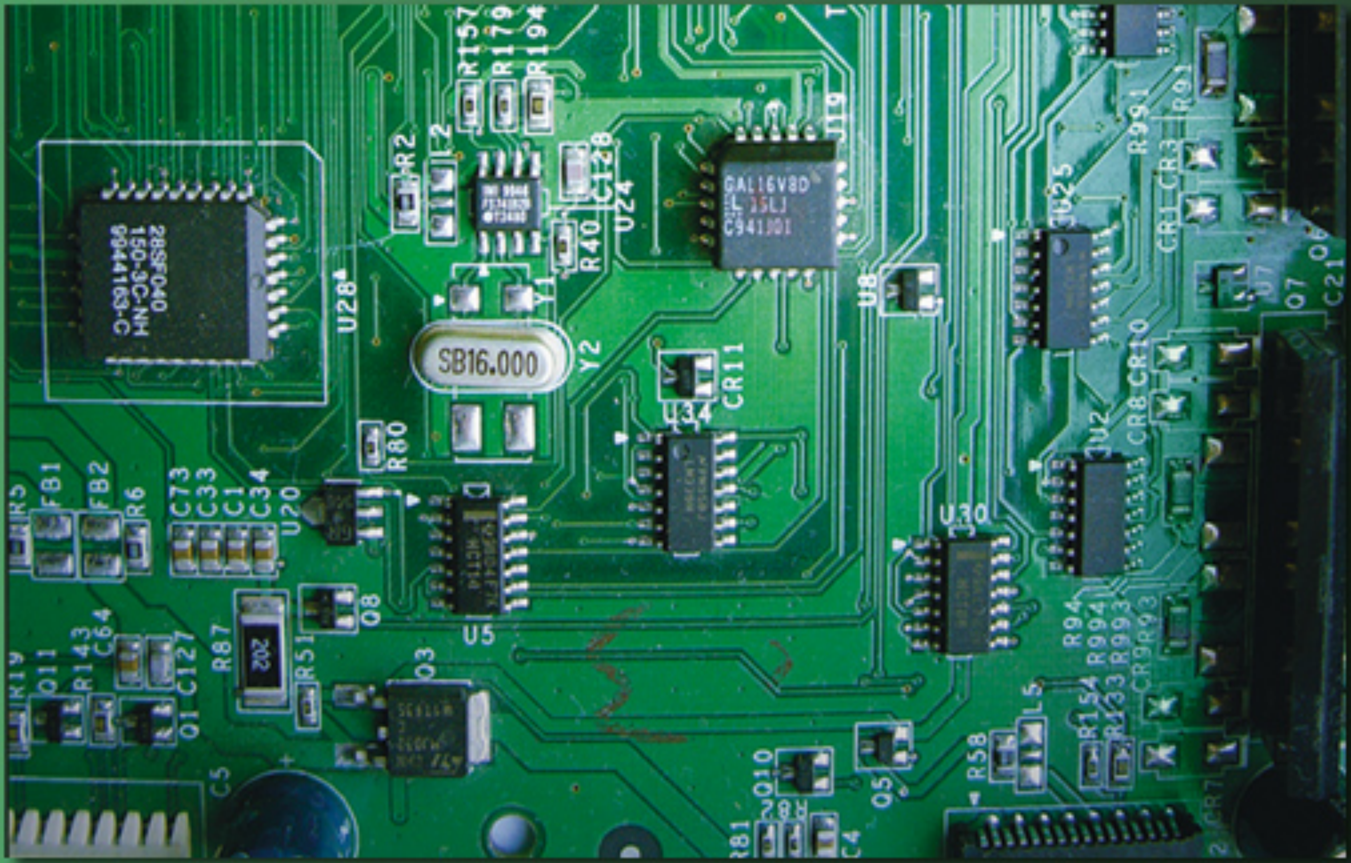
The Intel Microprocessors

8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium 4, and Core2 with 64-bit Extensions

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 4: Data Movement Instructions

Introduction

- This chapter concentrates on the data movement instructions.
- The data movement instructions include MOV, MOVSX, MOVZX, PUSH, POP, BSWAP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LFS, LGS, LSS, LAHF, SAHF.
- String instructions: MOVS, LODS, STOS, INS, and OUTS.

Chapter Objectives

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

- Explain the operation of each data movement instruction with applicable addressing modes.
- Explain the purposes of the assembly language pseudo-operations and key words such as ALIGN, ASSUME, DB, DD, DW, END, ENDS, ENDP, EQU, .MODEL, OFFSET, ORG, PROC, PTR, SEGMENT, USEI6, USE32, and USES.

Chapter Objectives *(cont.)*

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

- Select the appropriate assembly language instruction to accomplish a specific data movement task.
- Determine the symbolic opcode, source, destination, and addressing mode for a hexadecimal machine language instruction.
- Use the assembler to set up a data segment, stack segment, and code segment.

Chapter Objectives *(cont.)*

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

- Show how to set up a procedure using PROC and ENDP.
- Explain the difference between memory models and full-segment definitions for the MASM assembler.
- Use the Visual online assembler to perform data movement tasks.

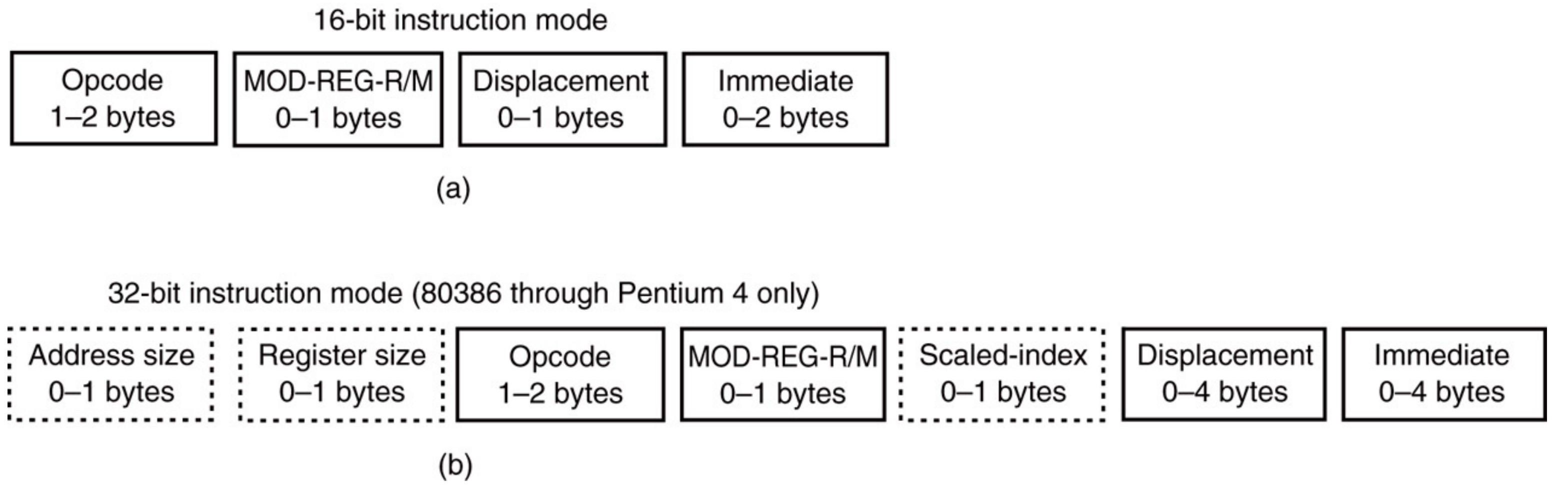
4–1 MOV Revisited

- In this chapter, the MOV instruction introduces machine language instructions available with various addressing modes and instructions.
- It may be necessary to interpret machine language programs generated by an assembler.
- Occasionally, machine language patches are made by using the DEBUG program available with DOS and Visual for Windows.

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.

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



- 80386 and above assume all instructions are 16-bit mode instructions when the machine is operated in the *real mode* (*DOS*).
- in *protected mode* (*Windows*), the upper byte of the descriptor contains the D-bit that selects either the 16- or 32-bit instruction mode

The Opcode

- Selects the operation (addition, subtraction, etc.,) performed by the microprocessor.
  - either 1 or 2 bytes long for most instructions
- Figure 4–2 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 4–2 Byte 1 of many machine language instructions, showing the position of the D- and W-bits.

