

---

# **Assembly Language and Microcomputer Interface**

## **Chapter 4: Data Movement Instructions**

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology  
Zhejiang University

# 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.
- Data movement instructions do not affect flags.

# 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`, `USE16`, `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.

# 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.

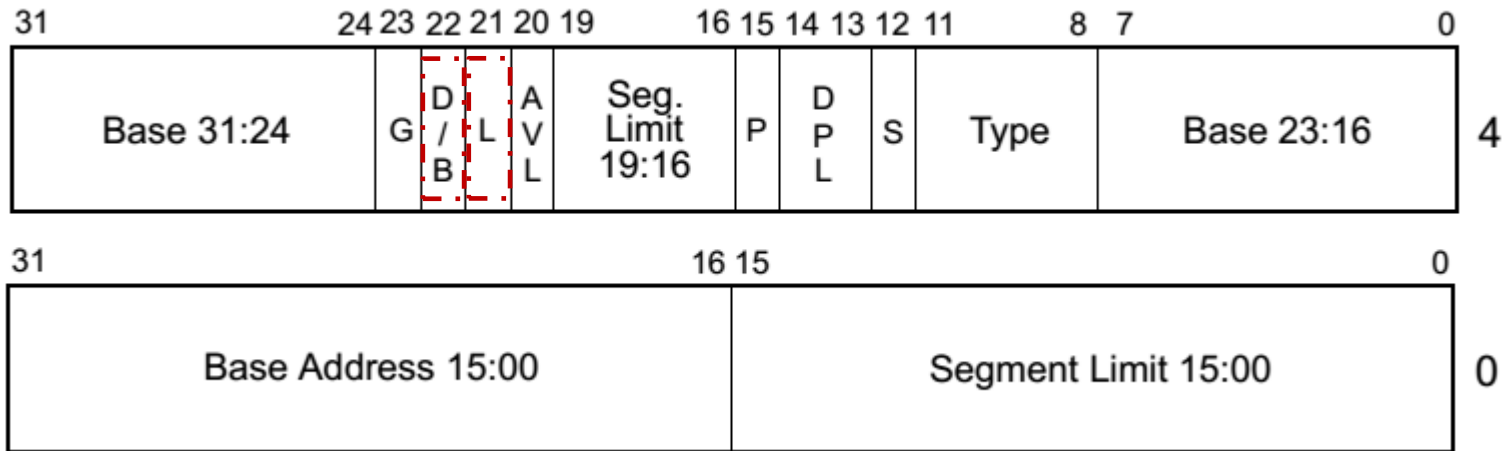
# Machine Language

- Native binary code microprocessor uses as its instructions to control its operation.
  - instructions vary in length from 1 to 15 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.

# Operation Mode

- There are three operation modes with following default address and operand size:
  - 16-bit modes (real, vm86, protected): default address and operand-size are 16-bit
  - 32-bit protected mode (protected): default address and operand-size are 32-bit
  - 64-bit mode: default address size is 64-bit, default operand-size is 32-bit

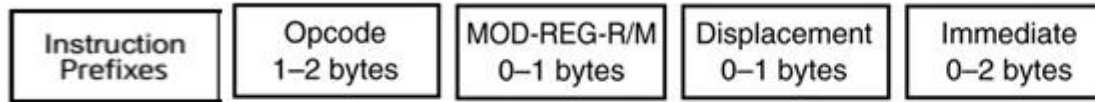




code segment descriptor

- In the code segment descriptor, **L-bit** and **D-bit** indicate the operation mode:
  - **L=1** for 64-bit instruction mode
  - **L=0 and D =1** for 32-bit instruction mode
  - **L=0 and D =0** for 16-bit instruction mode

### 16-bit instruction mode



### 16-bit instruction format

(a)

### 32-bit instruction mode (80386 through Pentium 4 only)



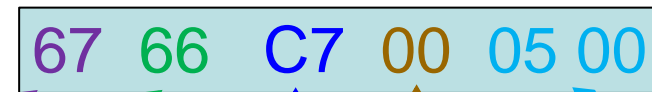
1 byte each

(b)

### 32-bit instruction format

- The **operand-size prefix 66H** selects the non-default operand size other than the default, while the **address-size prefix 67H** changes the default address size for memory operands.
- Instruction encoding example in 64 bit mode (default address size = 64 bit, default operand size = 32 bit):

– **MOV WORD PTR [EAX], 5 ;**



Address  
Prefix

Operand  
Prefix

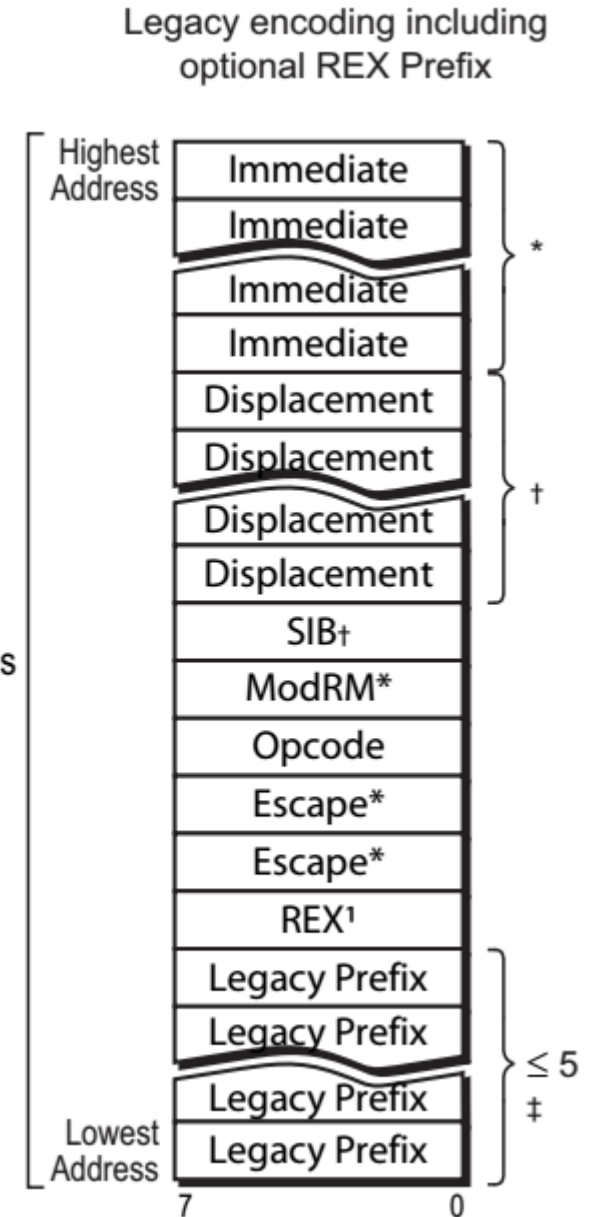
Opcode

ModR/M

Immediate

# Instruction Stored in Memory

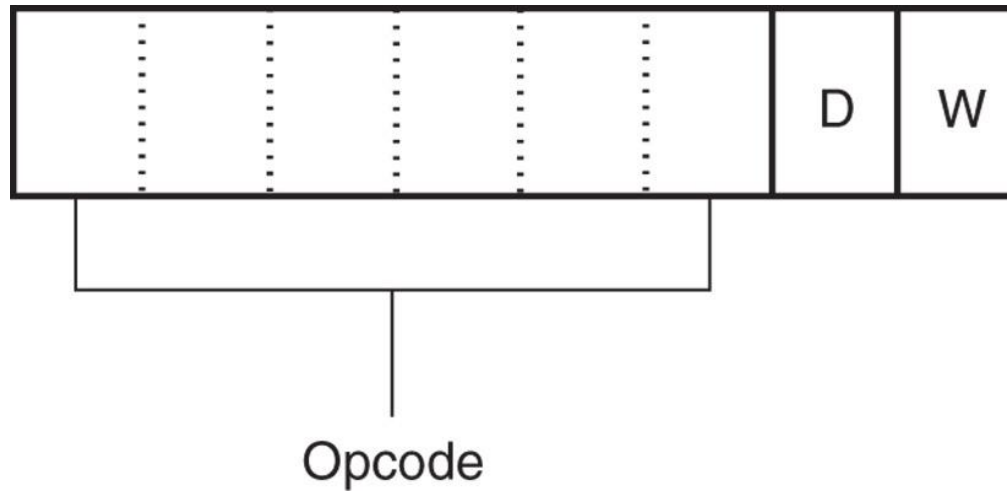
- Instructions are stored in memory in **little-endian** order. The first byte of an instruction is stored at the lowest memory address.
- Since instructions are strings of bytes, they may start at any memory address.
- The total instruction length must be less than or equal to 15. If this limit is exceeded, a **general-protection exception** results.



# The Opcode-Byte 1

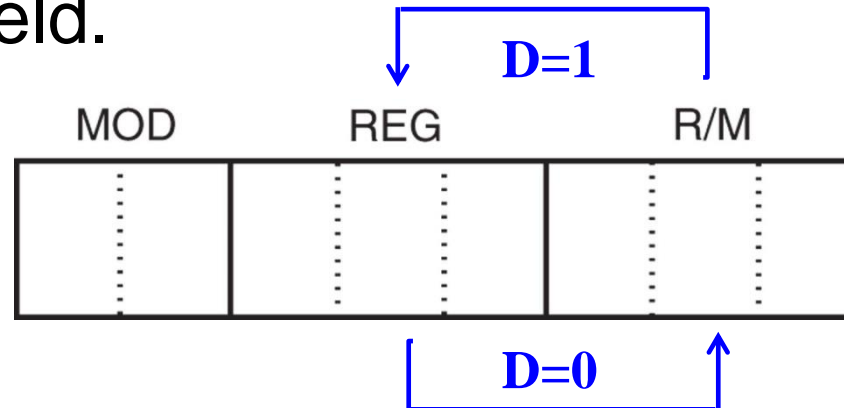
- Selects the operation (addition, subtraction, etc.,) performed by the microprocessor.
  - either 1 or 2 bytes long for most instructions
- Figure 4–2 illustrates the general form of the first opcode byte of many instructions.
  - first 6 bits of the first byte are the binary opcode
  - remaining 2 bits indicate the **direction** (D) of the data flow, and indicate the **data size** (W) (a byte or a word)

**Figure 4–2** Byte 1 of many machine language instructions, showing the position of the D- and W-bits.



- If the direction bit  $D=1$ , data flow to the register REG field from the R/M field.
- If the direction bit  $D=0$ , data flow to the R/M field from the REG field.

MOD-REG-R/M

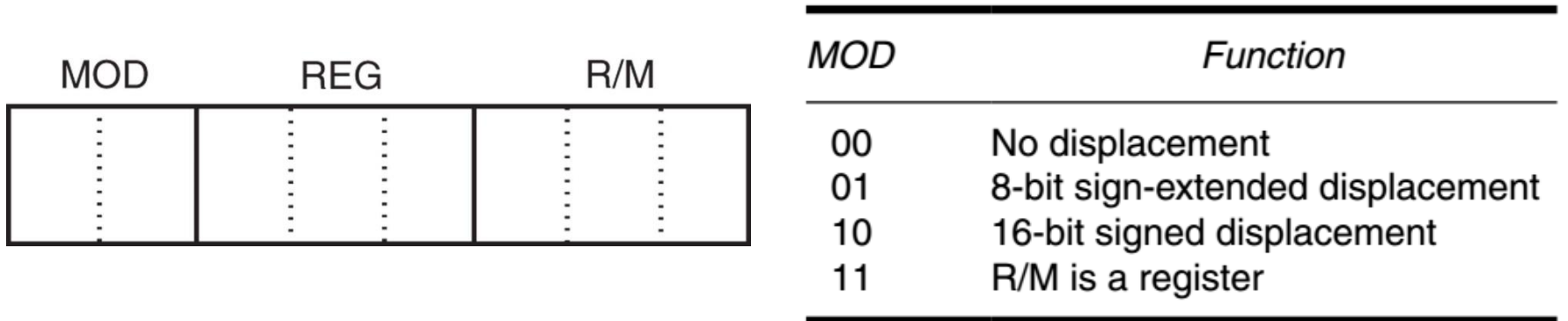


- If the W-bit=1, the data size is a *word* or *doubleword*.
- If the W-bit=0, the data size is always a *byte*.
- The W-bit appears in most instructions, while the D-bit appears mainly with the MOV and some instructions.

<i>Code</i>	<i>W = 0 (Byte)</i>	<i>W = 1 (Word)</i>	<i>W = 1 (Doubleword)</i>
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

# MOD Field-Byte 2

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



MOD field for the 16-bit instruction mode

- The MOD field specifies the addressing mode (MOD) for the selected instruction.
- The MOD field selects the type of addressing and whether a displacement is present with the selected type.

