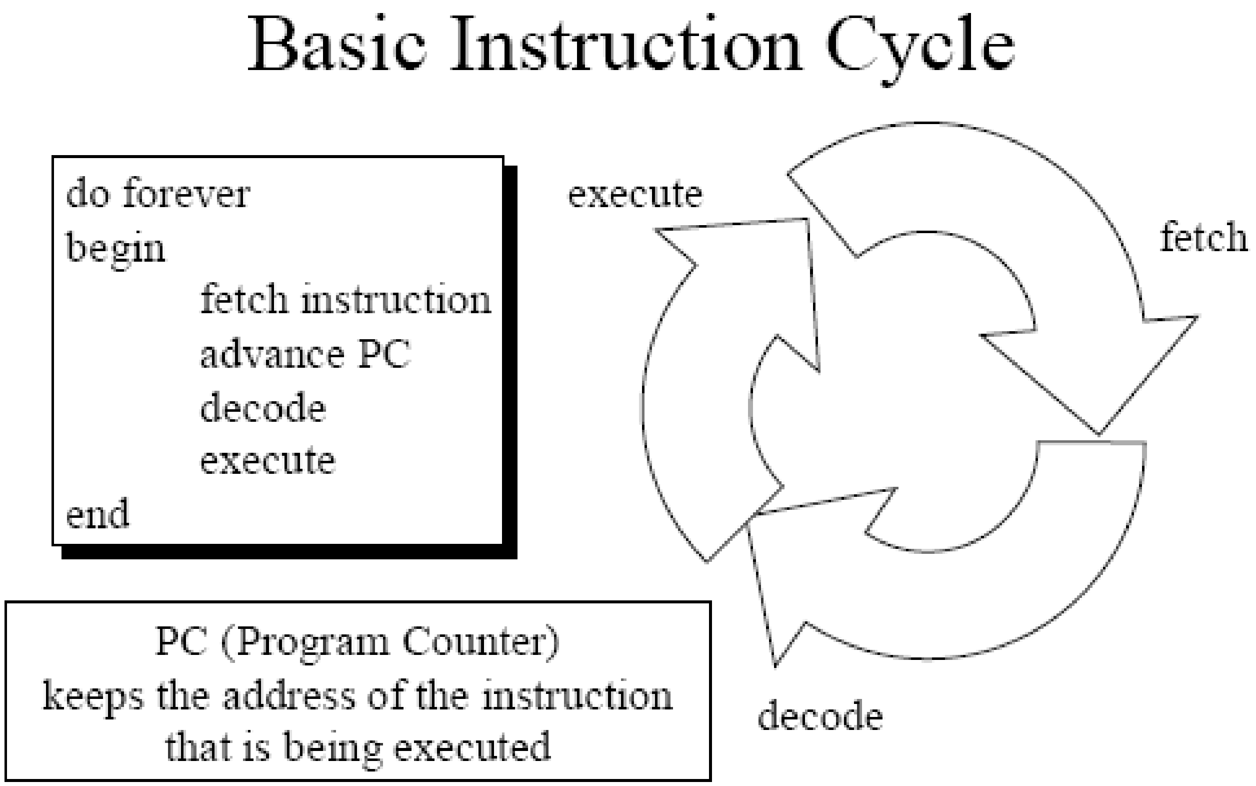
- This chapter concentrates on common data movement instructions.

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Explain the operation of each data movement instruction with applicable addressing modes.
- Select the appropriate assembly language instruction to accomplish a specific data movement task.

Instruction Cycle


Basic Instruction Cycle

```
do forever
begin
        fetch instruction
        advance PC
        decode
        execute
end
```

PC (Program Counter)
keeps the address of the instruction
that is being executed

Machine Language

- Native binary code microprocessor uses as its instructions to control its operation.
  - instructions vary in length from 1 to 13 bytes
- Over 100,000 variations of machine language instructions.
  - there is no complete list of these variations
- Some bits in a machine language instruction are given; remaining bits are determined for each variation of the instruction.