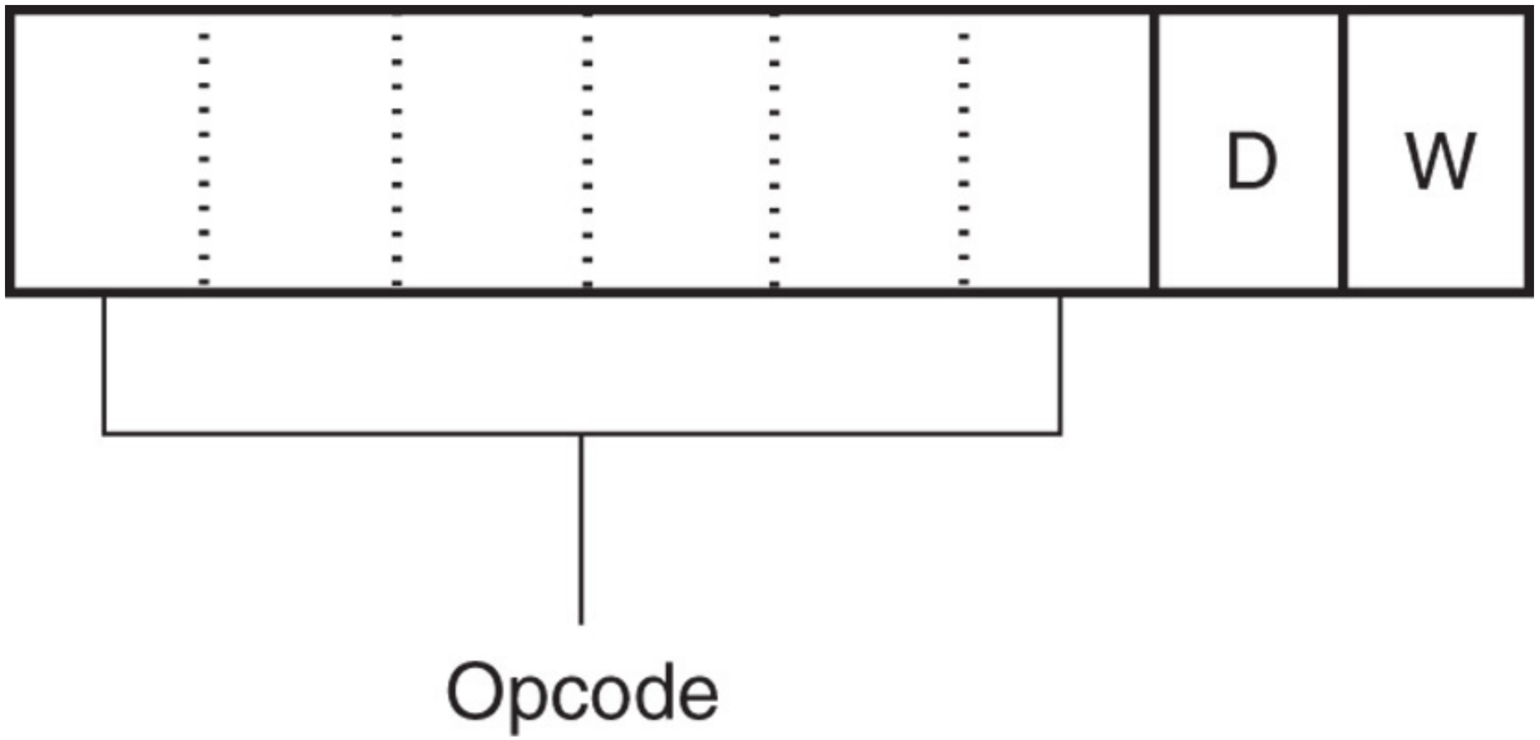
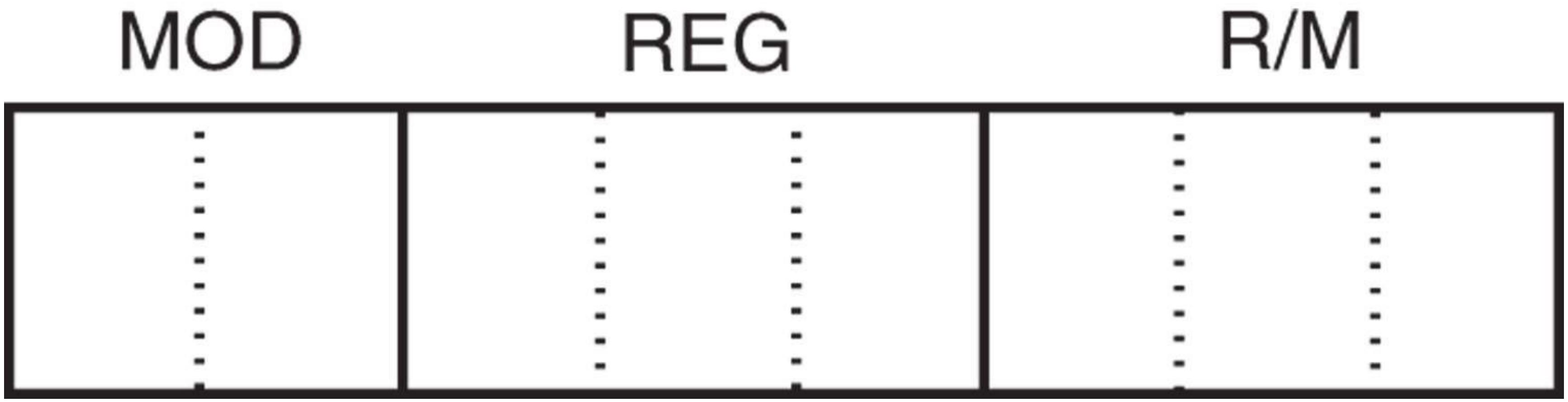


Figure 4–3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.



MOD Field

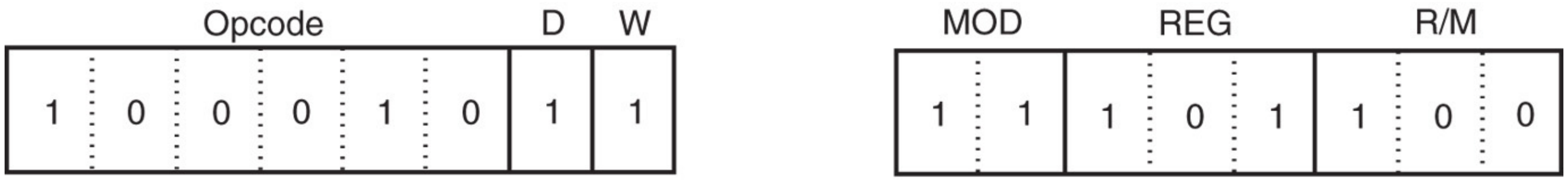
- Specifies addressing mode (MOD) and whether a displacement is present with the selected type.
  - If MOD field contains an 11, it selects the register-addressing mode
  - Register addressing specifies a register instead of a memory location, using the R/M field
- If the MOD field contains a 00, 01, 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
- Some assembler programs do not use the 8-bit displacements and in place default to all 16-bit displacements.

Register Assignments ✓

- Suppose a 2-byte instruction, 8BECH, appears in a machine language program.
  - neither a 67H (operand address-size override prefix) nor a 66H (register-size override prefix) appears as the first byte, thus the first byte is the opcode
- 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.



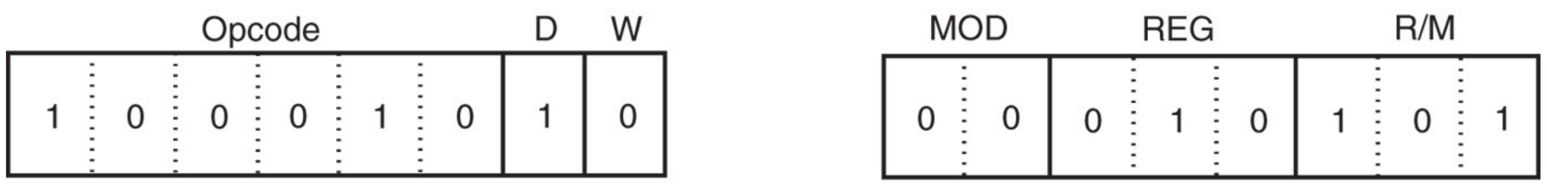
Opcode = MOV
D = Transfer to register (REG)
W = Word
MOD = R/M is a register
REG = BP
R/M = SP

- 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 (8AI5H).
- 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.



Opcode = MOV
D = Transfer to register (REG)
W = Byte
MOD = No displacement
REG = DL
R/M = DS:[DI]

- 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

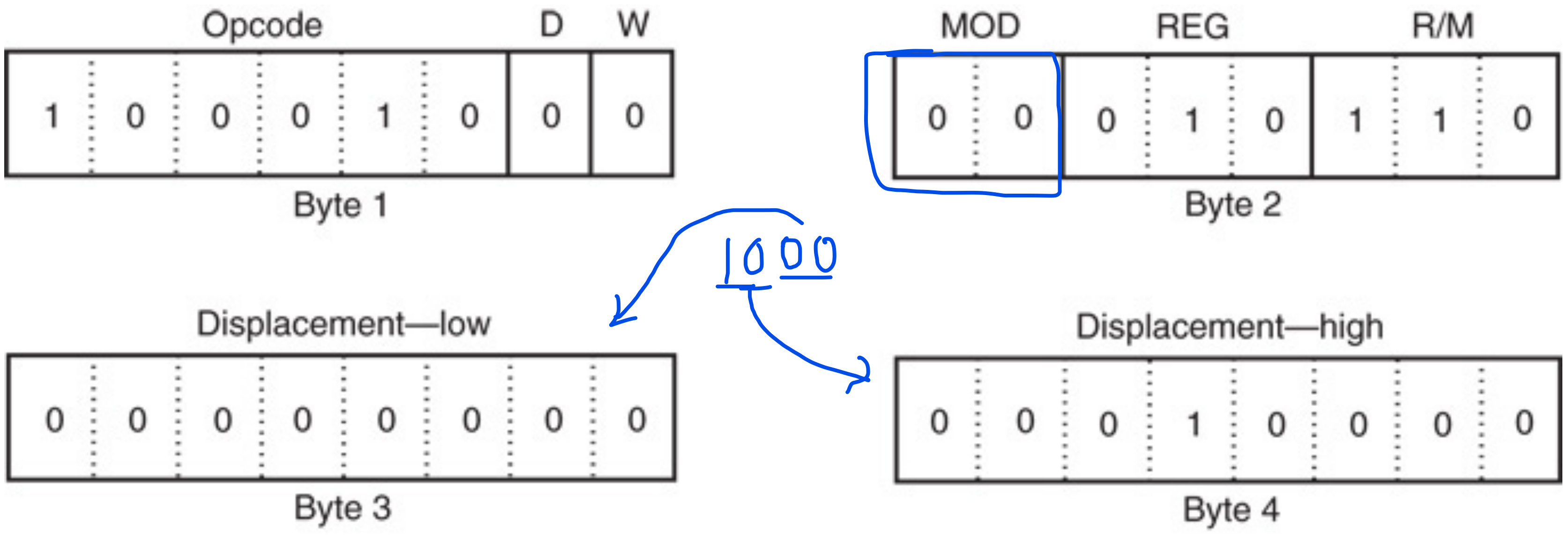
- Because the MOD field contains a 11, the R/M field also indicates a register.
- = 100(SP); therefore, this instruction moves data from SP into BP.
  - written in symbolic form as a MOV BP,SP instruction
- The assembler program keeps track of the register- and address-size prefixes and the mode of operation.