# ***MOD Field-Byte 2***

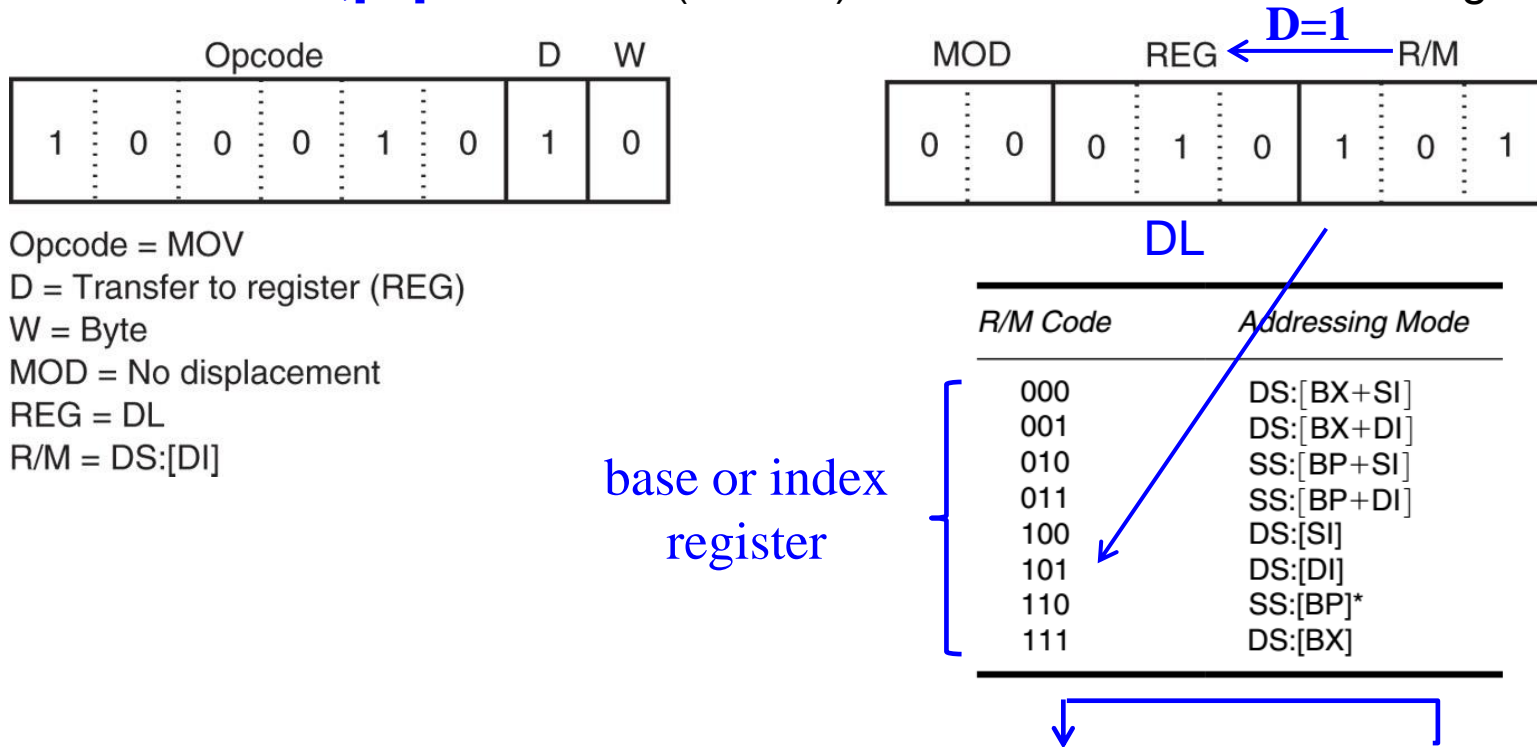
- Specifies addressing mode (MOD) and whether a displacement is present with the selected type.
  - If MOD field contains an 11, it selects the register-addressing mode
  - Register addressing specifies a register instead of a memory location, using the R/M field
- If the MOD field contains a 00, 01,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



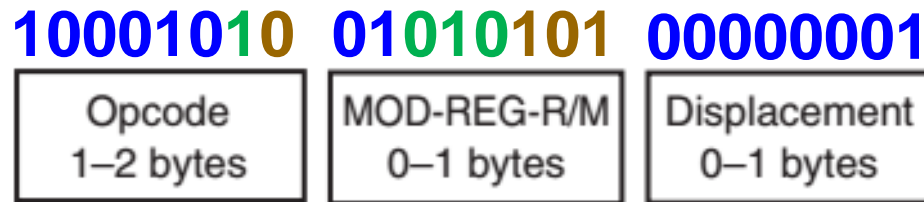
# ***R/M Memory Addressing***

- 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 (**8A15H**) converted to its machine language form.



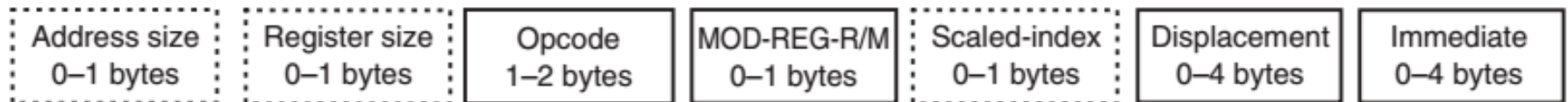
- If the instruction becomes **8A5501H**
- first 1 byte of the instruction remains the same
- the MOD field changes to 01 for 8-bit displacement, the instruction changes to **MOV DL, [DI+1]**



# 32-Bit Addressing Modes

- When R/M=100, an additional byte called a **scaled-index byte** appears in the instruction.
- A scaled-index byte indicates additional forms of scaled-index addressing.

<i>R/M Code</i>	<i>Function</i>
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	Uses scaled-index byte
101	SS:[EBP]*
110	DS:[ESI]
111	DS:[EDI]



- Over 32,000 variations of the MOV instruction alone in the 80386 - Core2 microprocessors.
- Figure 4–8 shows the format of the scaled-index byte as selected by a value of 100 in the R/M field of an instruction when the 80386 and above use a 32-bit address.
- The leftmost 2 bits select a scaling factor (multiplier) of 1x, 2x, 4x, 8x.
- Scaled-index addressing can also use a single register multiplied by a scaling factor.

**Figure 4–8** The scaled-index byte.



SS

00 =  $\times 1$

01 =  $\times 2$

10 =  $\times 4$

11 =  $\times 8$

— the index and base fields both contain register numbers

- If the microprocessor operates in the 32-bit instruction mode, The instruction

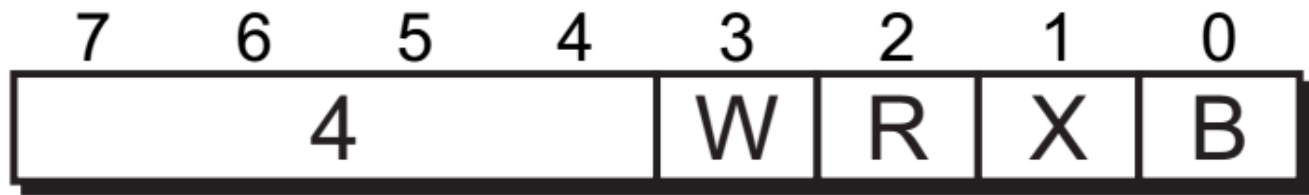
**MOV EAX,[EBX+4\*ECX]** is encoded as **8B048BH**.

- First byte (8BH): **100010 11** → **MOV**
- Second byte (04H): **00 000 100** → **EAX, scaled-index**
- Third byte (8BH): **10 001 011** →  **$\times 4$ , ECX, EBX**

# The 64-Bit Mode for the Pentium 4 and Core2

- In 64-bit mode, a prefix called **REX** (*register extension*) is added to enable use of the register and operand size extensions.
- REX is not a single unique value, but occupies a range (40h to 4Fh), following other prefixes and placing before the opcode.
- Purpose is to modify reg and r/m fields in the second byte of the instruction.
  - REX is needed to be able to **address registers R8 through R15**

- REX contains five fields. The upper nibble is unique to the REX prefix and identifies it as such. The lower nibble is divided into four 1-bit fields (W, R, X, and B).



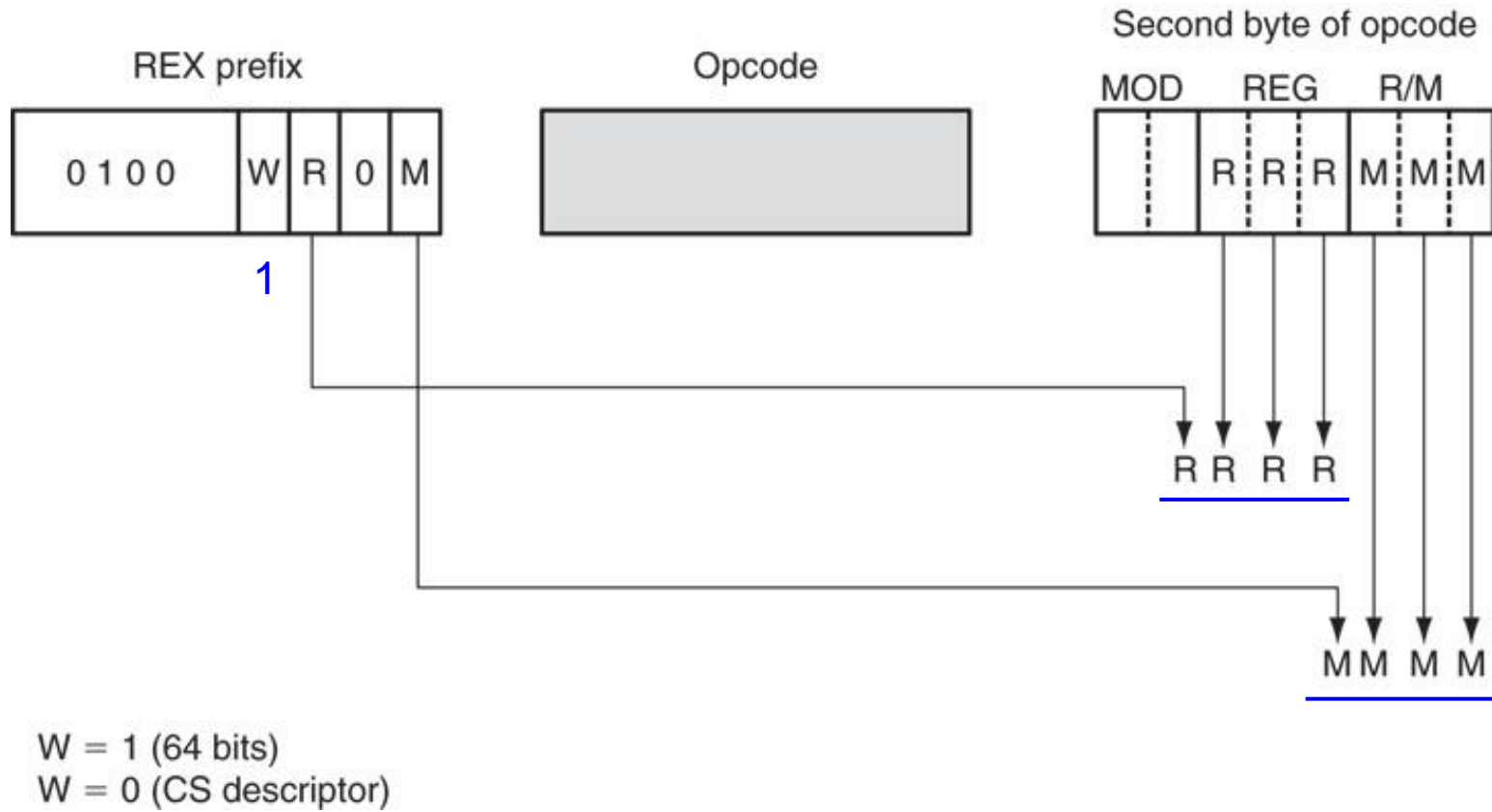
Mnemonic	Bit Position(s)	Definition
—	7:4	0100 (4h)
REX.W	3	0 = Default operand size 1 = 64-bit operand size
REX.R	2	1-bit (msb) extension of the ModRM <i>reg</i> field <sup>1</sup> , permitting access to 16 registers.
REX.X	1	1-bit (msb) extension of the SIB <i>index</i> field <sup>1</sup> , permitting access to 16 registers.
REX.B	0	1-bit (msb) extension of the ModRM <i>r/m</i> field <sup>1</sup> , SIB <i>base</i> field <sup>1</sup> , or opcode <i>reg</i> field, permitting access to 16 registers.