Instruction Format

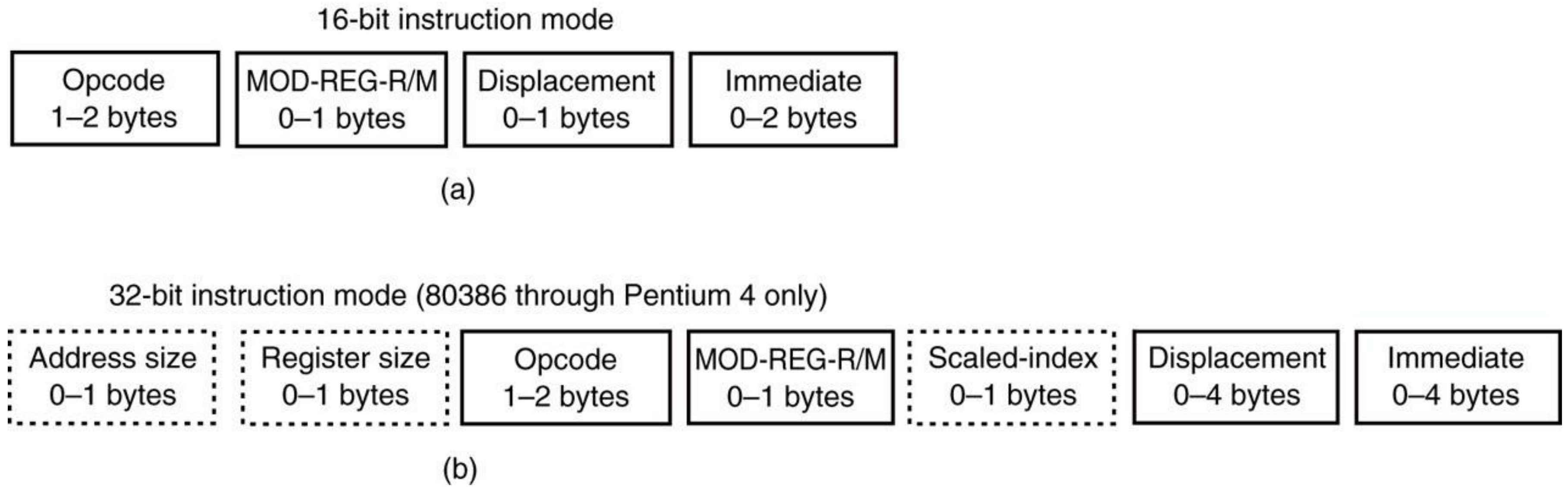


Figure 4–1 The formats of the 8086–Core2 instructions. (a) The 16-bit form and (b) the 32-bit form.

The Opcode

- Selects the operation (addition, subtraction, etc.,) performed by the microprocessor.
- Figure below illustrates the general form of the first opcode byte of many instructions.
  - first 6 bits of the first byte are the binary opcode
  - remaining 2 bits indicate the **direction** (D) of the data flow, and indicate whether the data are a byte or a word (W)