Special Addressing Mode

- A special addressing mode occurs when memory data are referenced by only the displacement mode of addressing for 16-bit instructions.
- Examples are the MOV [1000H],DL and MOV NUMB,DL instructions.
  - first instruction moves contents of register DL into data segment memory location 1000H
  - second moves register DL into symbolic data segment memory location NUMB

- When an instruction has only a displacement, MOD field is always 00; R/M field always 110.
  - You cannot actually *use* addressing mode [BP] without a displacement in machine language
- If the individual translating this symbolic instruction into machine language does not know about the special addressing mode, the instruction would incorrectly translate to a MOV [BP], DL instruction.

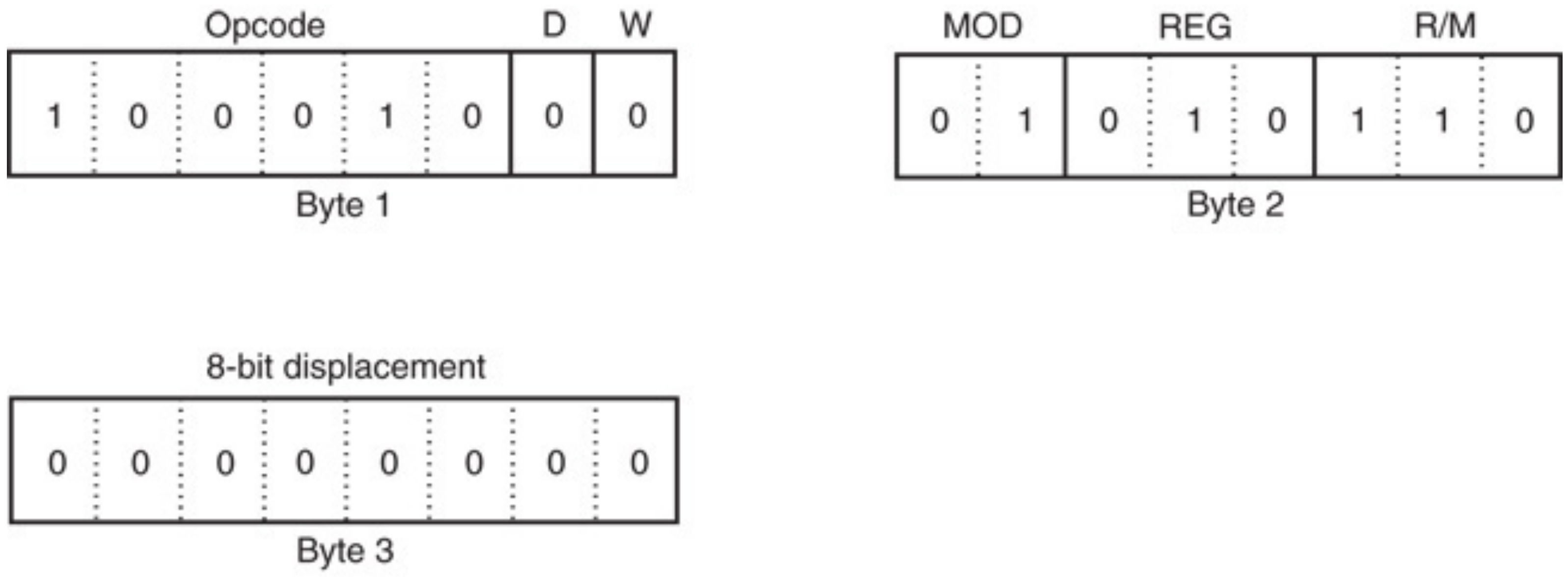
Figure 4–6 The MOV [1000H],DL instruction uses the special addressing mode.



Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 1000H

– bit pattern required to encode the MOV [1000H],DL instruction in machine language

Figure 4–7 The MOV [BP],DL instruction converted to binary machine language.



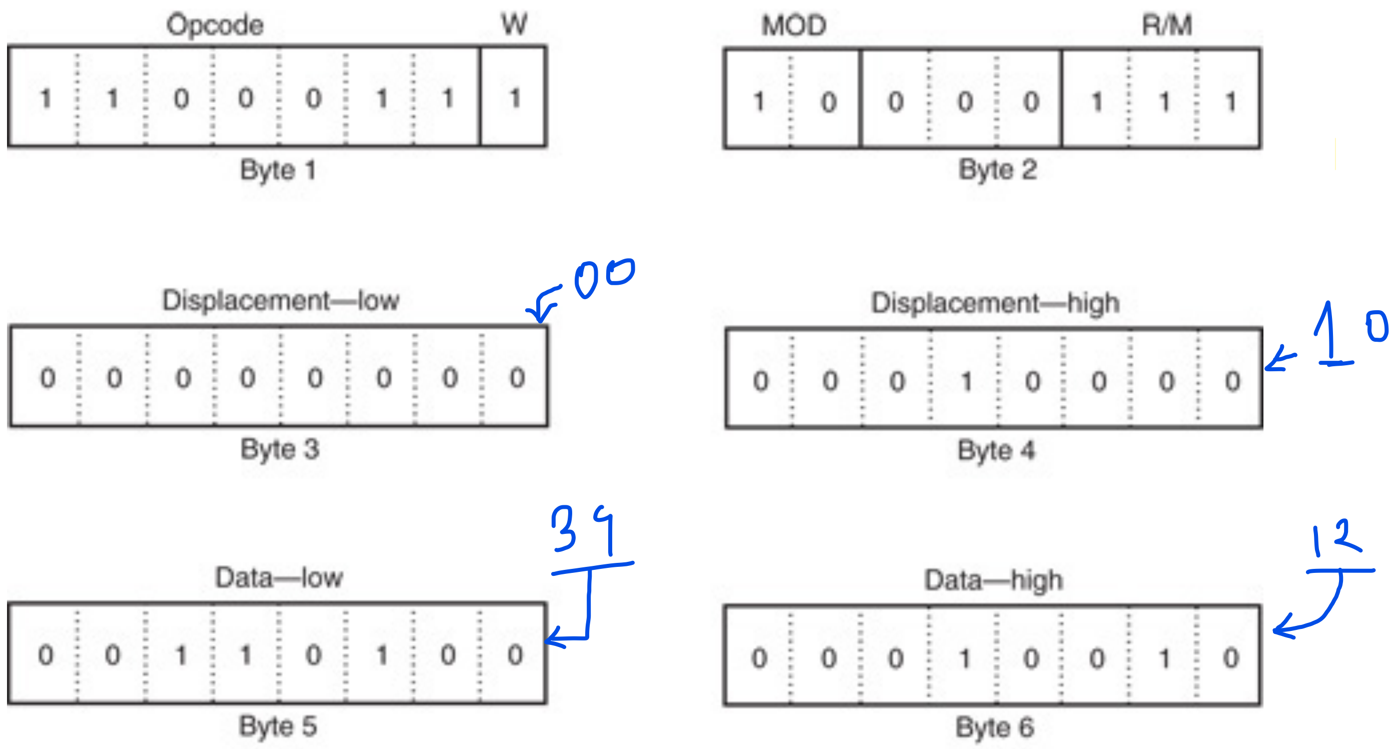
Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 00H

- actual form of the MOV [BP],DL instruction
- a 3-byte instruction with a displacement of 00H

An Immediate Instruction ✓✓

- An example of a 16-bit instruction using immediate addressing.
  - MOV WORD PTR [BX+1000H] ,1234H moves a 1234H into a word-sized memory location addressed by sum of 1000H, BX, and DS x 10H
- 6-byte instruction
  - 2 bytes for the opcode; 2 bytes are the data of 1234H; 2 bytes are the displacement of 1000H
- Figure 4–9 shows the binary bit pattern for each byte of this instruction.

Figure 4–9 A MOV WORD PTR, [BX+1000H] 1234H instruction converted to binary machine language.



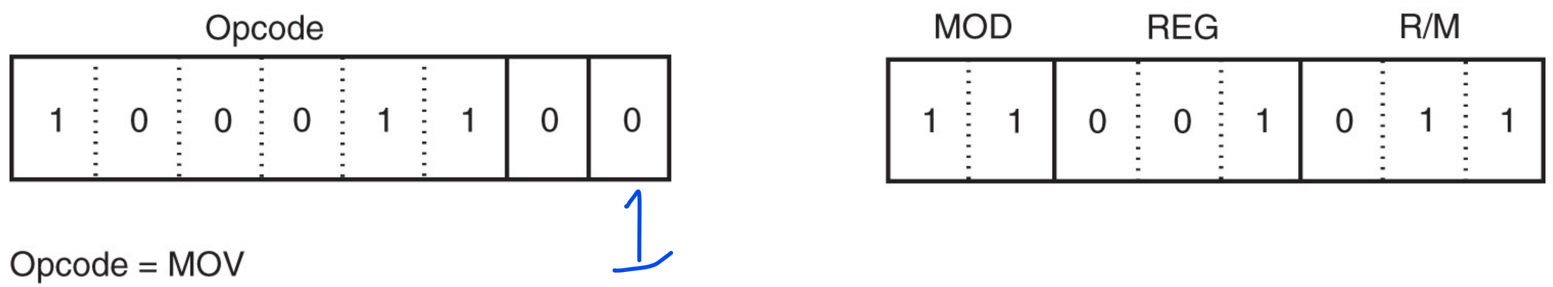
Opcode = MOV (immediate)
W = Word
MOD = 16-bit displacement
REG = 000 (not used in immediate addressing)
R/M = DS:[BX]
Displacement = 1000H
Data = 1234H

- This instruction, in symbolic form, includes WORD PTR.
  - directive indicates to the assembler that the instruction uses a word-sized memory pointer
- If the instruction moves a byte of immediate data, BYTE PTR replaces WORD PTR.
  - if a doubleword of immediate data, the DWORD PTR directive replaces BYTE PTR
- Instructions referring to memory through a pointer do not need the BYTE PTR, WORD PTR, or DWORD PTR directives.