## REX Prefix-Byte Fields

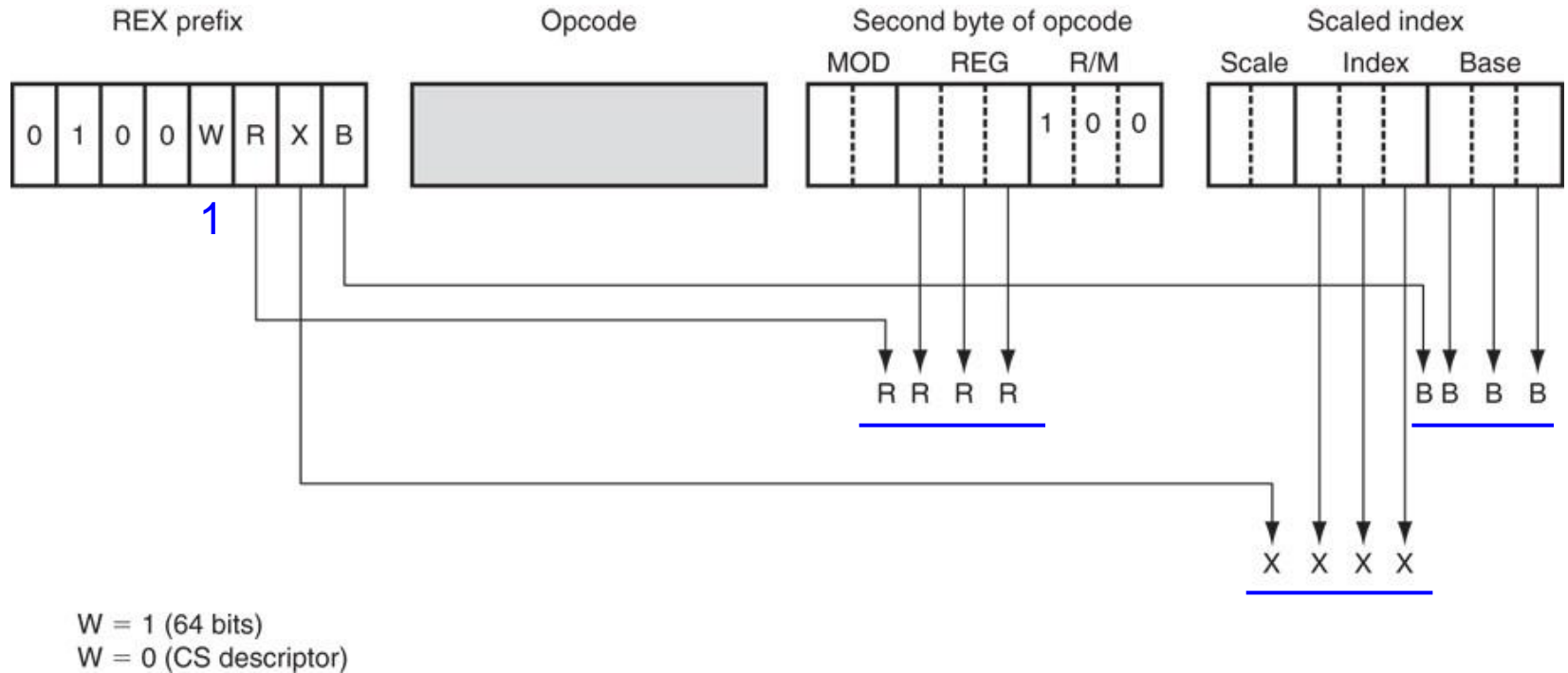
- Figure 4–11 illustrates the structure and application of REX to the second byte of the opcode.
- The reg field can only contain register assignments as in other modes of operation
- The r/m field contains either a register or memory assignment.
- Figure 4–12 shows the scaled-index byte with the REX prefix for more complex addressing modes and also for using a scaling factor in the 64-bit mode of operation.



**Figure 4–11** The application of REX without scaled index.



**Figure 4–12** The scaled-index byte and REX prefix for 64-bit operations.



# Legacy Prefixes

- Instruction prefixes are divided into four groups. For each instruction, only one prefix from each of the four groups is allowable.
- **Group 1**
  - 0xF0: LOCK
  - 0xF2: REPNE/REPZ
  - 0xF3: REP or REPE/REPZ
- **Group 2: segment override prefix**
  - 0x2E: CS segment override
  - 0x26: ES segment override
  - 0x36: SS segment override
  - 0x64: FS segment override
  - 0x3E: DS segment override
  - 0x65: GS segment override
- **Group 3**
  - 0x66: Operand-size override prefix
- **Group 4**
  - 0x67: Address-size override prefix

# Lock Prefix

- The **LOCK** prefix causes certain kinds of memory read-modify-write instructions to **perform atomic operation**.
- The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.
- The LOCK prefix can only be used with following instructions that write a memory operand: **BTC, BTR, BTS, ADC, ADD, AND, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, CMPXCHG, CMPXCHG8B, CMPXCHG16B, XADD and XCHG**.
- An **undefined opcode exception (#UD)** occurs if the LOCK prefix is used with any other instruction.

# Segment Override Prefix

- The processor can automatically choose a default segment according to the following rules:
  - Instructions: CS
  - Local Data: DS
  - Stack: SS
  - Destination Strings: ES
- Programmers can override the default segment with a segment-override prefix, which is a byte placed at the beginning of an instruction.
- For example, in 32-bit mode:
  - `MOV EAX, [EBX]` ; `8B 03`, default segment = DS.
  - `MOV EAX, CS:[EBX]` ; `2E 8B 03`, the `2E` is CS segment  
; override prefix.

# Operand-Size Override Prefix

- In 64-bit mode, instructions default to a 32-bit operand size.
- The prefix allows mixing of 16, 32, and 64-bit data:
  - a **REX (REX.W) prefix** can specify a 64-bit operand size
  - a **66H prefix** specifies a 16-bit operand size
  - the REX prefix takes precedence over the 66h prefix.

Operating Mode		Default Operand Size (Bits)	Effective Operand Size (Bits)	Instruction Prefix <sup>1</sup>	
				66h	REX.W <sup>3</sup>
Long Mode	64-Bit Mode	32 <sup>2</sup>	64	don't care	yes
			32	no	no
			16	yes	no
	Compatibility Mode	32	32	no	Not Appli- cable
			16	yes	
		16	32	yes	
			16	no	
			32	no	
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no	
			16	yes	
		16	32	yes	
			16	no	

Operand-Size  
Overrides

# How can instructions share an opcode?

Instruction	Opcode	Description
MOV reg/m16, imm16	C7	Move 16-bit immediate to 16-bit reg/memory operand
MOV r/m32, imm32	C7	Move 32-bit immediate to 32-bit reg/memory operand
MOV r/m64, imm32	REX.W + C7	Move 32-bit immediate to 64-bit reg/memory operand

- The default operand size is defined by the current operation mode, but the **operand-size override prefix (REX or 66H)** can change the default operand size. E.g., in 64 bit mode (default operand size = 32 bit):

- MOV WORD PTR [RAX], 5 ; 66 C7 00 05 00  
66H prefix opcode operand
- MOV DWORD PTR [RAX], 5 ; C7 00 05 00 00 00  
opcode operand
- MOV QWORD PTR [RAX], 5 ; 48 C7 00 05 00 00 00  
REX prefix opcode operand

# Address-Size Override Prefix

- The default address size is 64 bits in 64-bit mode. The size can be overridden to 32 bits, but 16-bit addresses are not supported in 64-bit mode.
- The address-size override prefix (67H) selects the non-default address size.

Operating Mode		Default Address Size (Bits)	Effective Address Size (Bits)	Address-Size Prefix (67h) <sup>1</sup> Required?
Long Mode	64-Bit Mode	64	64	no
			32	yes
	Compatibility Mode	32	32	no
			16	yes
		16	32	yes
			16	no
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no
			16	yes
		16	32	yes
			16	no

Address-Size Overrides



# How to change the address size for memory operands?

- The default address size for memory operands is determined by the current operation mode, but it can be overridden by an **address-size override prefix (67H)**, for example
  - In 64 bit mode, addresses are 64 bits by default, but they can be overridden to 32 bits by an address size prefix.
- Some examples:
  - in 64 bit mode (default address size = 64 bit)

**MOV** EAX, [RBX] ; **8B 03**  
opcode

**MOV** EAX, [EBX] ; **67 8B 03**  
67H prefix opcode

- in 32 bit mode (default address size = 32 bit)

**MOV** EAX, [EBX] ; **8B 03**  
opcode

**MOV** EAX, [BX] ; **67 8B 07**  
67H prefix opcode

# Summary of Prefixes

- **REX** prefix can be used in 64-bit mode to access the 64-bit registers and size extensions.
- **Legacy prefixes**
  - **Lock** prefix forces an operation that ensures exclusive use of shared memory in a multiprocessor environment.
  - **Repeat** prefix repeats its **associated string instruction** the number of times specified in the counter register (rCX).
  - **Segment override** prefix selects alternate segment register.
  - **Operand-size override** prefix affects the operand size for general-purpose instructions.
  - **Address-size override** prefix affects the address size of memory operands.

# Escape Sequence/Opcode

- Since there are more than 256 instructions defined by the architecture, multiple different opcode maps must be defined.
- Escape sequence** is used to expand the coding space by using one **escape opcode byte (0FH)** or **two-byte escape (0F38H, 0F3AH)** to provide another opcodes.

Primary  
Opcode Map


Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
0	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH CS <sup>3</sup>	escape to secondary opcode map
1	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH DS <sup>3</sup>	POP DS <sup>3</sup>
-	SUB							-

Secondary  
Opcode Map

Prefix	Nibble <sup>1</sup>	8	9	A	B	C	D	E	F
n/a	0	INVD	WBINVD (F3) WBNOINVD		UD2		Group P <sup>2</sup>  PREFETCH	FEMMS	3DNow! See "3DNow!™ Opcodes" on page 491
n/a	1	Group 16 <sup>2</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>	NOP <sup>3</sup>
none	2	MOVAPS Vps, Wps    Wps, Vps		CVTPI2PS Vps, Qpj	MOVNTPS Mo, Vps	CVTTSP2PI Ppj, Wps	CVTPS2PI Ppj, Wps	UCOMISS Vss, Wss	COMISS Vss, Wss
F3				CVTSI2SS Vss, Ey	MOVNTSS Md, Vss	CVTTSS2SI Gy, Wss	CVTSS2SI Gy, Wss		
66		MOVAPD Vpd, Wpd    Wpd, Vpd		CVTPI2PD Vpd, Qpj	MOVNTPD Mo, Vpd	CVTTSD2PI Ppj, Wpd	CVTPD2PI Ppj, Wpd	UCOMISD Vsd, Wsd	COMISD Vsd, Wsd
F2				CVTSI2SD Vsd, Ey	MOVNTSD Mq, Vsd	CVTTSD2SI Gy, Wsd	CVTSD2SI Gy, Wsd		
n/a	3	Escape to 0F_38h opcode map		Escape to 0F_3Ah opcode map					

# Escape Sequence/Opcode

- For example:
  - 0F AF is the opcode of **IMUL** *r16, r/m16*
  - 0F B6 is the opcode of **MOVZX** *r16, r/m8*
- These common instructions were introduced after the original 8086, and there was no coding-space left to give them single-byte opcodes.



Opcode	Instruction
F6 /5	IMUL <i>r/m8*</i>
F7 /5	IMUL <i>r/m16</i>
F7 /5	IMUL <i>r/m32</i>
REX.W + F7 /5	IMUL <i>r/m64</i>
0F AF /r	IMUL <i>r16, r/m16</i>
0F AF /r	IMUL <i>r32, r/m32</i>

IMUL—Signed Multiply

Opcode	Instruction	64-Bit Mode
0F B6 /r	MOVZX <i>r16, r/m8</i>	Valid
0F B6 /r	MOVZX <i>r32, r/m8</i>	Valid
REX.W + 0F B6 /r	MOVZX <i>r64, r/m8*</i>	Valid
0F B7 /r	MOVZX <i>r32, r/m16</i>	Valid
REX.W + 0F B7 /r	MOVZX <i>r64, r/m16</i>	Valid

MOVZX—Move with Zero-Extend

## 4-2 LOAD EFFECTIVE ADDRESS

- Load-effective address instruction set was designed to support high-level languages like C.
- Two types of load-effective address instructions:
  - **LEA**: Loads a **near pointer** (offset)
  - **LDS**, **LES**, **LFS**, **LGS**, and **LSS**: Loads a **far pointer** (segment selector and offset)

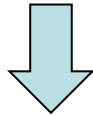
- Consider a struct representing (x, y) coordinates:

```
struct point
{
    int x;
    int y
};
```

- Now imagine two functions:

// load the value of y

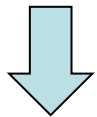
```
int query(struct point points[], int i)
{
    return points[i].y;
}
```



**MOV** EDX, [EBX + 8\*EAX + 4]

// load the address of y

```
int query(struct point points[], int i)
{
    return &(points[i].y);
}
```



**LEA** ESI, [EBX + 8\*EAX + 4]

note: EBX is the base of array (points); EAX is for variable i; 8 is the scale factor of each point and 4 is the offset of y.

# LEA

- Loads a 16- or 32-bit register with the offset address of the data specified by the operand.
- By comparing LEA with MOV, we observe that:
  - `LEA BX,[DI]` loads the **offset address** specified by [DI] (contents of DI) into the BX register;
  - `MOV BX,[DI]` loads the **data** stored at the memory location addressed by [DI] into register BX.
- The **SEG** and **OFFSET** directive return the segment and offset value of a memory location.