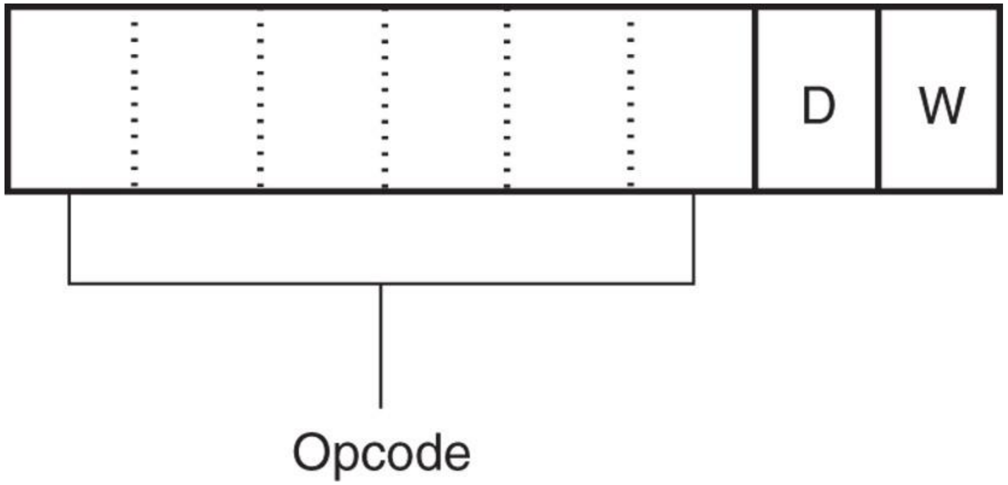
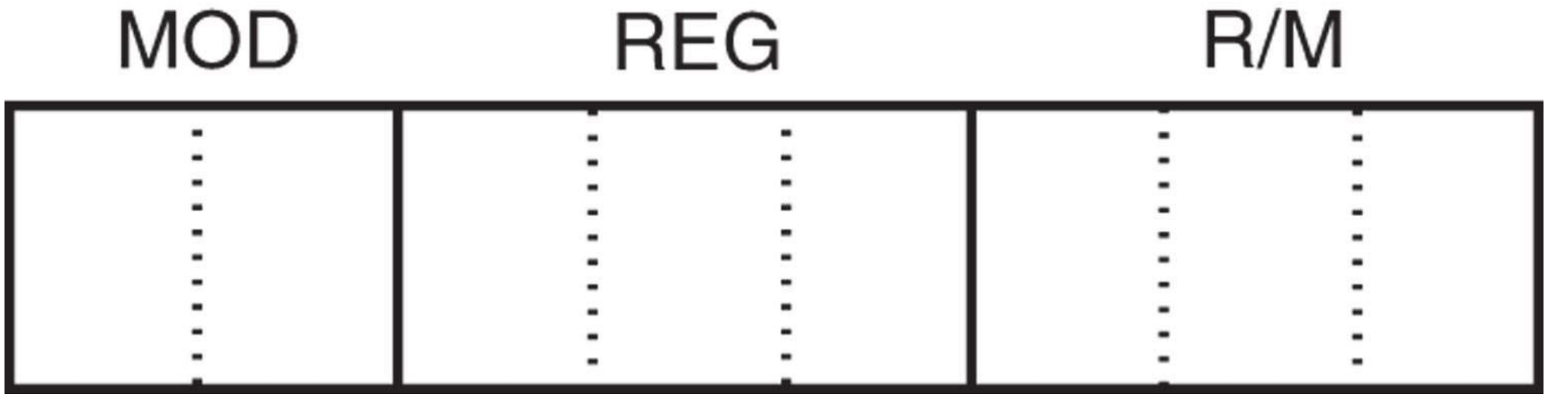


Figure 4–3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.



MOD Field

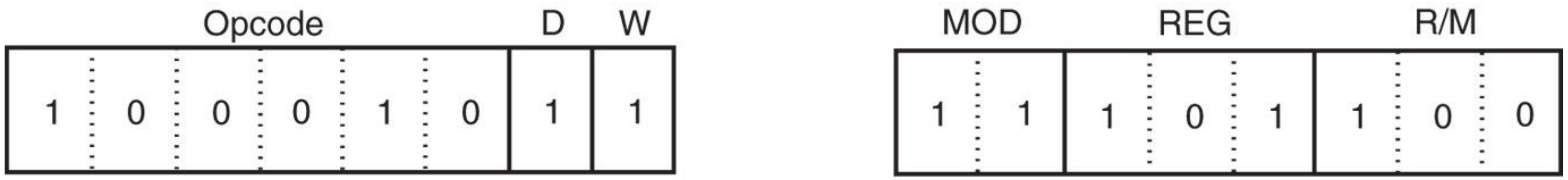
- Specifies addressing mode (MOD) and whether a displacement is present with the selected type.
  - If MOD=11, register-addressing mode is selected
  - Register addressing specifies a register instead of a memory location, using the R/M field
- If the MOD field contains a 00, 01, or 10, the R/M field selects one of the data memory-addressing modes.

- All 8-bit displacements are sign-extended into 16-bit displacements when the processor executes the instruction.
  - if the 8-bit displacement is 00H–7FH (positive), it is sign-extended to 0000H–007FH before adding to the offset address
  - if the 8-bit displacement is 80H–FFH (negative), it is sign-extended to FF80H–FFFFH

Register Assignments

- Suppose a 2-byte instruction, 8BECH, appears in a machine language program.
- In 16-bit mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2, as illustrated in Figure 4–4.

Figure 4–4 The 8BEC instruction placed into bytes 1 and 2 formats from Figures 4–2 and 4–3. This instruction is a MOV BP,SP.



Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP

– the opcode is 100010, a MOV instruction

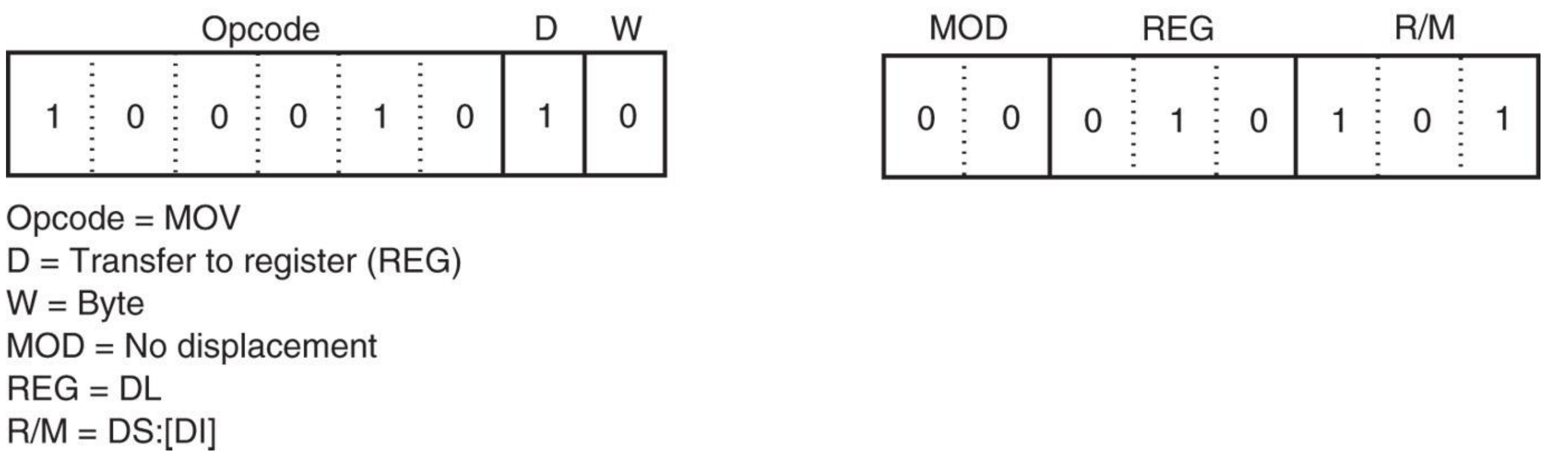
– D and W bits are a logic 1, so a word moves into the destination register specified in the REG field

– REG field contains 101, indicating register BP, so the MOV instruction moves data into register BP

R/M Memory Addressing

- If the MOD field contains a 00, 01, or 10, the R/M field takes on a new meaning.
- Figure 4–5 illustrates the machine language version of the 16-bit instruction MOV DL,[DI] or instruction (8A15H).
- This instruction is 2 bytes long and has an opcode 100010, D=1 (to REG from R/M), W=0 (byte), MOD=00 (no displacement), REG=010 (DL), and R/M=101 ([DI]).

Figure 4–5 A MOV DL,[DI] instruction converted to its machine language form.



– If the instruction changes to MOV DL, [DI+1], the MOD field changes to 01 for 8-bit displacement

– first 2 bytes of the instruction remain the same

– instruction now becomes 8A5501H instead of 8A15H

- Because the MOD field contains a 11, the R/M field also indicates a register.
- R/M = 100(SP); therefore, this instruction moves data from SP into BP.
  - written in symbolic form as a MOV BP,SP instruction
- The assembler program keeps track of the register- and address-size prefixes and the mode of operation.

Data Types

- In order to tell the assembler about data type, these prefixes should be used:
  - **BYTE PTR** - for byte.
  - **WORD PTR** - for word (two bytes)
- Examples:
  - **MOV AL, BYTE PTR [BX]** ; byte access
  - **MOV CX, WORD PTR [BX]** ; word access
- Assembler supports shorter prefixes as well:
  - **B.** - for **BYTE PTR**
  - **W.** - for **WORD PTR**
- In certain cases the assembler can calculate the data type automatically.

MOV Instruction

- Copies the **second operand** (source) to the **first operand** (destination).
- The source operand can be an immediate value, general-purpose register or memory location.
- The destination register can be a general-purpose register, or memory location.
- Both operands must be the same size, which can be a byte or a word.
- the **MOV** instruction <u>cannot</u> be used to set the value of the **CS** and **IP** registers.