Segment MOV Instructions

- If contents of a segment register are moved by MOV, PUSH, or POP instructions, a special bits (REG field) select the segment register.
  - the opcode for this type of MOV instruction is different for the prior MOV instructions
  - an immediate segment register MOV is not available in the instruction set
- To load a segment register with immediate data, first load another register with the data and move it to a segment register.

Figure 4–10 A MOV BX,CS instruction converted to binary machine language.



Opcode = MOV
MOD = R/M is a register
REG = CS
R/M = BX

- Figure 4–10 shows a MOV BX,CS instruction converted to binary.
- Segment registers can be moved between any 16-bit register or 16-bit memory location.

- A program written in symbolic assembly language *(assembly language)* is rarely assembled by hand into binary machine language.
- An assembler program converts symbolic assembly language into machine language.

4–2 PUSH/POP

- Important instructions that *store* and *retrieve* data from the LIFO (last-in, first-out) stack memory.
- Six forms of the PUSH and POP instructions:
  - register, memory, immediate
  - segment register, flags, all registers
- The PUSH and POP immediate & PUSHA and POPA (all registers) available 80286 - Core2.

PUSH and POP instructions

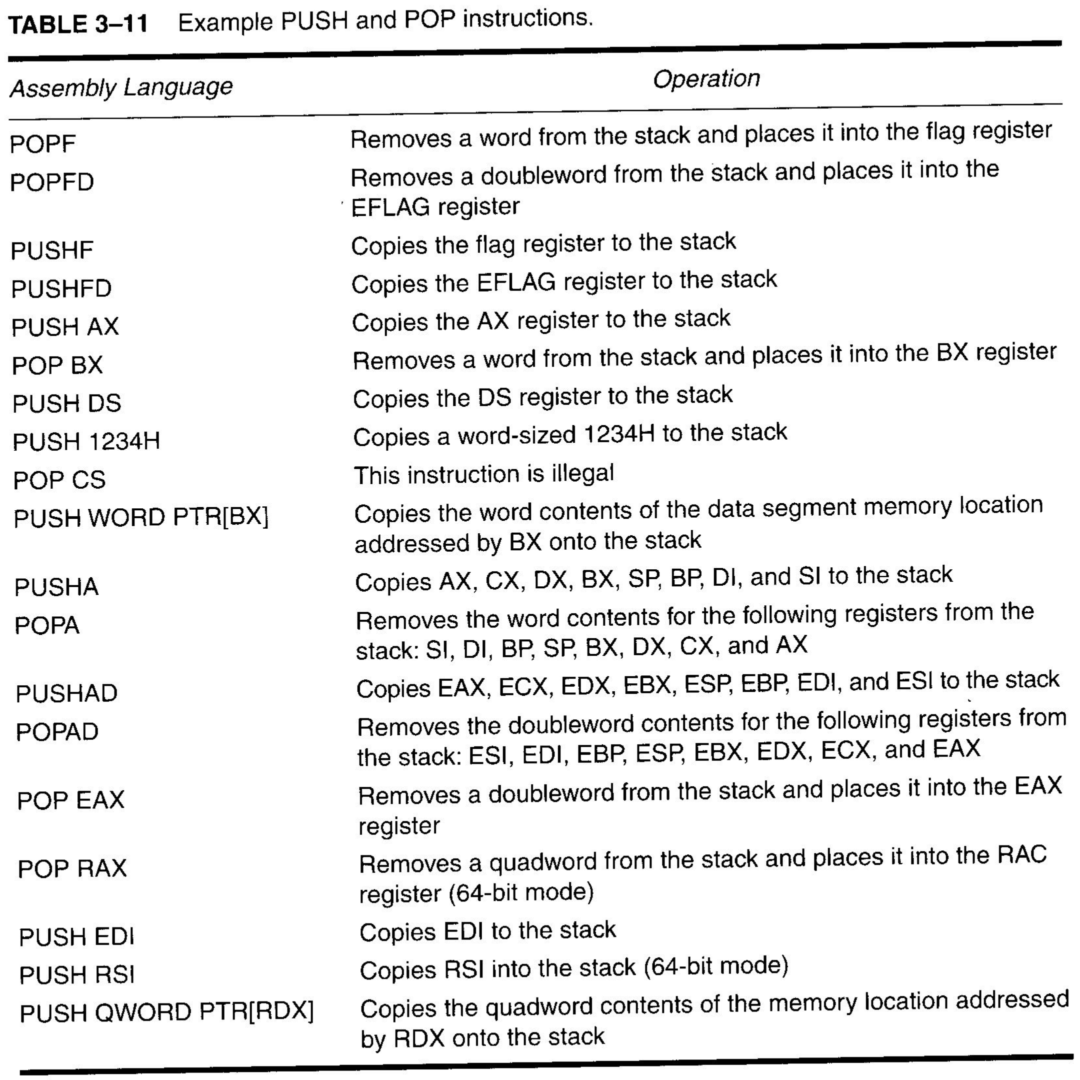


TABLE 3–11 Example PUSH and POP instructions.

Assembly Language	*Operation*
POPF	Removes a word from the stack and places it into the flag register
POPFD	Removes a doubleword from the stack and places it into the EFLAG register
PUSHF	Copies the flag register to the stack
PUSHFD	Copies the EFLAG register to the stack
PUSH AX	Copies the AX register to the stack
POP BX	Removes a word from the stack and places it into the BX register
PUSH DS	Copies the DS register to the stack
PUSH 1234H	Copies a word-sized 1234H to the stack
POP CS	This instruction is illegal
PUSH WORD PTR[BX]	Copies the word contents of the data segment memory location addressed by BX onto the stack
PUSHA	Copies AX, CX, DX, BX, SP, BP, DI, and SI to the stack
POPA	Removes the word contents for the following registers from the stack: SI, DI, BP, SP, BX, DX, CX, and AX
PUSHAD	Copies EAX, ECX, EDX, EBX, ESP, EBP, EDI, and ESI to the stack
POPAD	Removes the doubleword contents for the following registers from the stack: ESI, EDI, EBP, ESP, EBX, EDX, ECX, and EAX
POP EAX	Removes a doubleword from the stack and places it into the EAX register
POP RAX	Removes a quadword from the stack and places it into the RAC register (64-bit mode)
PUSH EDI	Copies EDI to the stack
PUSH RSI	Copies RSI into the stack (64-bit mode)
PUSH QWORD PTR[RDX]	Copies the quadword contents of the memory location addressed by RDX onto the stack

- Register addressing allows contents of any 16-bit register to transfer to & from the stack.
- Memory-addressing PUSH and POP instructions store contents of a 16- or 32 bit memory location on the stack or stack data into a memory location.
- Immediate addressing allows immediate data to be pushed onto the stack, but not popped off the stack.

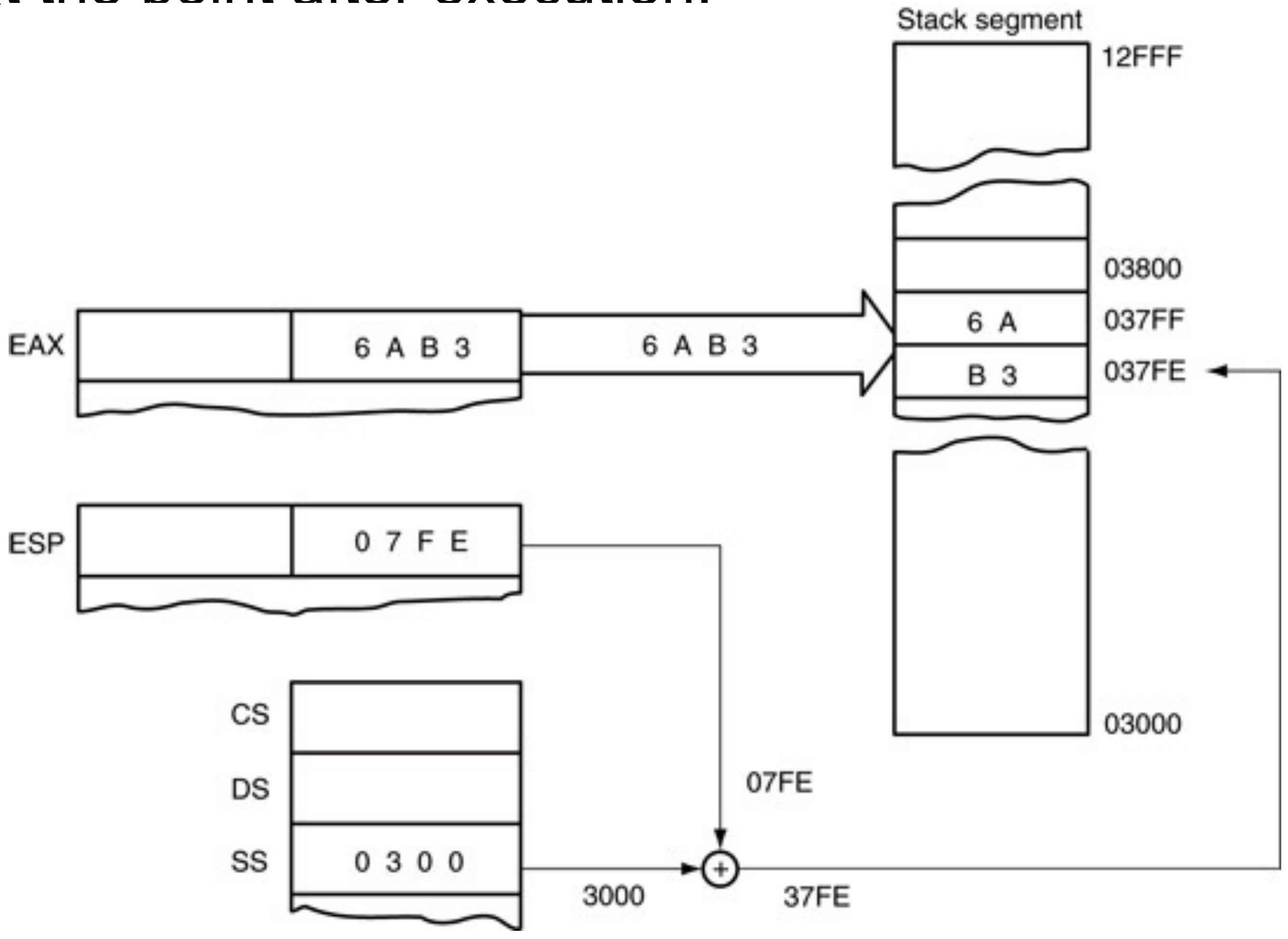
- Segment register addressing allows contents of any segment register to be pushed onto the stack or removed from the stack.
  - ES may be pushed, but data from the stack may never be popped into ES
- The flags may be pushed or popped from that stack.
  - contents of all registers may be pushed or popped

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.