- **OFFSET** performs same function as LEA instruction if the **operand is a displacement**.

**Example 4–3** is a program that loads SI with the address of DATA1 and DI with the address of DATA2. It then exchanges the contents of these memory locations.

			.MODEL SMALL	;select small model
0000			.DATA	;start data segment
0000 2000	DATA1	DW	2000H	;define DATA1
0002 3000	DATA2	DW	3000H	;define DATA2
0000			.CODE	;start code segment
			.STARTUP	;start program
0017 BE 0000 R			<u>LEA SI,DATA1</u>	;address DATA1 with SI
001A BF 0002 R			<u>MOV DI,OFFSET DATA2</u>	;address DATA2 with DI
001D 8B 1C			MOV BX,[SI]	;exchange DATA1 with DATA2
001F 8B 0D			MOV CX,[DI]	
0021 89 0C			MOV [SI],CX	
0023 89 1D			MOV [DI],BX	
			.EXIT	
			END	

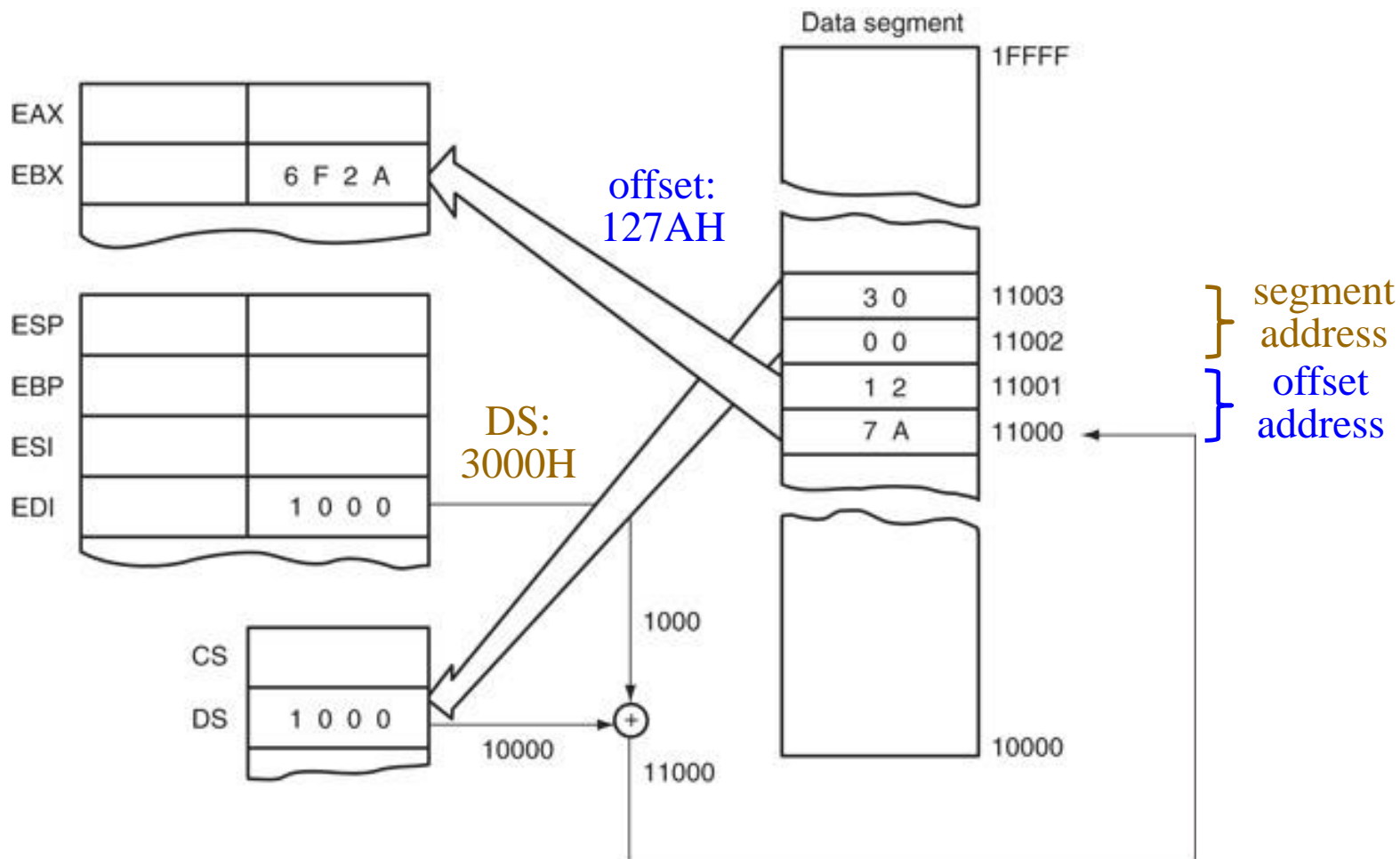


- **OFFSET** is more efficient than **LEA** instruction
  - one clock for **MOV BX,OFFSET LIST**
  - two clocks for **LEA BX,LIST** in 80486 microprocessor
- The **MOV BX,OFFSET LIST** instruction is actually assembled as a move immediate instruction and is more efficient (e.g., **MOV BX, 0x9**).
- Why is **LEA** instruction available if **OFFSET** accomplishes the same task?
  - **OFFSET** only functions with simple operands such as **LIST** , not for an operand such as **[DI]**, **LIST [SI]**, e.g.,
  - **LEA BX, [DI]** → **MOV BX, DI**
  - **LEA SI, [BX+DI]**. This instruction adds **BX** to **DI** and stores the sum in the **SI** register. The sum generated by this instruction is a modulo-64K sum.

# LDS, LES, LFS, LGS, and LSS

- LEA instruction loads any 16-bit register with the offset address
  - determined by the addressing mode selected
- LDS and LES load **far addresses** from memory
  - load a 16 or 32-bit register with offset address retrieved from a memory location
  - then load either DS or ES with a segment address or segment selector retrieved from memory
  - 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.



- Instructions use any memory-addressing modes to access a **32-bit** or **48-bit memory section** that contain both segment and offset address
  - **32-bit far pointer**: 16-bit segment + 16-bit offset
  - **48-bit far pointer**: 16-bit selector + 32-bit offset
- In 80386 and above, LFS, LGS, and LSS are added to the instruction set.
- 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 or segment selector.

- LDS, LES, LFS, LGS, and LSS instructions obtain a new far address from memory.
  - offset address appears first, followed by the segment address or segment selector
- This format is used for storing all 32-bit memory addresses.

---

#### *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

---

### Load-effective address instructions

- A far address can be stored in memory by the assembler. **The most useful of the load instructions is the LSS instruction.**
- The following program shows an example of reactivating the old stack area by loading both SS and SP with the LSS instruction.

CLI  
 MOV AX, SP      *save old SP, SS*  
 MOV WORD PTR SADDR, AX  
 MOV AX, SS  
 MOV WORD PTR SADDR+2, AX

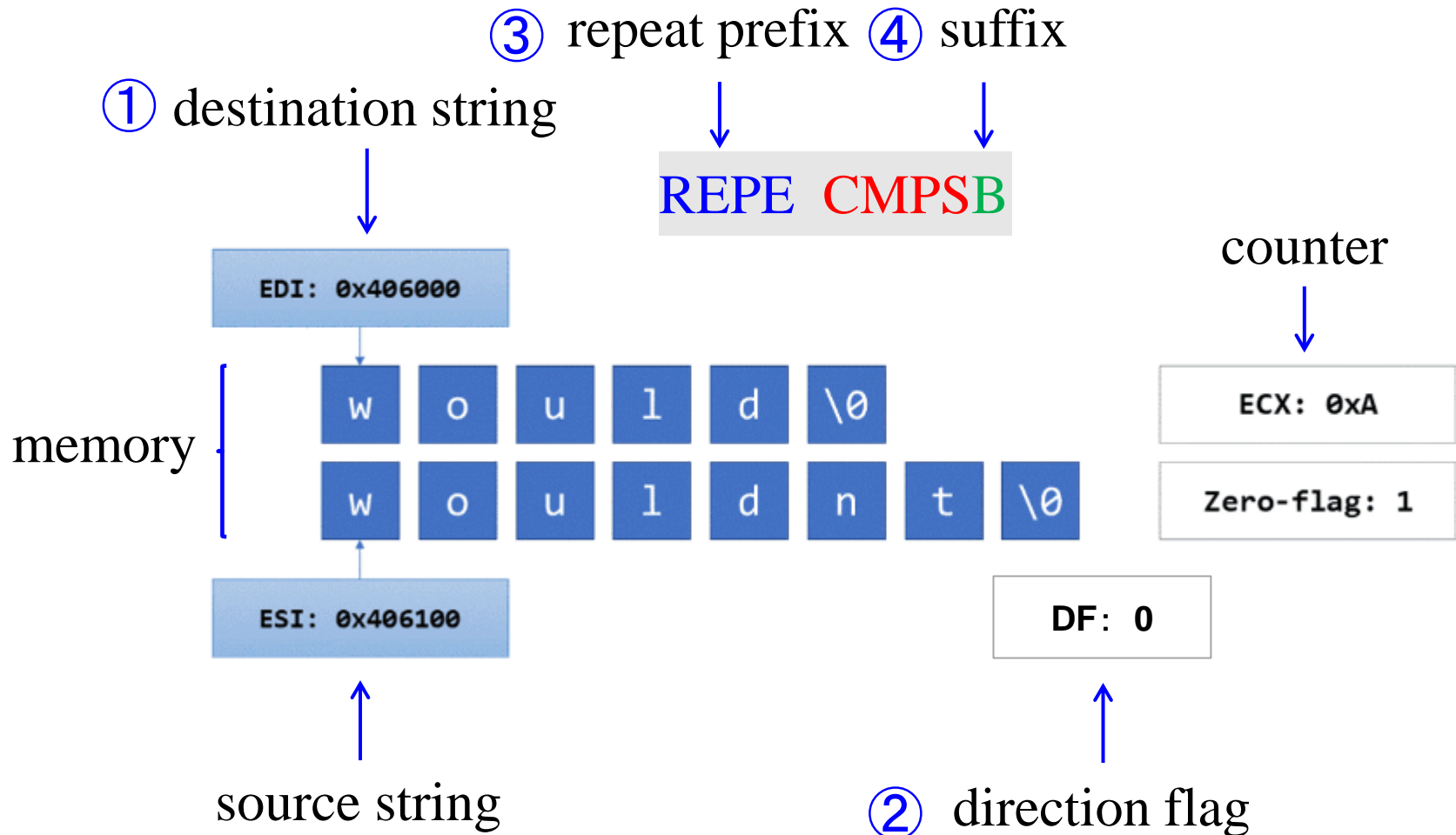
MOV AX, DS  
 MOV SS, AX      *load new SP, SS*  
 MOV AX, OFFSET STOP  
 MOV SP, AX  
 STI  
 .....  
 MOV AX, AX  
 MOV AX, AX  
 LSS SP, SADDR      *restore old stack*

- CLI (**disable interrupt**) and STI (**enable interrupt**) instructions must be included to disable interrupts.

# 4-3 STRING DATA TRANSFERS

- Five string data transfer instructions
  - LODS, STOS, MOVS, INS, OUTS
- Two string comparison instructions
  - SCAS, CMPS
    - first 2-3 letters manifest what the instruction does,
    - the “S” in all instructions stands for “String”.
- Each allows data transfers or comparison as a single byte, word, or doubleword and implicitly uses DI, SI, or both registers to address memory.
- String instructions execute efficiently because they automatically repeat and increase array indexes.