Operands of MOV

- These types of operands are supported:
  - MOV REG, memory
  - MOV memory, REG
  - MOV REG, REG
  - MOV memory, immediate
  - MOV REG, immediate
- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **memory**: [BX], [BX+SI+7], variable, etc.
- **immediate**: 5, -24, 3Fh, 10001101b, etc.

Segment Register Operands

- For segment registers only these types of **MOV** are supported:
  - MOV SREG, memory
  - MOV memory, SREG
  - MOV REG, SREG
  - MOV SREG, REG
- **SREG**: DS, ES, SS, and only as second operand: CS.
- **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
- **memory**: [BX], [BX+SI+7], variable, etc.

MOV Example

```
ORG 100h              ; this directive required
MOV AX, 0B800h        ; set AX to hexadecimal value of B800h.
MOV DS, AX            ; copy value of AX to DS.
MOV CL, 'A'           ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 11011111b     ; set CH to binary value.
MOV BX, 15Eh          ; set BX to 15Eh.
MOV [BX], CX          ; copy contents of CX to memory at B800:015E
RET                   ; returns to operating system.
```

Variables

- Syntax for a variable declaration:
  - *name* **DB** *value*
  - *name* **DW** *value*
- **DB** - stays for Define Byte.
- **DW** - stays for Define Word.
- *name* - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).
- *value* - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

Example

```
ORG 100h
MOV AL, var1
MOV BX, var2
RET ; stops the program.
var1 DB 7
var2 DW 1234H
```

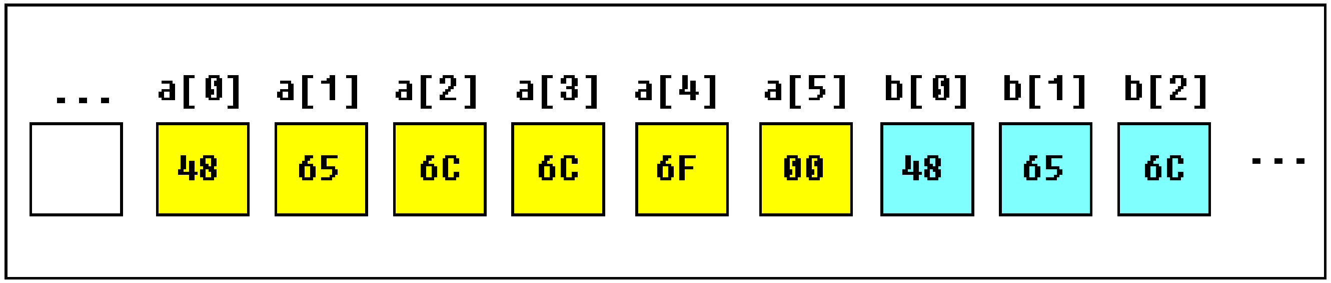
ORG Directive

- **ORG 100h** is a assembler directive (it tells assembler how to handle the source code).
- It tells assembler that the executable file will be loaded at the **offset** of 100h (256 bytes), so assembler should calculate the correct address for all variables when it replaces the variable names with their **offsets**.
- Directives are never converted to any real **machine code**.
- Why executable file is loaded at **offset** of **100h**?
- Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.

Arrays

- Arrays can be seen as chains of variables.
- A text string is an example of a byte array, each character is represented as an ASCII code value (0..255).
- Examples:
  - **a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h**
  - **b DB 'Hello', 0**
- **b** is an exact copy of the **a** array, when assembler sees a string inside quotes it automatically converts it to set of bytes.

Accessing Array Elements



- You can access the value of any element in array using square brackets, for example:
  - MOV AL, a[3]
- You can also use any of the memory index registers **BX, SI, DI, BP**, for example:
  - MOV SI, 3
  - MOV AL, a[SI]

Declaring Large Arrays

- If you need to declare a large array you can use **DUP** operator.
- The syntax for **DUP**:
  - number DUP (value(s))
  - number - number of duplicate to make (any constant value).
  - value - expression that DUP will duplicate.
- Example:

  c DB 5 DUP(9)

  is an alternative way of declaring:

  c DB 9, 9, 9, 9, 9

Declaring Large Arrays

- One more example:

  d DB 5 DUP(1, 2)

  is an alternative way of declaring:

  d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
- Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128.
- **DW** cannot be used to declare strings.