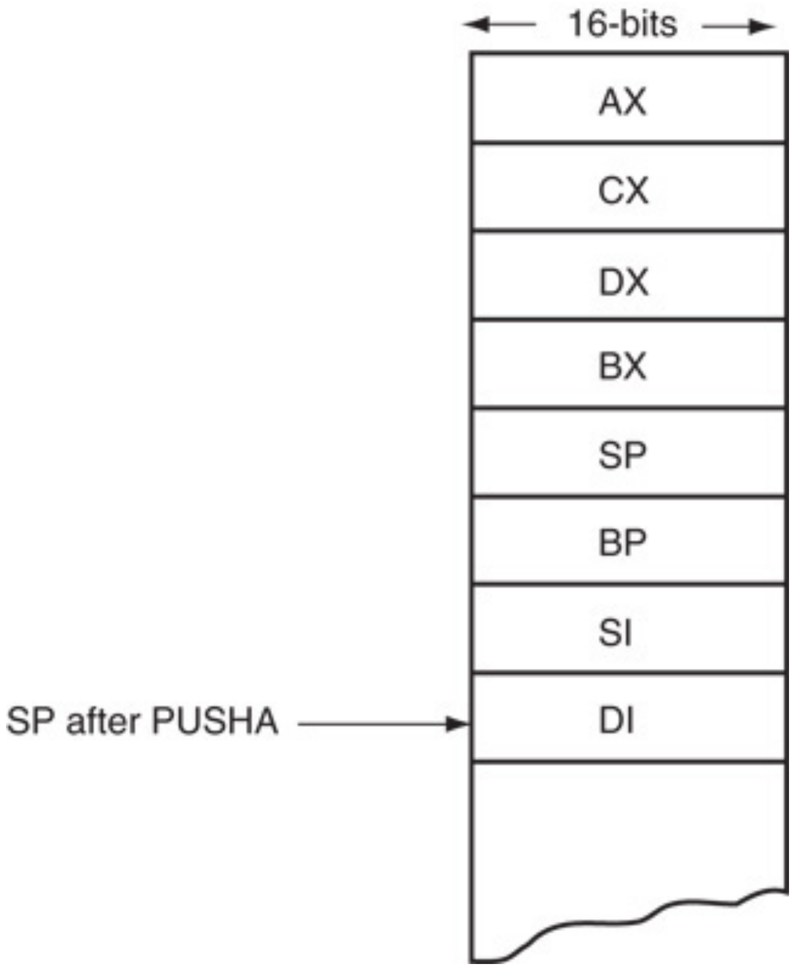
- PUSHF (**push flags**) instruction copies the contents of the flag register to the stack.
- PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set in 80386 - Pentium 4.
  - PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4

Figure 4–13 The effect of the PUSH AX instruction on ESP and stack memory locations 37FFH and 37FEH. This instruction is shown at the point after execution.



- PUSHA instruction pushes all the internal 16-bit registers onto the stack, illustrated in 4–14.
  - requires 16 bytes of stack memory space to store all eight 16-bit registers
- After all registers are pushed, the contents of the SP register are decremented by 16.
- PUSHA is very useful when the entire register set of 80286 and above must be saved.
- PUSHAD instruction places 32-bit register set on the stack in 80386 - Core2.
  - PUSHAD requires 32 bytes of stack storage

Figure 4–14 The operation of the PUSHA instruction, showing the location and order of stack data.

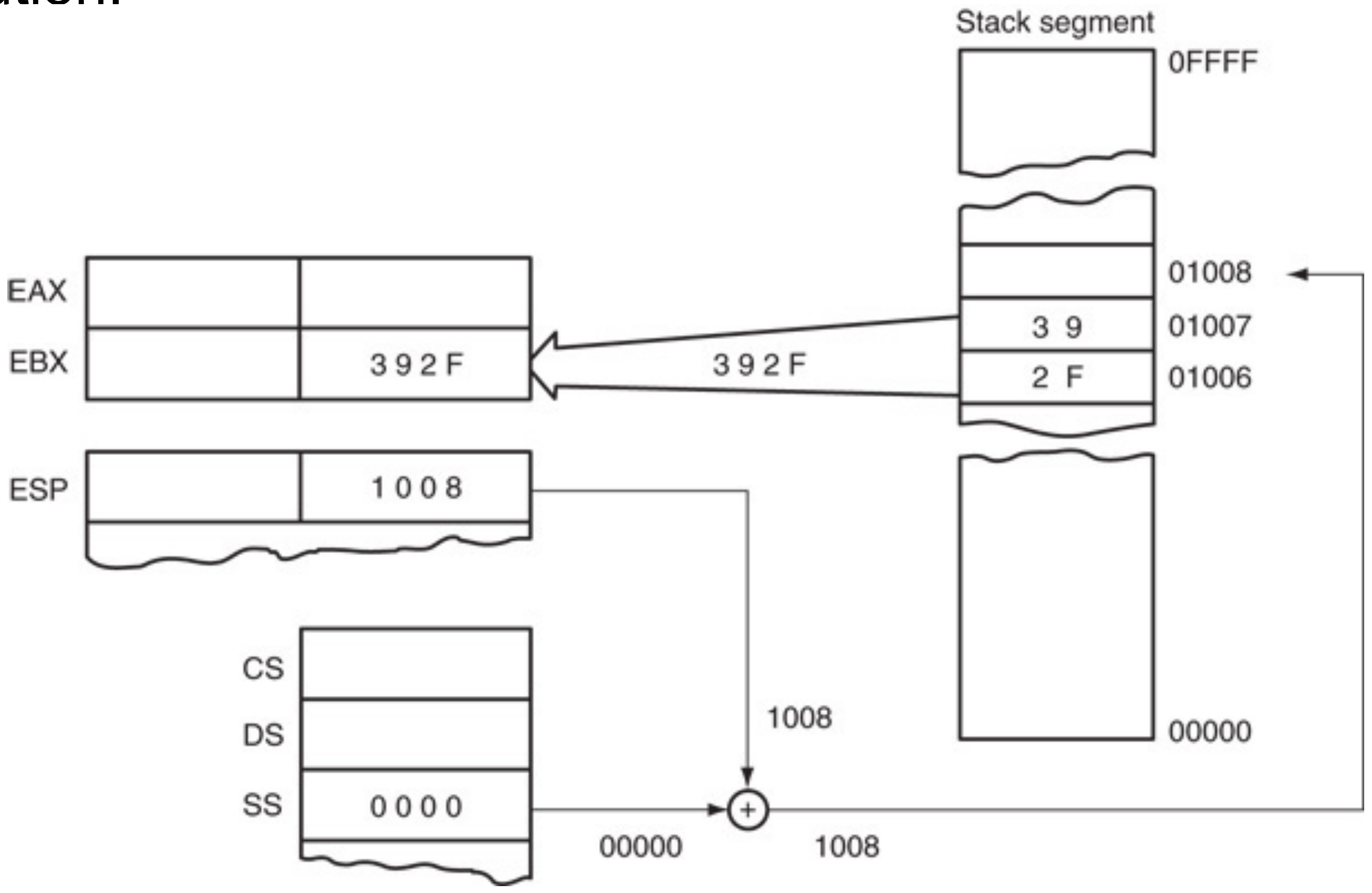


POP

- 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
- Figure 4–15 shows how the POP BX instruction removes data from the stack and places them into register BX.

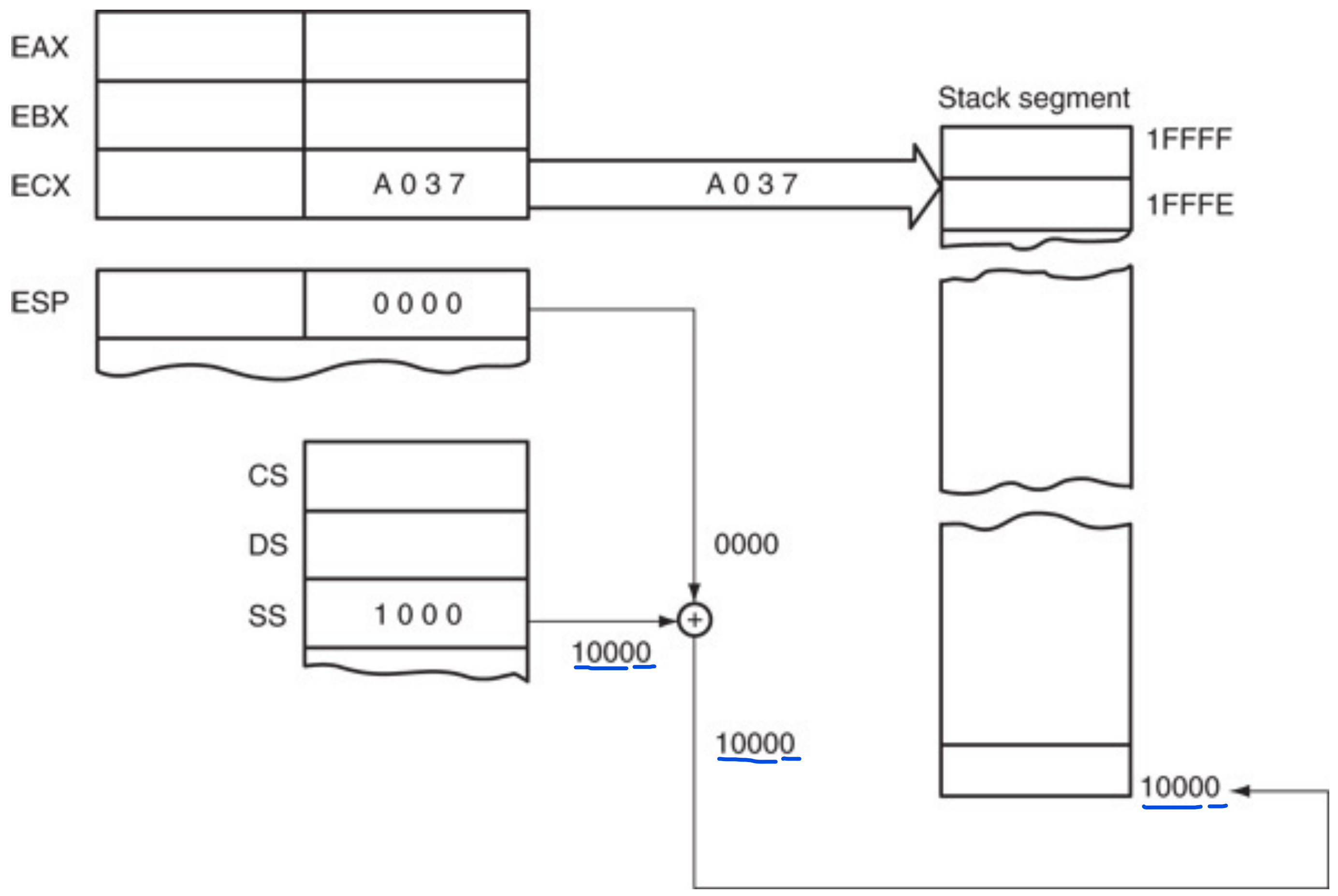
Figure 4–15 The POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.