# Example: String Comparison





- **DI/EDI, SI/ESI**
  - DI/EDI with extra segment ES, **cannot** be overridden
  - SI/ESI with data segment DS, **can** be overridden
- **Direction Flag**
  - D=0, auto-increment
  - D=1, auto-decrement
- **REP and CX/ECX**
  - a repeat prefix (REP) allows the instruction to be repeated n times where n is the value stored in CX/ECX.
- **Permissible forms with suffix**
  - B, byte e.g.,
  - W, word –MOV**S**B, byte-sized MOV
  - D, doubleword –LODS**W**, word-sized LOD

# 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 (ES)** for all string instructions that use it
  - SI offset address accesses data by default in the **data segment (DS)**
- 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

# 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
  - STD instruction selects the auto-decrement mode

# Using a Repeat Prefix

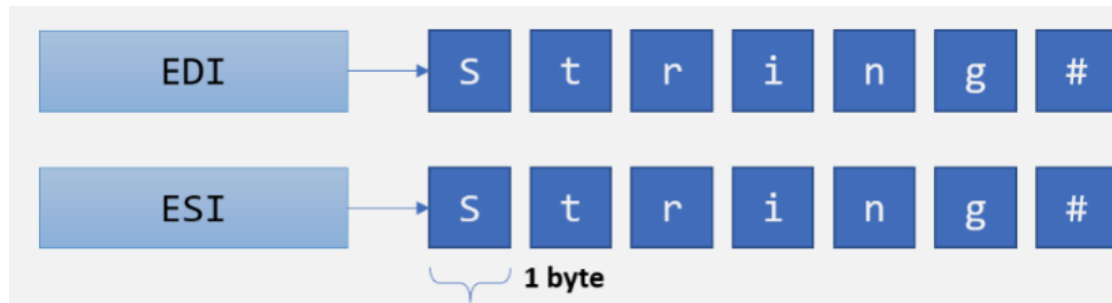
- A string primitive instruction processes only a single memory value or pair of values. If you add a **repeat prefix**, the instruction repeats, using **CX or ECX as a counter**.
- The repeat prefix processes an entire array using a single instruction.
- The following repeat prefixes are used:

REP	Repeat while ECX > 0
REPZ, REPE	Repeat while the Zero flag is set and ECX > 0
REPNZ, REPNE	Repeat while the Zero flag is clear and ECX > 0

# Example: Copy a String

- In the following example, MOVSB moves 10 bytes from string1 to string2.
- ESI and EDI are automatically incremented when MOVSB repeats. This behavior is controlled by Direction flag.

```
cld                                ; clear direction flag
mov  esi,OFFSET string1           ; ESI points to source
mov  edi,OFFSET string2           ; EDI points to target
mov  ecx,10                       ; set counter to 10
rep  movsb                        ; move 10 bytes
```



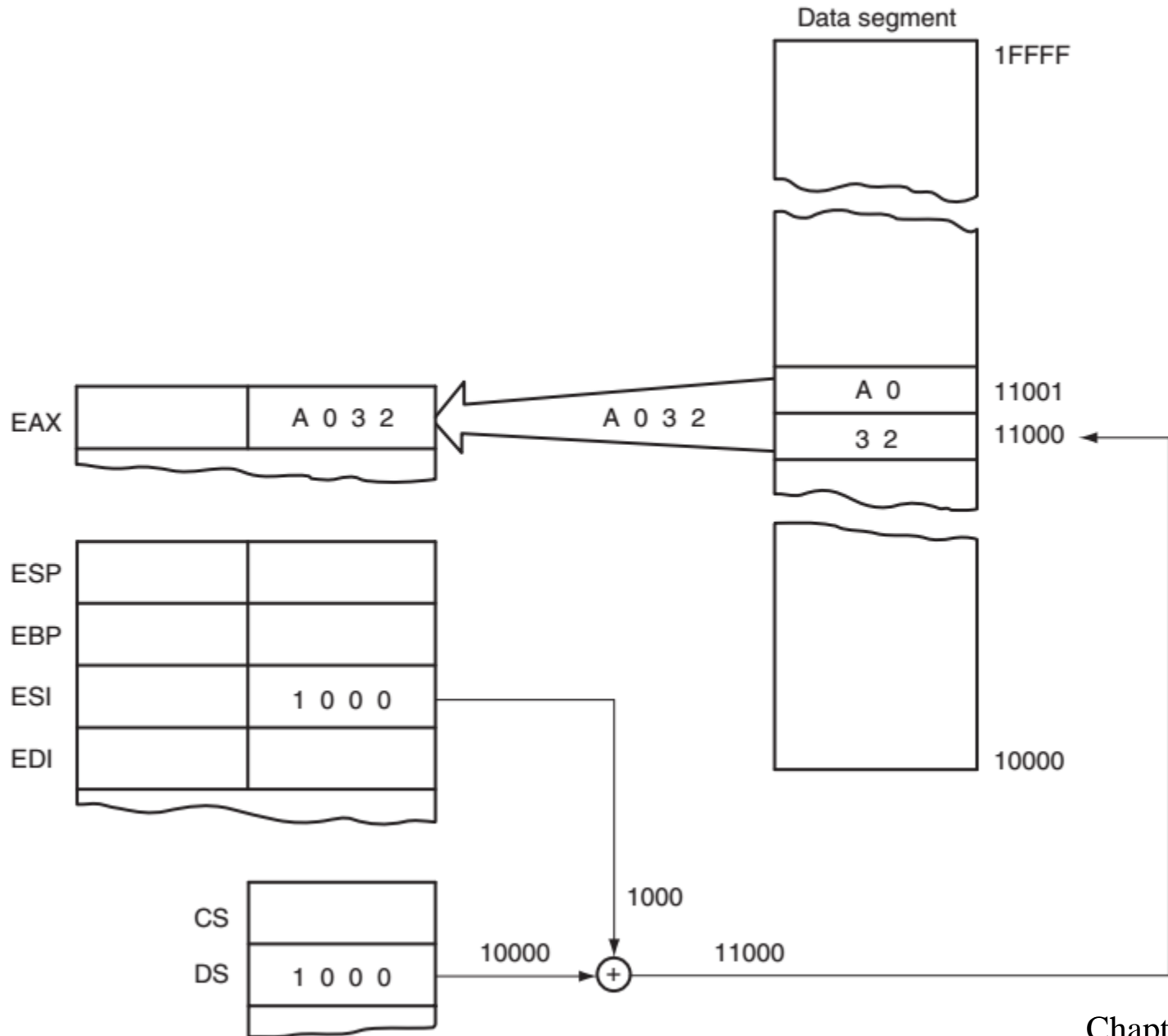
string2

string1

# LODS

- LODS transfers a byte, word or doubleword from the source location **DS:SI** to AL, AX or EAX. A suffix represents the size to operate on.
  - LODS**B** :  $\pm 1$  is added to DS: SI/ESI
  - LODS**W**:  $\pm 2$  is added to DS: SI/ESI
  - LODS**D** :  $\pm 4$  is added to DS: SI/ESI
- LODS uses **implicit operands** (AL, AX, EAX), which are not explicitly mentioned in the arguments or the opcode.
- Figure 4–18 shows the LODSW instruction.

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



# STOS

- The STOS<sup>B</sup>, STOS<sup>W</sup>, and STOS<sup>D</sup> store AL, AX, or EAX to the destination location ES:DI.
  - STOS<sup>B</sup> :  $\pm 1$  is added to ES: DI/EDI
  - STOS<sup>W</sup>:  $\pm 2$  is added to ES: DI/EDI
  - STOS<sup>D</sup> :  $\pm 4$  is added to ES: DI/EDI
- STOS uses implicit operands (AL, AX, EAX), which are not explicitly mentioned in the arguments or the opcode.



# ***STOS with a REP***

- The **repeat prefix** (REP) is added to any string data transfer instruction **except LODS to prevent the data in the register from being overwritten.**
- If CX reaches a value of 0, the instruction terminates and the program continues.
- If CX is loaded with 100 and a REP STOSB instruction executes, the microprocessor automatically repeats the STOSB 100 times.

# ***STOS with a REP***

- When used with the REP prefix, the STOS instruction is useful for filling all elements of a string or array with a single value.
- For example, the following code initializes each byte in string1 to 0FFh.

```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov     al,0FFh                ; value to be stored
mov     edi,OFFSET string1     ; EDI points to target
mov     ecx,Count              ; character count
cld                                     ; direction = forward
rep     stosb                   ; fill with contents of AL
```

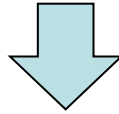
# MOVS

- MOVS transfers a byte, word, or doubleword from **DS: SI** to **ES: DI** and updates SI and DI accordingly. A suffix (B, W or D) indicates the data size to operate on
  - MOVSB :  $\pm 1$  is added to DS: SI and ES: DI
  - MOVSW :  $\pm 2$  is added to DS: SI and ES: DI
  - MOVSD :  $\pm 4$  is added to DS: SI and ES: DI
- MOVS is used to move a block of memory.
- MOVS is the **only memory-to-memory transfer** allowed in the 8086–Pentium 4 microprocessors.

- Example, transferring two blocks of doubleword memory

### C++ version of example

```
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
    for (int a = 0; a < blockSize; a++)
    {
        *blockB = *blockA++;
        blockB++;
    }
}
```

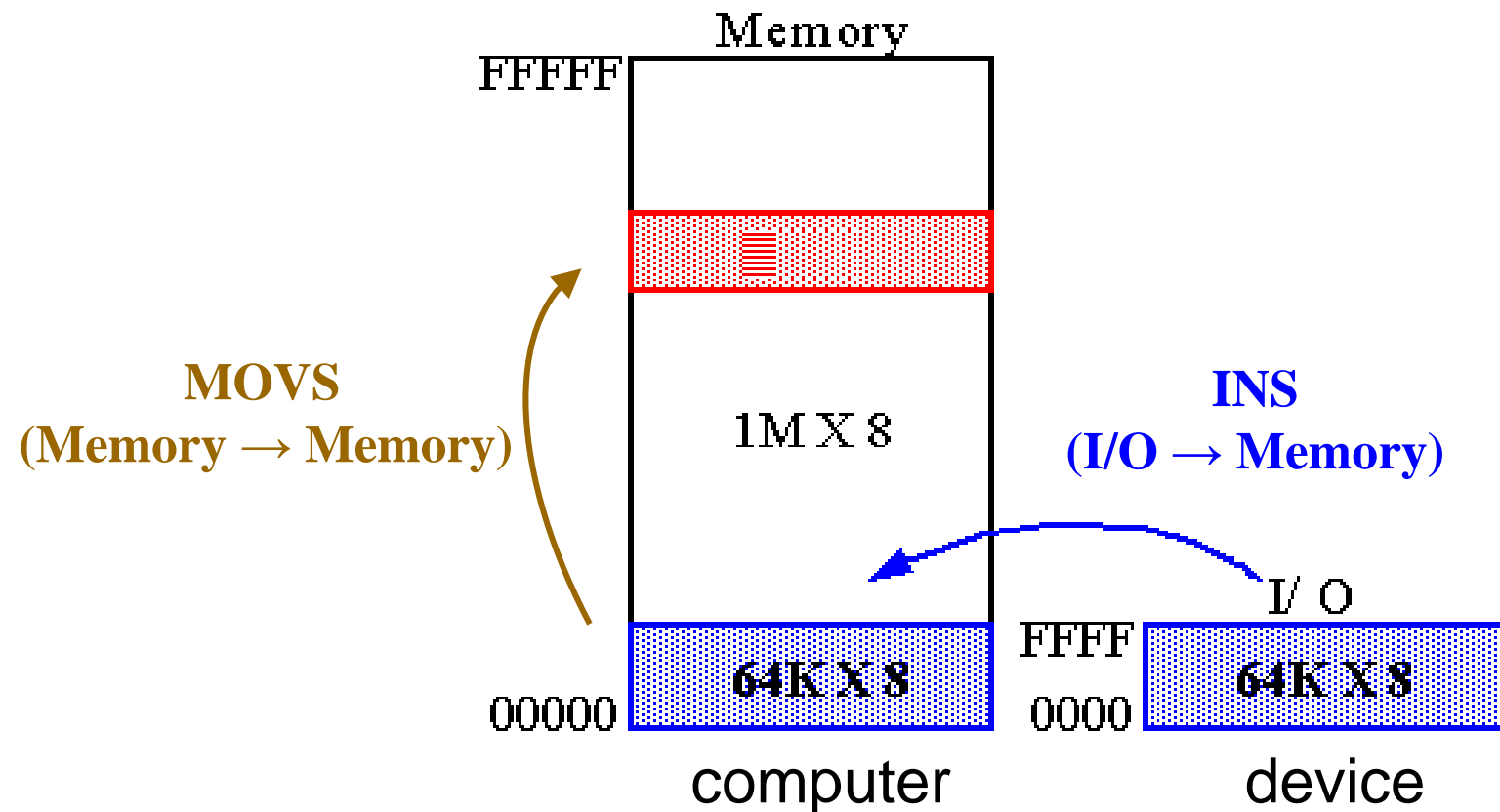


```
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
    _asm{
        push es                ;save registers
        push edi
        push esi
        push ds                ;copy DS into ES
        pop es
        mov esi, blockA        ;address blockA
        mov edi, blockB        ;address blockB
        mov ecx, blockSize     ;load count
        rep movsd              ;move data
        pop esi
        pop edi
        pop es                  ;restore registers
    }
}
```

### inline assembler of example

# INS

- Transfers a byte, word, or doubleword of data from an I/O device into the extra segment memory location.



- INS uses DX or EDX as the source operand to specify the I/O address or I/O port.
- Destination operand is a memory location addressed by the ES:DI or ES:EDI.
- 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

- Two forms of INS instruction are allowed: the **explicit-operands form** and **no-operands form**.
- The explicit-operands form allows the source and destination operands to be specified explicitly, e.g., **INS WORD PTR** [DI], DX.
- The source operand must be DX, and the destination operand should be ES:DI or ES:EDI.
- The no-operands form provides “short forms” of the byte, word, and doubleword versions of the INS instructions.

- Three basic forms of **no-operands form**:
  - **INSB** inputs data from an 8-bit I/O device and stores it in a memory location indexed by DI.
  - **INSW** instruction inputs 16-bit I/O data and stores it in a word-sized memory location.
  - **INSW** instruction inputs 16-bit I/O data and stores it in a word-sized memory location.
  - **INSD** instruction inputs 32-bit I/O data and stores it in a doubleword-sized memory location.
- 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



- Example 4–9 shows a sequence of instructions that inputs 50 bytes of data from an I/O device whose address is 03ACH and stores the data in extra segment memory array LISTS.