Getting the Address of a Variable

- The **LEA** instruction can be used to get the offset address of a variable.
- Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Example

```
ORG 100h
MOV AL, VAR1 ; check value of VAR1 by moving it to AL.
LEA BX, VAR1    ; get address of VAR1 in BX.
MOV BYTE PTR [BX], 44h ; modify the contents of VAR1.
MOV AL, VAR1 ; check value of VAR1 by moving it to AL.
RET
VAR1 DB 22h
END
```

Constants

- Constants are just like variables, but they exist only until your program is compiled (assembled).
- After definition of a constant its value cannot be changed.
- To define constants **EQU** directive is used:

  name EQU < any expression >
- Example:

  k EQU 5

  MOV AX, k
- The above example is functionally identical to code:

  MOV AX, 5

PUSH

- Always transfers 2 bytes of data to the stack;
  - 80386 and above transfer 2 or 4 bytes
- PUSHA instruction copies contents of the internal register set, except the segment registers, to the stack.
- PUSHA (**push all**) instruction copies the registers to the stack in the following order: AX, CX, DX, BX, SP, BP, SI, and DI.

- PUSHF (**push flags**) instruction copies the contents of the flag register to the stack.
- PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set in 80386 - Pentium 4.
  - PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4

POP

- Performs the inverse operation of PUSH.
- POP removes data from the stack and places it in a target 16-bit register, segment register, or a 16-bit memory location.
  - not available as an immediate POP
- POPF (pop flags) removes a 16-bit number from the stack and places it in the flag register;
  - POPFD removes a 32-bit number from the stack and places it into the extended flag register

- POPA (pop all) removes 16 bytes of data from the stack and places them into the following registers, in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX.
  - reverse order from placement on the stack by PUSHA instruction, causing the same data to return to the same registers

4–4 STRING DATA TRANSFERS

- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.

The Direction Flag

- The direction flag (D, located in the flag register) selects auto-increment or auto-decrement operation for the DI and SI registers during string operations.
  - used only with the string instructions
- The CLD instruction clears the D flag and the STD instruction sets it .
  - CLD instruction selects the auto-increment mode and STD selects the auto-decrement mode

DI and SI

- During execution of string instruction, memory accesses occur through DI and SI registers.
  - DI offset address accesses data in the extra segment for all string instructions that use it
  - SI offset address accesses data by default in the data segment

LODSB Instruction

- Load byte at DS:[SI] into AL. Update SI.

  Algorithm:
  AL = DS:[SI]
- if DF = 0 then
  - SI = SI + 1
- else
  - SI = SI - 1

Example

```
ORG 100h
LEA SI, a1
MOV CX, 5
MOV AH, 0Eh
m: LODSB
INT 10h
LOOP m
RET
a1 DB 'H', 'e', 'l', 'l', 'o'
```

STOSB Instruction

- Store byte in AL into ES:[DI]. Update DI.

  Algorithm:
  ES:[DI] = AL
- if DF = 0 then
  - DI = DI + 1
- else
  - DI = DI - 1

Example

```
ORG 100h
LEA DI, a1
MOV AL, 12h
MOV CX, 5
REP
STOSB
RET
a1 DB 5 dup(0)
```

MOVSB Instruction

- Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.

  Algorithm:
  ES:[DI] = DS:[SI]

- if DF = 0 then
  - SI = SI + 1
  - DI = DI + 1
- else
  - SI = SI - 1
  - DI = DI - 1