Initializing the Stack

- When the stack area is initialized, load both the stack segment (SS) register and the stack pointer (SP) register.
- Figure 4–16 shows how this value causes data to be pushed onto the top of the stack segment with a PUSH CX instruction.
- All segments are cyclic in nature
  - the top location of a segment is contiguous with the bottom location of the segment

Figure 4–16 The PUSH CX instruction, showing the cyclical nature of the stack segment. This instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.



- Assembly language stack segment setup:
  - first statement identifies start of the segment
  - last statement identifies end of the stack segment
- Assembler and linker programs place correct stack segment address in SS and the length of the segment (top of the stack) into SP.
- There is no need to load these registers in your program.
  - unless you wish to change the initial values for some reason

- If the stack is not specified, a warning will appear when the program is linked.
- Memory section is located in the **program segment prefix** (PSP), appended to the beginning of each program file.
- If you use more memory for the stack, you will erase information in the PSP.
  - information critical to the operation of your program and the computer
- Error often causes the program to crash.

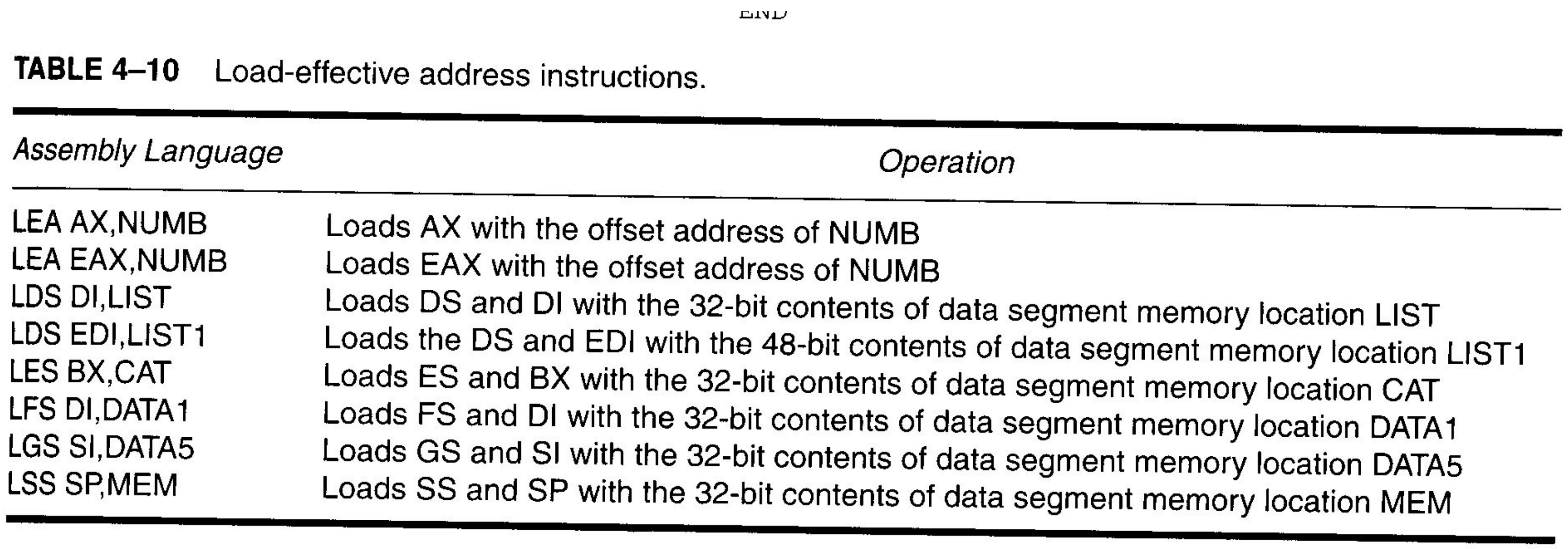
LEA

- Loads a 16- or 32-bit register with the offset address of the data specified by the operand.
- Earlier examples presented by using the OFFSET directive.
  - OFFSET performs same function as LEA instruction if the operand is a displacement
- LEA and MOV with OFFSET instructions are both the same length (3 bytes).

4–3 LOAD EFFECTIVE ADDRESS

- LEA instruction loads any 16-bit register with the offset address
  - determined by the addressing mode selected
- LDS and LES load a 16-bit register with offset address retrieved from a memory location
  - then load either DS or ES with a segment address retrieved from memory
- In 80386 and above, LFS, LGS, and LSS are added to the instruction set.

Load-effective address instructions



END

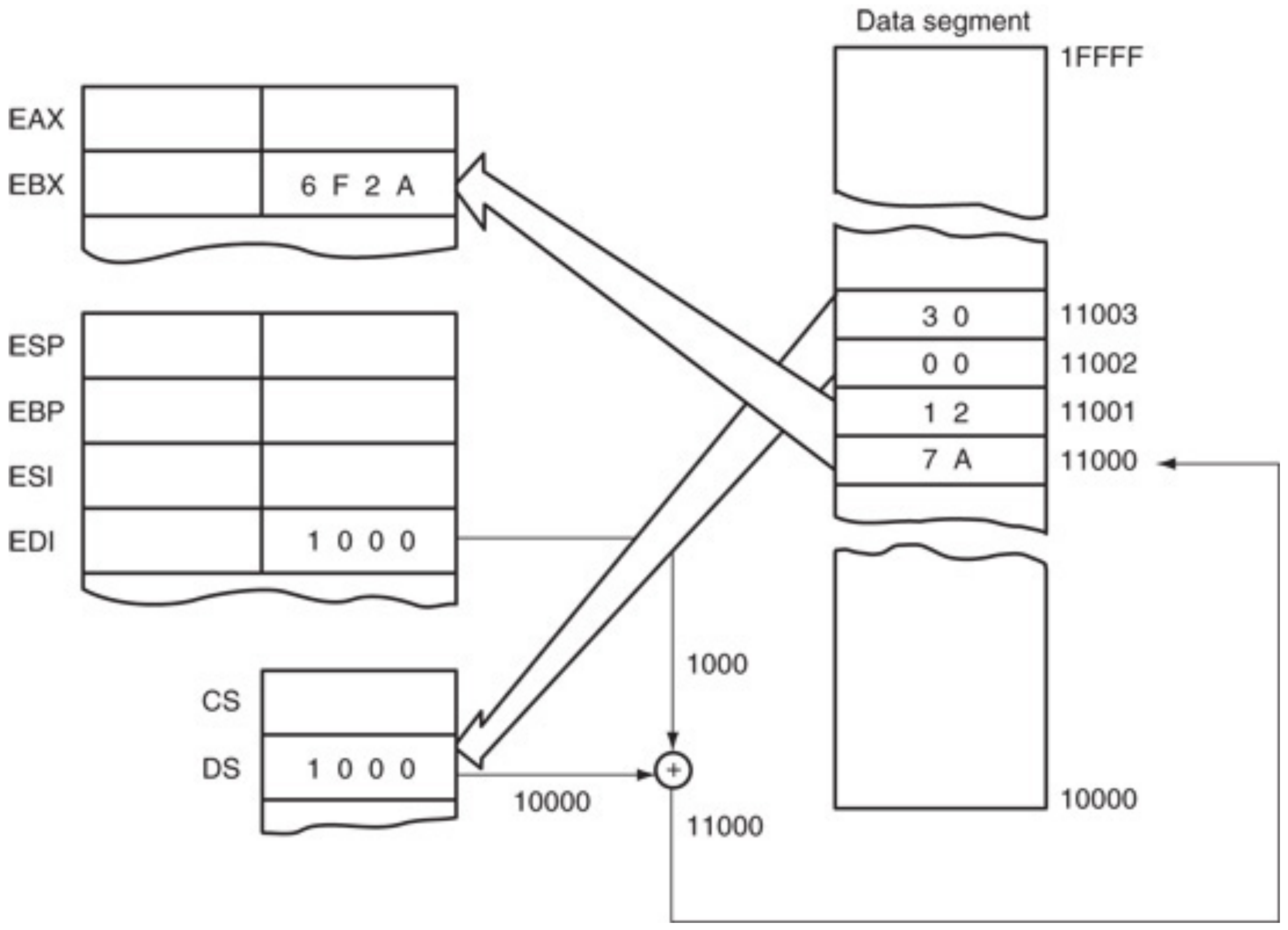
TABLE 4–10 Load-effective address instructions.