#### EXAMPLE 4–9

```
                                ;Using the REP INSB to input data to a memory array
                                ;
0000 BF 0000 R                 MOV DI,OFFSET LISTS           ;address array
0003 BA 03AC                   MOV DX,3ACH                   ;address I/O
0006 FC                         CLD                           ;auto-increment
0007 B9 0032                   MOV CX,50                     ;load counter
000A F3/6C                     REP INSB                       ;input data
```

# OUTS

- Transfers a byte, word, or doubleword of data from the data segment memory location address to an I/O device.
  - source operand is a memory location addressed by the DS:SI or DS:ESI.
  - destination operand (I/O address or I/O port) is contained in 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

- Two forms of OUTS instruction are allowed: the **explicit-operands form** and **no-operands form**.
- The explicit-operands form allows the source and destination operands to be specified explicitly, e.g., **OUTS** DX, **WORD PTR** [SI].
- The source operand should be DS:SI or DS:ESI and the destination operand must be DX.
- The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions.

- Three basic forms of **no-operands form**:
  - OUTSB instruction outputs a byte-sized memory data indexed by SI to an 8-bit I/O device.
  - OUTSW instruction outputs a word-sized memory data to a 16-bit I/O device.
  - OUTSD instruction outputs a doubleword-sized memory data to a 32-bit I/O device.
- Note that in the 64-bit mode, there is no 64-bit output, but the memory address is 64 bits and located in RDI for the OUTS instructions.

- Example 4–10 shows a short sequence of instructions that transfer data from a data segment memory array (ARRAY) to an I/O device at I/O address 3ACH.

#### EXAMPLE 4–10

;Using the REP OUTSB to output data from a memory array  
;

0000 BE 0064 R	MOV SI,OFFSET ARRAY	;address array
0003 BA 03AC	MOV DX,3ACH	;address I/O
0006 FC	CLD	;auto-increment
0007 B9 0064	MOV CX,100	;load counter
000A F3/6E	REP OUTSB	;output data

# 4-4 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
- Is a rare type of instruction that has two outputs.
- 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. This instruction occupies 1 byte of memory.

# various forms of the XCHG instruction

<i>Assembly Language</i>	<i>Operation</i>
XCHG AL,CL	Exchanges the contents of AL with CL
XCHG CX,BP	Exchanges the contents of CX with BP
XCHG EDX,ESI	Exchanges the contents of EDX with ESI
XCHG AL,DATA2	Exchanges the contents of AL with data segment memory location DATA2
XCHG RBX,RCX	Exchange the contents of RBX with RCX (64-bit mode)

- When using a memory-addressing mode and the assembler, it doesn't matter which operand addresses memory. For example, the **XCHG AL,[DI]** is identical to the **XCHG [DI],AL**.
- **XCHG** is useful for implementing semaphores for process synchronization.



## acquire the spinlock

; lock variable. 1 = locked, 0 = unlocked.

```
locked dd 0
```

; Set the EAX register to 1.

```
spin_lock:
```

```
    mov    eax, 1
```

; Atomically swap the EAX with the lock.

```
    xchg   eax, [locked]
```

; Test EAX with itself.

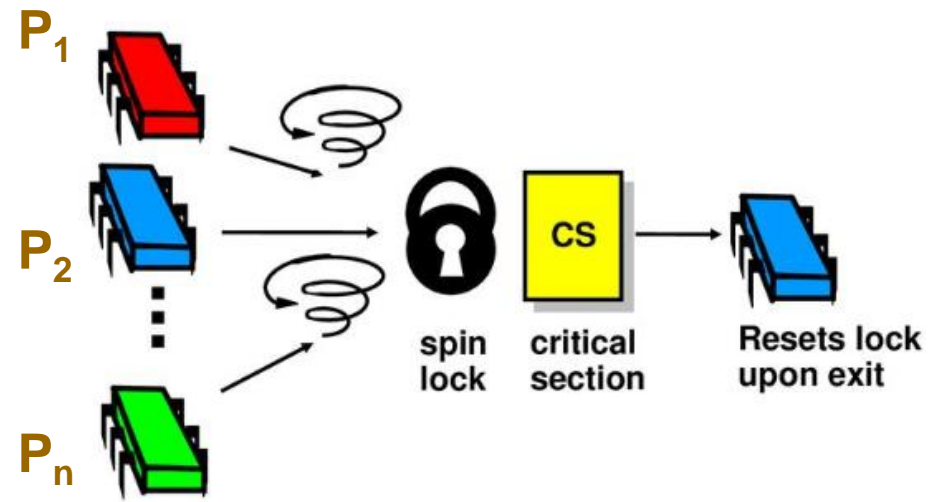
```
    test   eax, eax
```

; If EAX is 0, we just obtain and lock it.

; Otherwise, repeatedly request the lock.

```
    jnz    spin_lock
```

```
    ret
```



## release the spinlock

; Set the EAX register to 0.

```
spin_unlock:
```

```
    mov    eax, 0
```

; Atomically swap the EAX register

; with the lock variable.

```
    xchg   eax, [locked]
```

```
    ret
```

# LAHF and SAHF

- **LAHF** instruction transfers the lower 8 bits of the EFLAGS register into the AH register.
- **AH := EFLAGS (SF:ZF:0:AF:0:PF:1:CF)**
  - Load sign flag (SF), zero flag (ZF), auxiliary carry flag (AF), parity flag (PF), and carry flag (CF).
  - Set the reserved bits 1, 3, and 5 of the EFLAGS to 1, 0, and 0, respectively, in the AH register.
- The LAHF instruction is available in 64-bit mode if CPUID.800000001H:ECX.LAHF-SAHF[bit 0] = 1.

# LAHF and SAHF

- **SAHF** instruction transfers the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively).
- SAHF instruction ignores bits 1, 3, and 5 of register AH; it sets those bits in the EFLAGS register to 1, 0, and 0, respectively.
- The SAHF instruction is available in 64-bit mode if CPUID.800000001H:ECX.LAHF-SAHF[bit 0] = 1.

# XLAT

- The **XLAT** (Table Look-up Translation) instruction uses **implicit operands** (AL, BX):
  - the unsigned integer in **AL** register as an offset into a table (**[BX]**) and copies the contents of the table at that location (**[BX+AL]**) to AL register
  - XLAT works like **MOV AL, [seg:BX + AL]**
  - **seg:[BX]** is the base address of the table. The DS is the default segment. It may be overridden by a segment prefix.
- Note that the **[seg:BX + AL]** is not a legal memory operand, only XLAT accepts it.

- XLAT is often used to translate data from one format to another. For example, to translate index of food in menu into price for food:
  - first, reserve 256 bytes for table containing price;
  - next, load DS: BX with address of that table, and put index of food into AL;
  - then XLAT will translate the index into price from the table.
- XLAT writes AL without changing EAX[31:8].
- The workflow of XLAT instruction:
  - first adds the contents of AL to BX to form a memory address within the data segment
  - then copies the contents of this address into AL

- Suppose that a 7-segment LED display lookup table is stored in memory at address TABLE. The XLAT instruction then uses the lookup table to translate the BCD number in AL to a 7-segment code in AL.

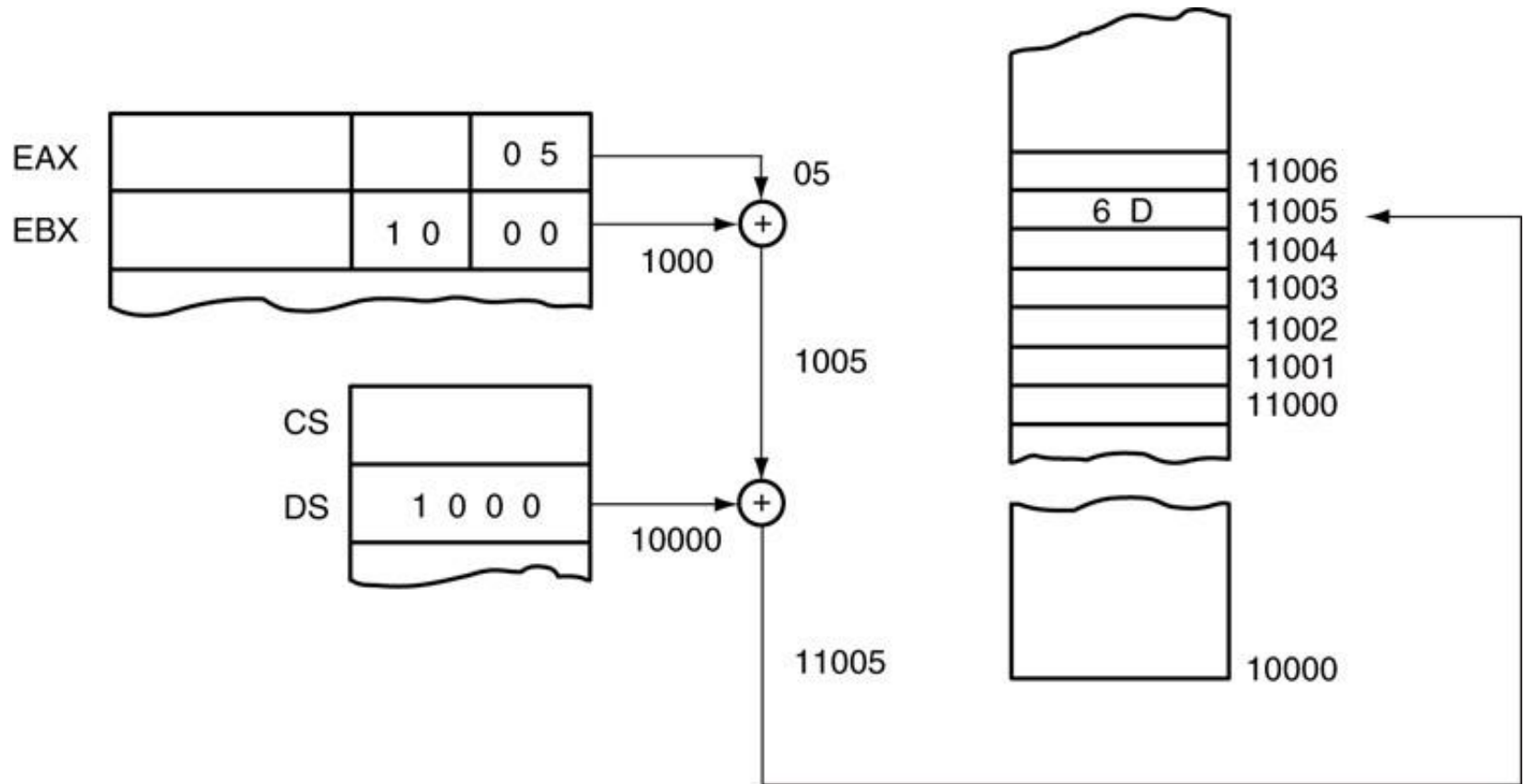
```
TABLE  DB  3FH, 06H, 5BH, 4FH      ;lookup table
        DB  66H, 6DH, 7DH, 27H
        DB  7FH, 6FH

LOOK:   MOV  AL,5                    ;load AL with 5 (a test number)
        MOV  BX,OFFSET TABLE      ;address lookup table
        XLAT                        ;convert
```

**7-segment code**

**BCD number**

**Figure 4–19** shows the operation of aforementioned example program if TABLE = 1000H, DS = 1000H, and the initial value of AL = 05H (BCD). After the translation, AL = 6DH.