Example

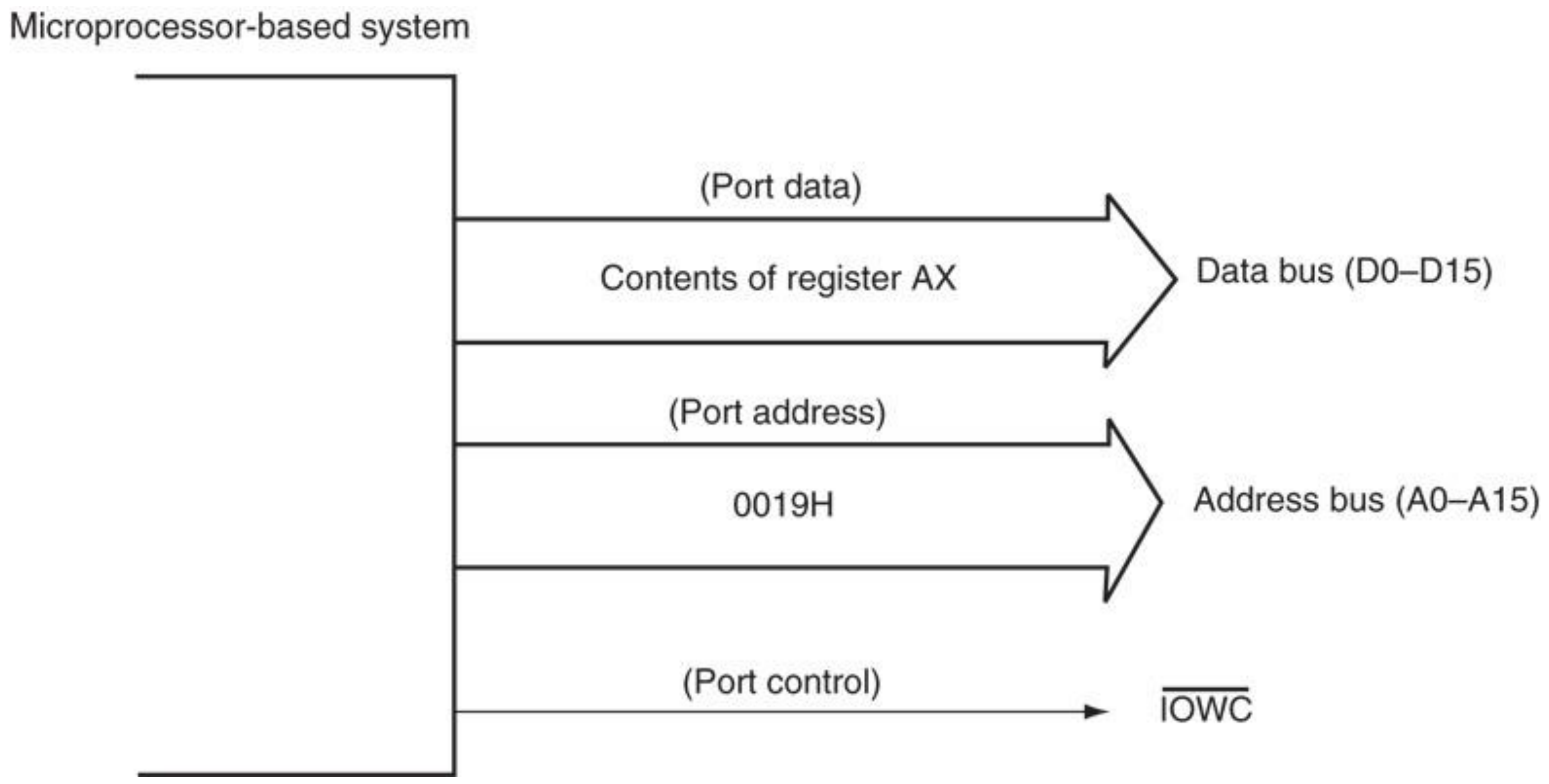
```
ORG 100h
CLD
LEA SI, a1
LEA DI, a2
MOV CX, 5
REP
MOVSB
RET
a1 DB 1,2,3,4,5
a2 DB 5 DUP(0)
```

IN and OUT

- IN & OUT instructions perform I/O operations.
- Contents of AL, AX, or EAX are transferred only between I/O device and microprocessor.
  - an IN instruction transfers data from an external I/O device into AL, AX, or EAX
  - an OUT transfers data from AL, AX, or EAX to an external I/O device

- Two forms of I/O device (port) addressing:
- *Fixed-port addressing* allows data transfer between AL, AX, or EAX using an 8-bit I/O port address.
  - port number follows the instruction's opcode
- *Variable-port addressing* allows data transfers between AL, AX, or EAX and a 16-bit port address.
  - the I/O port number is stored in register DX, which can be changed (varied) during the execution of a program.

Figure 4–20 The signals found in the microprocessor-based system for an OUT 19H,AX instruction.


Microprocessor-based system
(Port data)
Contents of register AX
Data bus (D0–D15)
(Port address)
0019H
Address bus (A0–A15)
(Port control)
$\overline{\text{IOWC}}$