Assembly Language	*Operation*
LEA AX,NUMB	Loads AX with the offset address of NUMB
LEA EAX,NUMB	Loads EAX with the offset address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST1	Loads the DS and EDI with the 48-bit contents of data segment memory location LIST1
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT
LFS DI,DATA1	Loads FS and DI with the 32-bit contents of data segment memory location DATA1
LGS SI,DATA5	Loads GS and SI with the 32-bit contents of data segment memory location DATA5
LSS SP,MEM	Loads SS and SP with the 32-bit contents of data segment memory location MEM

LDS, LES, LFS, LGS, and LSS

- Load any 16- or 32-bit register with an offset address, and the DS, ES, FS, GS, or SS segment register with a segment address.
  - instructions use any memory-addressing modes to access a 32-bit or 48-bit memory section that contain both segment and offset address
- Figure 4–17 illustrates an example LDS BX,[DI] instruction.

Figure 4–17 The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.



- This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers.
- LDS, LES, LFS, LGS, and LSS instructions obtain a new far address from memory.
  - offset address appears first, followed by the segment address
- This format is used for storing all 32-bit memory addresses.

4–4 STRING DATA TRANSFERS

- Five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS.
- Each allows data transfers as a single byte, word, or doubleword.
- 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 the auto-increment or the 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

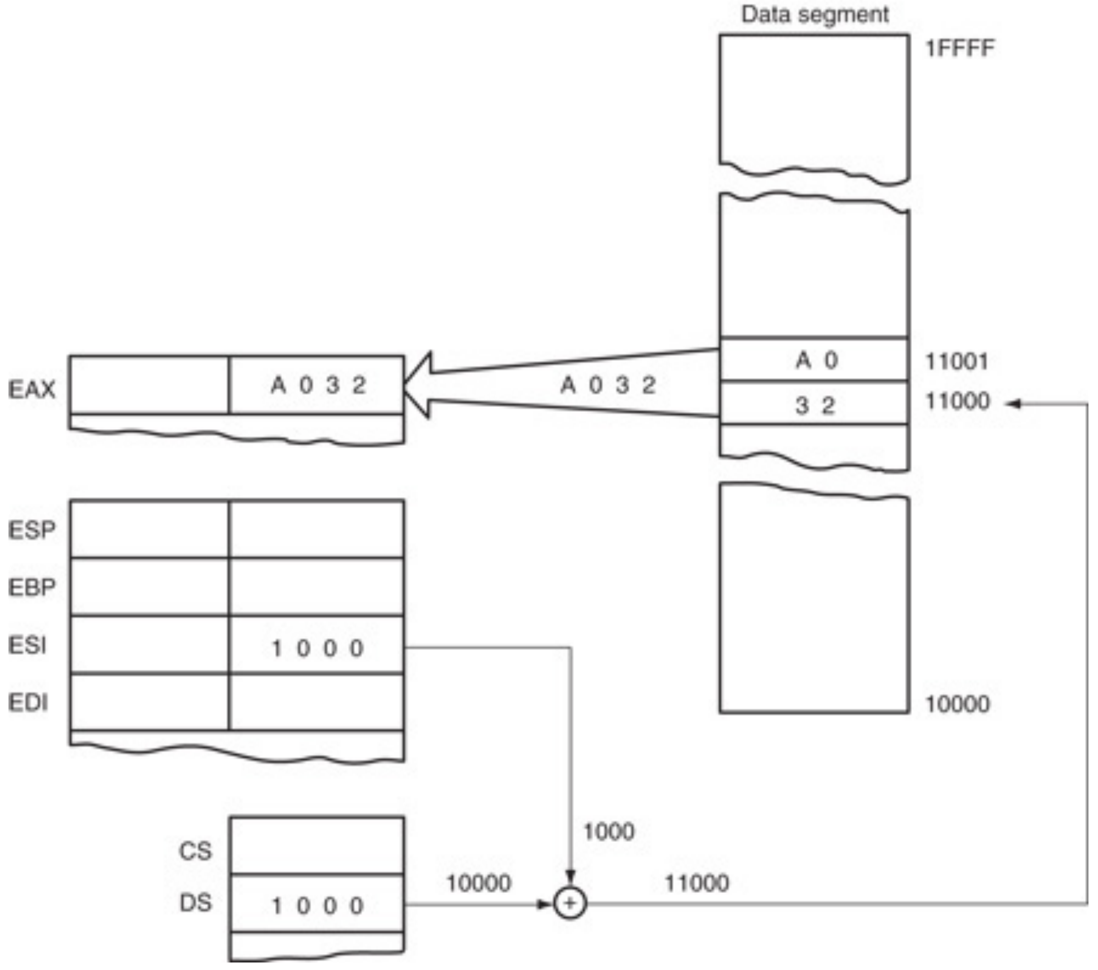
DI and SI

- 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
- Operating in 32-bit mode EDI and ESI registers are used in place of DI and SI.
  - this allows string using any memory location in the entire 4G-byte protected mode address space

LODS

- Loads AL, AX, or EAX with data at segment offset address indexed by the SI register.
- A 1 is added to or subtracted from SI for a byte-sized LODS
- A 2 is added or subtracted for a word-sized LODS.
- A 4 is added or subtracted for a doubleword-sized LODS.
- Figure 4–18 shows the LODSW instruction.

Figure 4–18 The operation of the LODSW instruction if DS=1000H, D=0,11000H, 11001H = A0. This instruction is shown after AX is loaded from memory, but before SI increments by 2.



STOS

- Stores AL, AX, or EAX at the extra segment memory location addressed by the DI register.
- STOSB (**stores a byte**) stores the byte in AL at the extra segment memory location addressed by DI.
- STOSW (**stores a word**) stores AX in the memory location addressed by DI.
- After the byte (AL), word (AX), or doubleword (EAX) is stored, contents of DI increment or decrement.

MOVS

- Transfers a byte, word, or doubleword a data segment addressed by SI to extra segment location addressed by SI.
  - pointers are incremented or decremented, as dictated by the direction flag
- Only the source operand (SI), located in the data segment may be overridden so another segment may be used.
- The destination operand (DI) must always be located in the extra segment.

INS

- Transfers a byte, word, or doubleword of data from an I/O device into the extra segment memory location addressed by the DI register.
  - I/O address is contained in the DX register
- Useful for inputting a block of data from an external I/O device directly into the memory.
- One application transfers data from a disk drive to memory.
  - disk drives are often considered and interfaced as I/O devices in a computer system

- Three basic forms of the INS.
- INSB inputs data from an 8-bit I/O device and stores it in a memory location indexed by SI.
- INSW instruction inputs 16-bit I/O data and stores it in a word-sized memory location.
- INSD instruction inputs a doubleword.
- These instructions can be repeated using the REP prefix
  - allows an entire block of input data to be stored in the memory from an I/O device

OUTS

- Transfers a byte, word, or doubleword of data from the data segment memory location address by SI to an I/O device.
  - I/O device addressed by the DX register as with the INS instruction
- In the 64-bit mode for Pentium 4 and Core2, there is no 64-bit output
  - but the address in RSI is 64 bits wide

4–5 MISCELLANEOUS DATA TRANSFER INSTRUCTIONS

- Used in programs, data transfer instructions detailed in this section are XCHG, LAHF, SAHF, XLAT, IN, OUT, BSWAP, MOVSX, MOVZX, and CMOV.

XCHG

- Exchanges contents of a register with any other register or memory location.
  - cannot exchange segment registers or memory-to-memory data
- Exchanges are byte-, word-, or doubleword and use any addressing mode except immediate addressing.
- XCHG using the 16-bit AX register with another 16-bit register, is most efficient exchange.

LAHF and SAHF

- Seldom used bridge instructions.
- LAHF instruction transfers the rightmost 8 bits of the flag register into the AH register.
- SAHF instruction transfers the AH register into the rightmost 8 bits of the flag register.
- SAHF instruction may find some application with the numeric coprocessor.
- As legacy instructions, they do not function in the 64-bit mode and are invalid instructions.