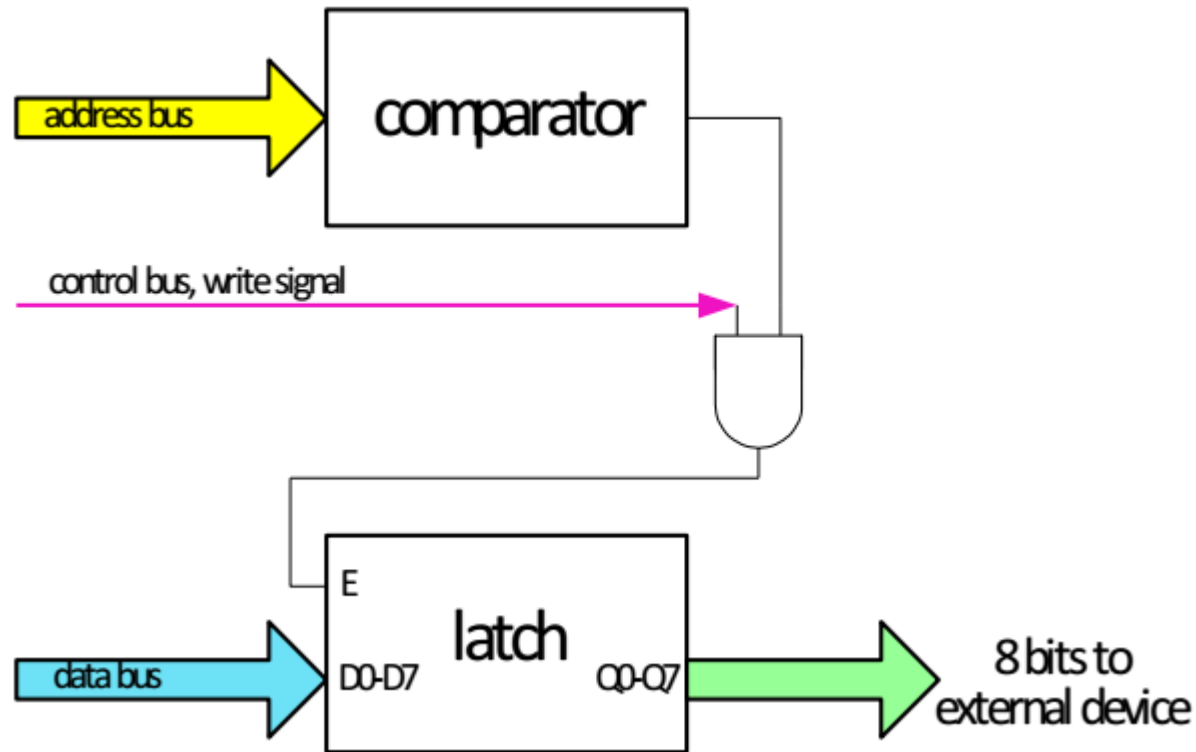


# Input Ports and Output Ports

- External devices such as screen, display, keyboard, mouse, hard disk, and network are connected to the data bus via input ports and output ports.
- Each input or output port has a unique address just like each byte-cell in the memory has a unique address.

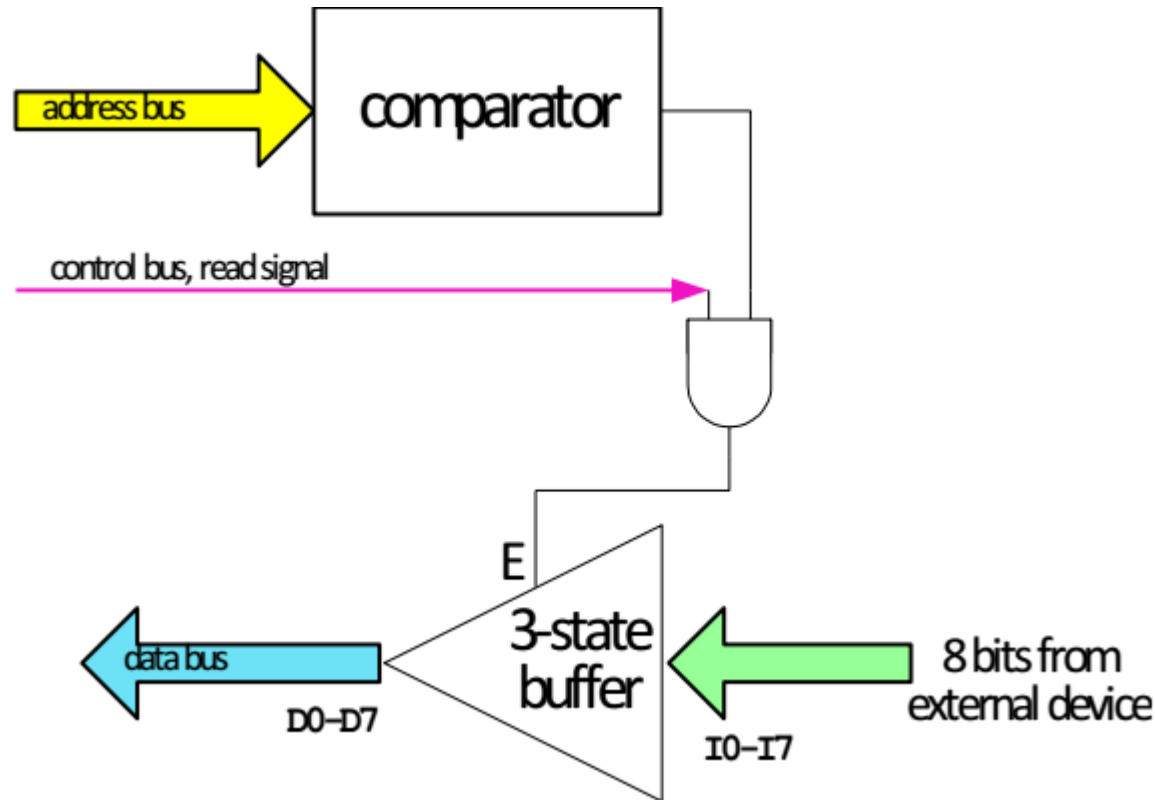


# Principle of Output Port



- An output port contains a comparator that compares the fixed address with the value on the address bus.
- A latch stores the value from the data bus if the address is equal to the port address and there is a write signal on the control bus.

# Principle of Input Port



- Each of the inputs from the external device goes through a three-state buffer to the data bus.
- The three-state buffer is enabled when the address bus is equal to the fixed address of the input port and there is a read signal on the control bus.

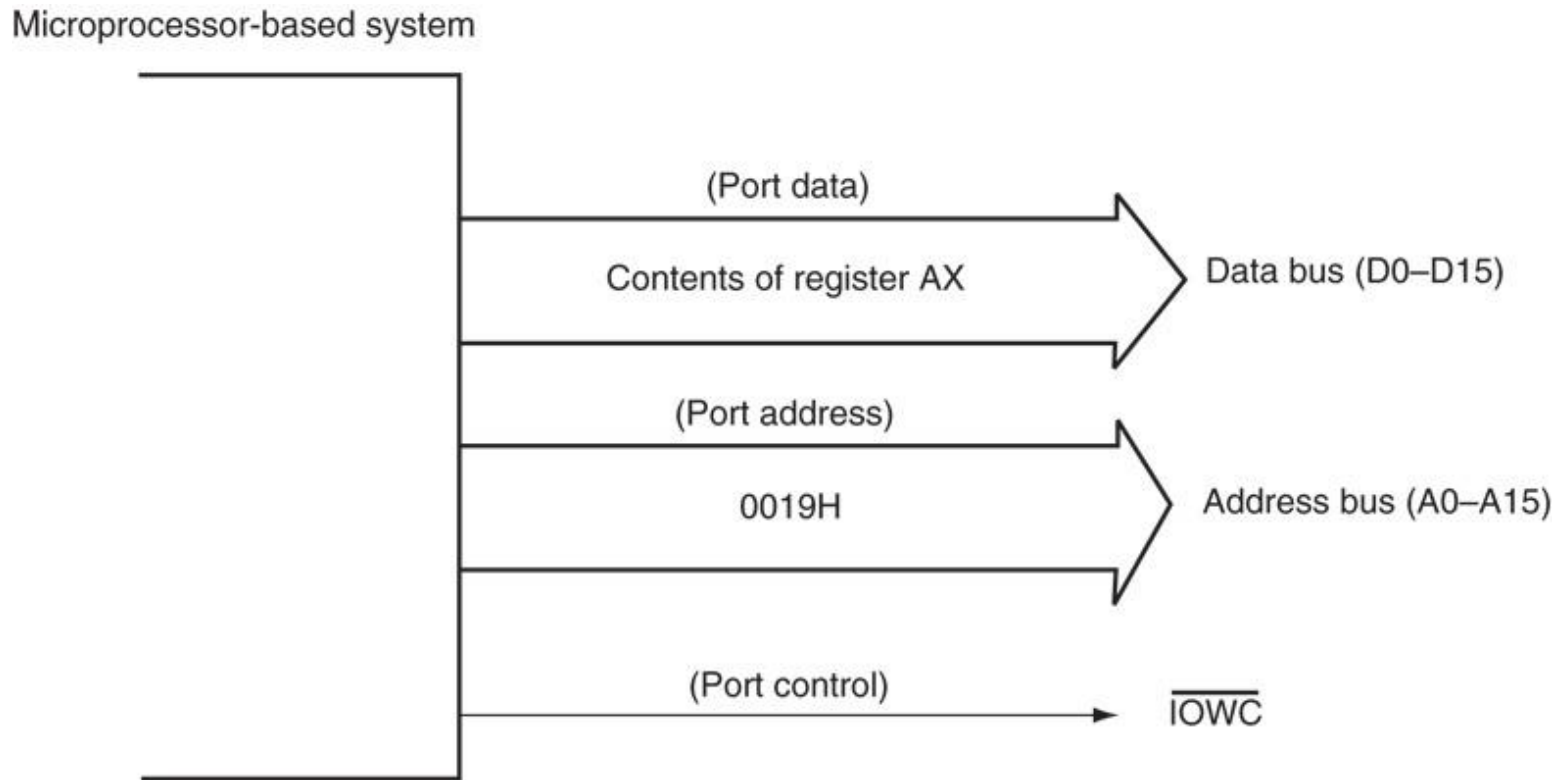
# IN and OUT

- **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, e.g., **IN AL, 19H**
  - an OUT transfers data from AL, AX, or EAX to an external I/O device, e.g., **OUT 32H, AX**
- 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, e.g., **IN AL, 12H, OUT 25H, AX**
  - port number is a byte-immediate value (00h to FFh) following the instruction's opcode
- *Variable-port addressing* allows data transfers between AL, AX, or EAX and a 16-bit port address, e.g., **IN AL, DX, OUT DX, AX**
  - the I/O port number is stored in register DX (0000h to FFFFh), which can be changed (varied) during the execution of a program.

**Figure 4–20** illustrates the execution of the **OUT 19H,AX** instruction, which transfers the contents of AX to I/O port 19H.



- The source register used in OUT instruction determines the size of the port (8, 16, or 32 bits).

# In and OUT instructions

<i>Assembly Language</i>	<i>Operation</i>
{ IN AL,p8	8 bits are input to AL from I/O port p8
{ IN AX,p8	16 bits are input to AX from I/O port p8
{ IN EAX,p8	32 bits are input to EAX from I/O port p8
IN AL,DX	8 bits are input to AL from I/O port DX
IN AX,DX	16 bits are input to AX from I/O port DX
IN EAX,DX	32 bits are input to EAX from I/O port DX
{ OUT p8,AL	8 bits are output to I/O port p8 from AL
{ OUT p8,AX	16 bits are output to I/O port p8 from AX
{ OUT p8,EAX	32 bits are output to I/O port p8 from EAX
OUT DX,AL	8 bits are output to I/O port DX from AL
OUT DX,AX	16 bits are output to I/O port DX from AX
OUT DX,EAX	32 bits are output to I/O port DX from EAX

Note: p8 = an 8-bit I/O port number (0000H to 00FFH) and DX = the 16-bit I/O port number (0000H to FFFFH) held in register DX.

- A program that clicks the speaker in the computer appears in following example.
- The speaker (in DOS only) is controlled by accessing **I/O port 61H**. If the rightmost 2 bits of this port are set (11) and then cleared (00), a click is heard on the speaker.

```
.MODEL TINY                ;select tiny model
.CODE                      ;start code segment
.STARTUP                   ;start program
    IN AL,61H              ;read I/O port 61H
    OR AL,3                ;set rightmost two bits
    OUT 61H,AL             ;speaker on
    MOV CX,8000H           ;load delay count
L1:
    LOOP L1                ;time delay
    IN AL,61H              ;speaker off
    AND AL,0FCH
    OUT 61H,AL
.EXIT
END
```





# MOVSX and MOVZX

- The MOVZX (**move and zero-extend**) and MOVSX (**move and sign-extend**) instructions are found in the 80386–Pentium 4 instruction sets. For example
  - if an 8-bit 34H is zero-extended into a 16-bit number, it becomes 0034H.
  - if an 8-bit 84H is sign-extended into a 16-bit number, it becomes FF84H.

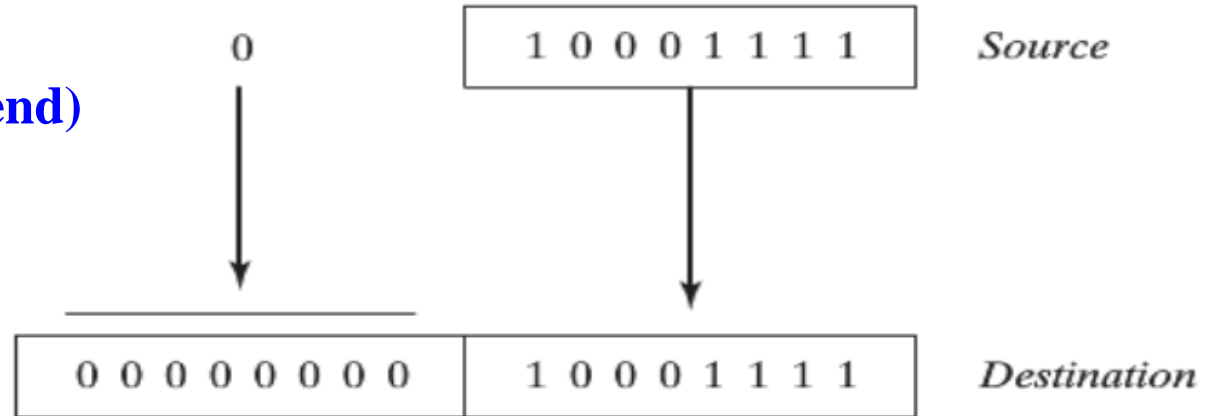
# MOVSX and MOVZX

Using MOVZX to copy a byte into a 16-bit destination.