IN and OUT

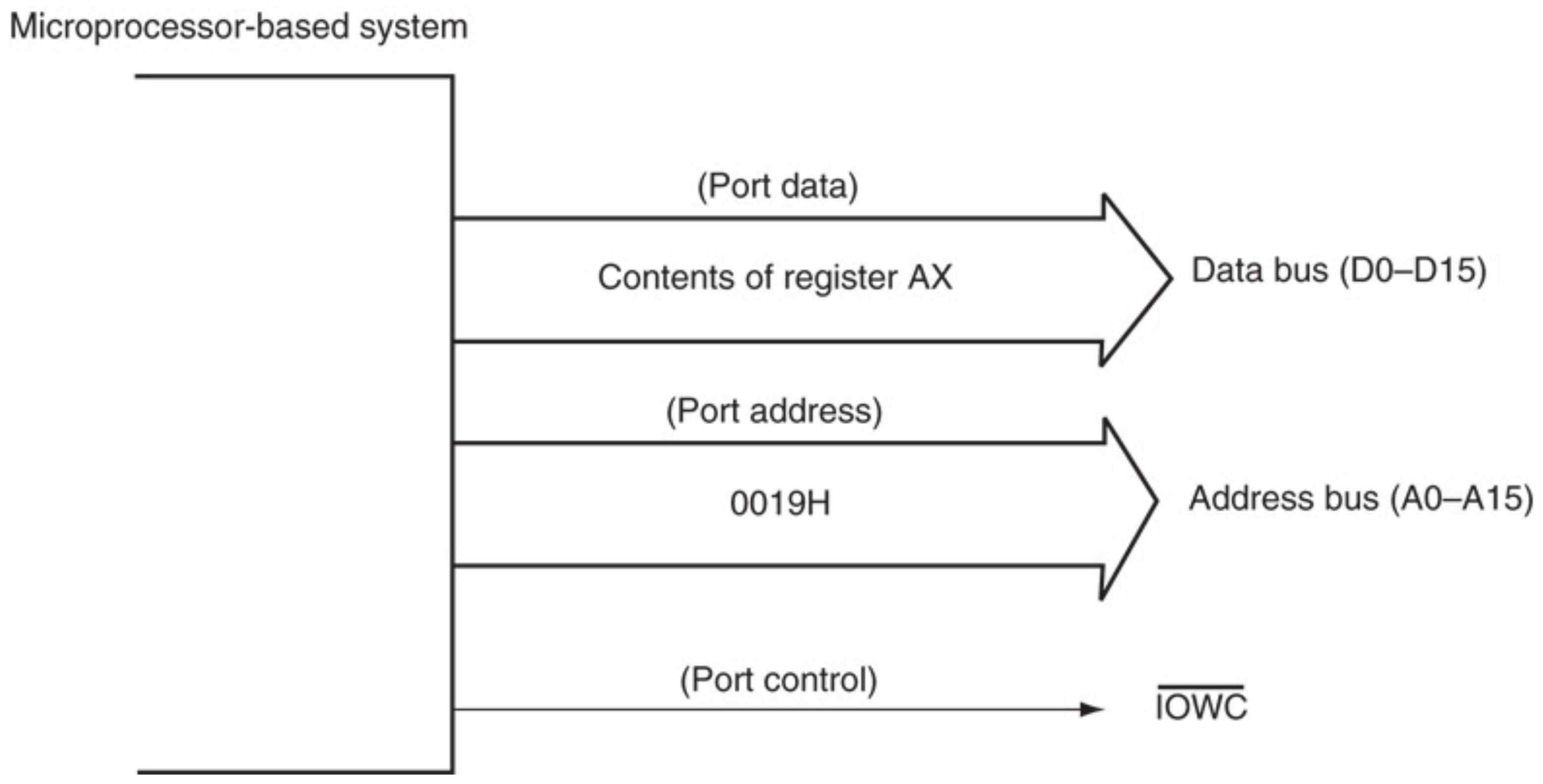
not
Imp

- 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
- Only the 80386 and above contain EAX

- Often, instructions are stored in ROM.
  - a fixed-port instruction stored in ROM has its port number permanently fixed because of the nature of read-only memory
- A fixed-port address stored in RAM can be modified, but such a modification does not conform to good programming practices.
- The port address appears on the address bus during an I/O operation.

- 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.



Storing Data in a Memory Segment

- DB (**define byte**), DW (**define word**), and DD (**define doubleword**) are most often used with MASM to define and store memory data.
- If a numeric coprocessor executes software in the system, the DQ (**define quadword**) and DT (**define ten bytes**) directives are also common.
- These directives label a memory location with a symbolic name and indicate its size.

- Memory is reserved for use in the future by using a question mark (?) as an operand for a DB, DW, or DD directive.
  - when ? is used in place of a numeric or ASCII value, the assembler sets aside a location and does not initialize it to any specific value
- It is important that word-sized data are placed at word boundaries and doubleword-sized data are placed at doubleword boundaries.
  - if not, the microprocessor spends additional time accessing these data types

ASSUME, EQU, and ORG

- Equate directive (EQU) equates a numeric, ASCII, or label to another label.
  - equates make a program clearer and simplify debugging
- The THIS directive always appears as THIS BYTE, THIS WORD, THIS DWORD, or THIS QWORD.
- The assembler can only assign either a byte, word, or doubleword address to a label.

- The ORG (origin) statement changes the starting offset address of the data in the data segment to location 300H.
- At times, the origin of data or the code must be assigned to an absolute offset address with the ORG statement.
- ASSUME tells the assembler what names have been chosen for the code, data, extra, and stack segments.

PROC and ENDP

- Indicate start and end of a procedure (subroutine).
  - they *force structure* because the procedure is clearly defined
- If structure is to be violated for whatever reason, use the CALLF, CALLN, RETF, and RETN instructions.
- Both the PROC and ENDP directives require a label to indicate the name of the procedure.

- The PROC directive, which indicates the start of a procedure, must also be followed with a NEAR or FAR.
  - A NEAR procedure is one that resides in the same code segment as the program, often considered to be *local*
  - A FAR procedure may reside at any location in the memory system, considered *global*
- The term *global* denotes a procedure that can be used by any program.
- *Local* defines a procedure that is only used by the current program.

SUMMARY

- Data movement instructions transfer data between registers, a register and mem-ory, a register and the stack, memory and the stack, the accumulator and I/O, and the flags and the stack.
- Memory-to-memory transfers are allowed only with the MOVS instruction.

SUMMARY (*cont.*)

- Data movement instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LOS, LES, LSS, LGS, LFS, LAHF, SAHF, and the following string instruc-tions: LODS, STOS, MOVS, INS, and OUTS.
- The first byte of an instruction contains the opcode, which specifies the operation performed by the microprocessor.
- The opcode may be preceded by one or more override prefixes.

SUMMARY (*cont.*)

- The D-bit, located in many instructions, selects the direction of data flow.
- The W-bit, found in most instructions, selects the size of the data transfer.
- MOD selects the addressing mode of operation for a machine language instruc-tion's R/M field.
- A 3-bit binary register code specifies the REG and R/M fields when the MOD = 11.

SUMMARY *(cont.)*

- The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL.
- The l6-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI.
- The 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
- To access the 64-bit registers, a new prefix is added called the REX prefix that contains a fourth bit.

SUMMARY *(cont.)*

- By default, all memory-addressing modes address data in the data segment unless BP or EBP addresses memory.
- The BP or EBP register addresses data in the stack segment.
- The segment registers are addressed only by the MOV, PUSH, or POP instruc-tions.
- The instruction may transfer a segment register to a 16-bit register, or vice versa.

SUMMARY (*cont.*)

- The 80386 through the Pentium 4 include two additional segment registers, FS & GS.
- Data are transferred between a register or a memory location and the stack by the PUSH and POP instructions.
- Variations of these instructions allow immediate data to be pushed onto the stack, the flags to be transferred between the stack; all 16-bit registers can transferr between the stack and registers.

SUMMARY (*cont.*)

- Opcodes that transfer data between the stack and the flags are PUSHF and POPF.
- Opcodes that transfer all the 16-bit registers between the stack and the registers are PUSHA and POPA.
- In 80386 and above, PUSHFD and POPFD transfer the contents of the EFLAGS between the microprocessor and the stack, and PUSHAD and POPAD transfer all the 32-bit registers.

SUMMARY (*cont.*)

- The PUSHA and POPA in-structions are invalid in the 64-bit mode.
- LEA, LDS, and LES instructions load a register or registers with an effective ad-dress.
- The LEA instruction loads any 16-bit register with an effective address; LDS and LES load any 16-bit register and either DS or ES with the effective address.

SUMMARY (*cont.*)

- In 80386 and above, additional instructions include LFS, LGS, and LSS, which load a 16-bit register and FS, GS, or SS.
- String data transfer instructions use either or both DI and SI to address memory. .
- The DI offset address is located in the extra segment, and the SI offset address is located in the data segment.
- If 80386-Core2 operates in protected mode, ESI & EDI are used with string instructions.

SUMMARY (*cont.*)

- The direction flag (D) chooses the auto-increment or auto-decrement mode of op-eration for DI and SI for string instructions.
- To clear D to 0, use the CLD instruction to select the auto-increment mode; to set D to 1, use the STD instruction to select the auto-decrement mode.
- Either/both DI and SI increment/decrement by 1 for a byte operation, by 2 for a word operation, and 4 for doubleword operation.

SUMMARY (*cont.*)

- LODS loads AL, AX, or EAX with data from the memory location addressed by SI; STOS stores AL, AX, or EAX in the memory location addressed by DI; and MOVS transfers a byte, a word, or a doubleword from the memory location addressed by SI into the location addressed by DI.

SUMMARY (*cont.*)

- INS inputs data from an I/O device addressed by DX and stores it in the memory location addressed by DI.
- The OUTS instruction outputs the contents of the memory location addressed by SI and sends it to the I/O device addressed by DX.

SUMMARY (*cont.*)

- The REP prefix may be attached to any string instruction to repeat it.
- The REP prefix repeats the string instruction the number of times found in register CX.
- Arithmetic and logic operators can be used in assembly language.
- An example is MOV AX,34*3, which loads AX with 102.

SUMMARY (*cont.*)

- Translate (XLAT) converts the data in AL into a number stored at the memory loca-tion addressed by BX plus AL.
- IN and OUT transfer data between AL, AX, or EAX and an external I/O device.
- The address of the I/O device is either stored with the instruction (fixed-port addressing) or in register DX (variable-port addressing).

SUMMARY (*cont.*)

- The Pentium Pro-Core2 contain a new instruction called CMOV, or conditional move.
- This instruction only performs the move if the condition is true.
- The segment override prefix selects a different segment register for a memory lo-cation than the default segment.

SUMMARY (*cont.*)

- Assembler directives DB (define byte), DW (define word), DD (define doubleword), and DUP (duplicate) store data in the memory system.
- The EQU (equate) directive allows data or labels to be equated to labels.
- The SEGMENT directive identifies the start of a memory segment and ENDS iden-tifies the end of a segment when full-segment definitions are in use.

SUMMARY (*cont.*)

- The ASSUME directive tells the assembler what segment names you have as-signed to CS, DS, ES, and SS when full-segment definitions are in effect.
- In the 80386 and above, ASSUME also indicates the segment name for FS and GS.
- The PROC and ENDP directives indicate the start and end of a procedure.

SUMMARY (*cont.*)

- The assembler assumes that software is being developed for the 8086/8088 mi-croprocessor unless the .286, .386, .486, .586, or .686 directive is used to select one of these other microprocessors.
- This directive follows the .MODEL statement to use the 16-bit instruction mode and precedes it for the 32-bit instruction mode.

SUMMARY

- Memory models can be used to shorten the program slightly, but they can cause problems for larger programs.
- Also be aware that memory models are not compatible with all assembler programs.