## MOVZX

(move and zero-extend)

- `mov AL, 8FH`
- `movzx AX, AL`

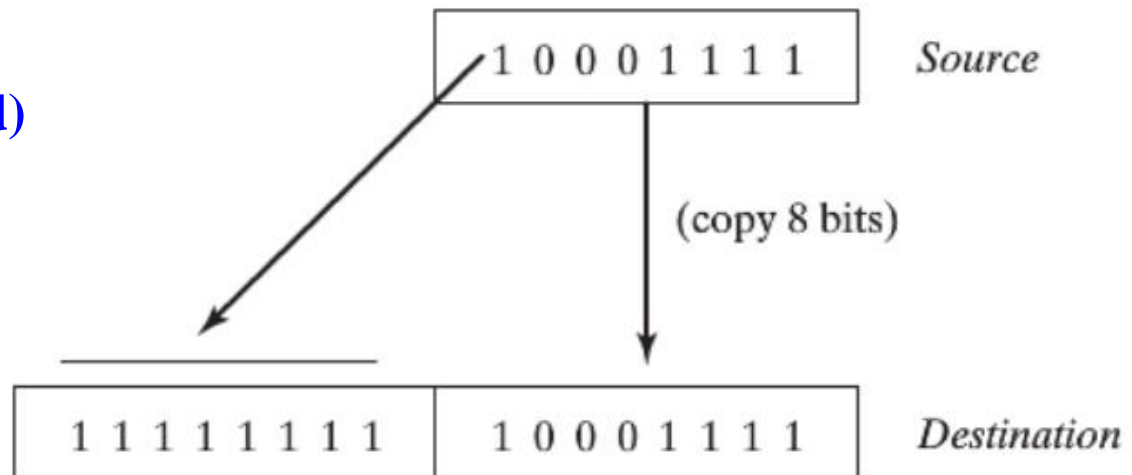


Using MOVSX to copy a byte into a 16-bit destination.

## MOVSX

(move and sign-extend)

- `mov AL, 8FH`
- `movsx AX, AL`



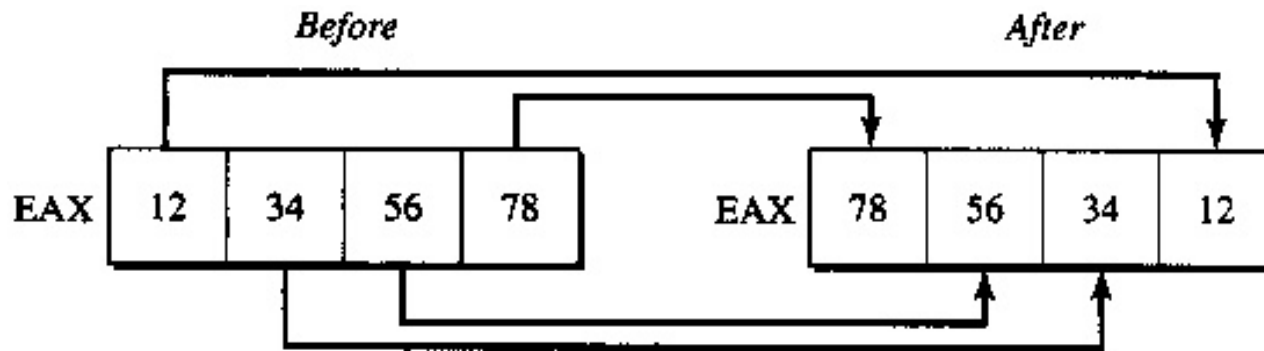
# MOVSX and MOVZX

<i>Assembly Language</i>	<i>Operation</i>
MOVSX CX,BL	Sign-extends BL into CX
MOVSX ECX,AX	Sign-extends AX into ECX
MOVSX BX,DATA1	Sign-extends the byte at DATA1 into BX
MOVSX EAX,[EDI]	Sign-extends the word at the data segment memory location addressed by EDI into EAX
MOVSX RAX,[RDI]	Sign-extends the doubleword at address RDI into RAX (64-bit mode)
MOVZX DX,AL	Zero-extends AL into DX
MOVZX EBP,DI	Zero-extends DI into EBP
MOVZX DX,DATA2	Zero-extends the byte at DATA2 into DX
MOVZX EAX,DATA3	Zero-extends the word at DATA3 into EAX
MOVZX RBX,ECX	Zero-extends ECX into RBX

# BSWAP

- The BSWAP (byte swap) instruction **reverses the byte order** in a 32-bit or 64-bit register operand.
- The BSWAP instruction is used to **convert data between the big and little endian** forms.
- Takes the contents of any 32-bit register and swaps the first byte with the fourth, and the second with the third.

- For example, the **BSWAP EAX** instruction with  $EAX = 12345678H$  swaps bytes in EAX, resulting in  $EAX = 78563412H$ .



- The proper byte swapping without using BSWAP is the following:

```
XCHG AH,AL  
ROR EAX,16  
XCHG AH,AL
```

; rotate right of EAX

- In 64-bit operation for a quadword, bits 7:0 are exchanged with bits 63:56, bits 15:8 with bits 55:48, bits 23:16 with bits 47:40, and bits 31:24 with bits 39:32.
- In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix permits access to additional registers (R8-R15).

Instruction	Opcode	Description
BSWAP reg32	<b>0F C8</b> + <b>rd</b>	Reverses the byte order of a 32-bit register
BSWAP reg64	<b>REX.W</b> + <b>0F C8</b> + <b>rd</b>	Reverses the byte order of a 64-bit register

Instruction	Opcode	Description
BSWAP reg32	0F C8 + <b>rd</b>	Reverses the byte order of a 32-bit register
BSWAP reg64	REX.W + 0F C8 + <b>rd</b>	Reverses the byte order of a 64-bit register

- In “Opcode” column, **+rd** indicates the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte, e.g.,

– BSWAP RAX ; 48 0F C8

REX prefix opcode

– BSWAP RCX ; 48 0F C9

– BSWAP R8 ; 49 0F C8

– BSWAP R9 ; 49 0F C9

- The result of applying the BSWAP instruction to a 16-bit register is undefined.
- To swap the bytes of a 16-bit register, use the XCHG instruction.
- For example, to swap the bytes of AX, use XCHG AL, AH.



# CMOV

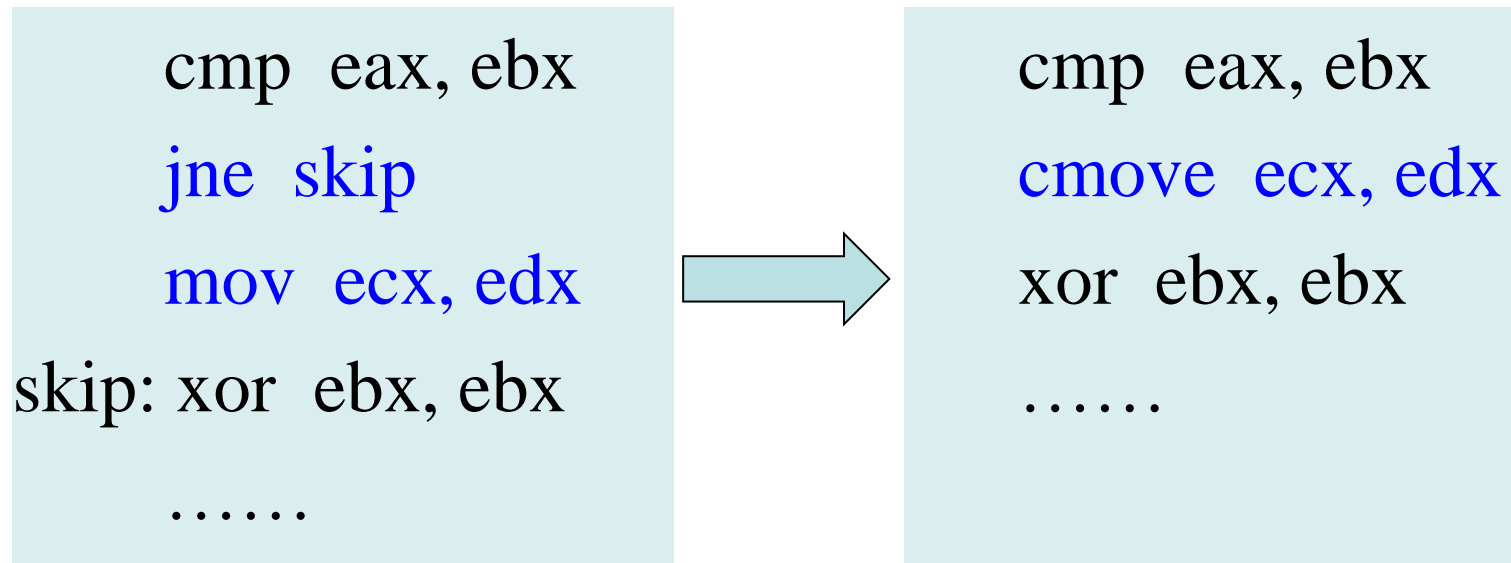
- Each of the **CMOVcc** instructions performs a move operation if the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) are in a specified state (or condition).
- A condition code (cc) is associated with each instruction to indicate the condition being tested for.
  - These move the data only if the condition is true
  - If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.

- For example, **CMOVZ** instruction moves data only if the result from some prior instruction was a zero.
- Destination is limited to only a 16, 32, or 64 bit register, but the source can be a 16, 32, or 64 bit register or memory location
- Because this is a new instruction, you cannot use it with the assembler unless the .686 switch is added to the program

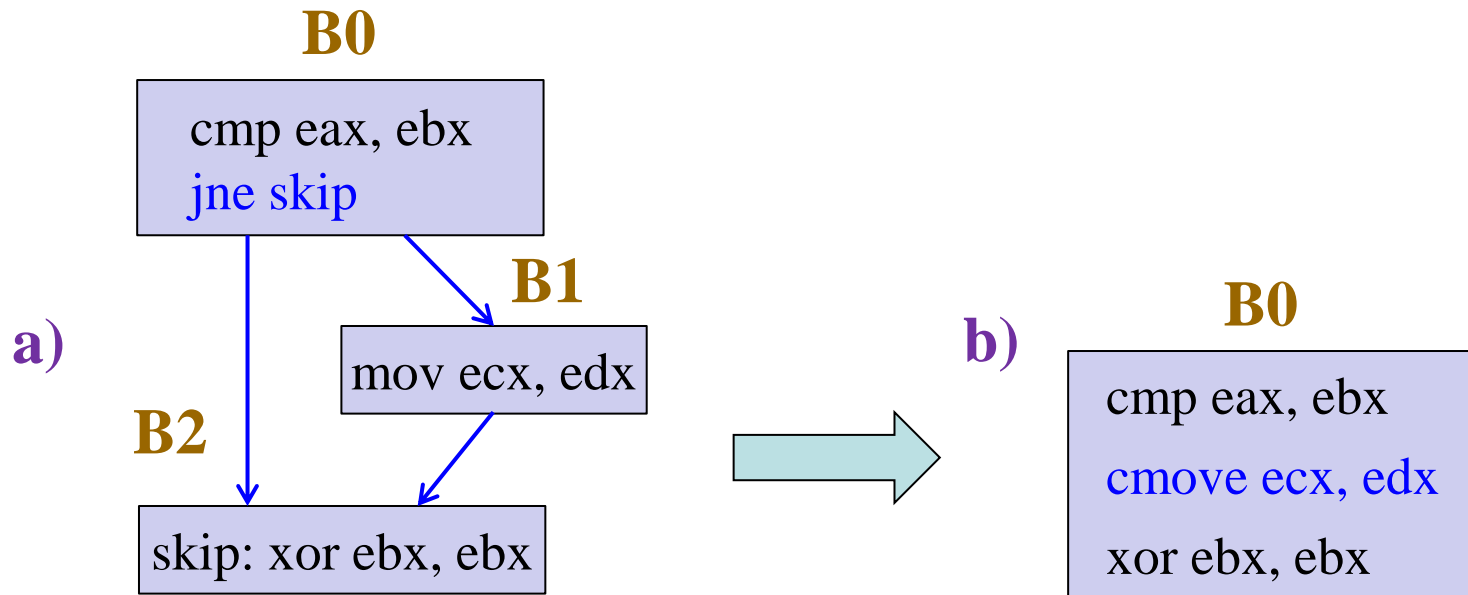
# the conditional move instructions

<i>Assembly Language</i>	<i>Flag(s) Tested</i>	<i>Operation</i>
CMOVB	C = 1	Move if below
CMOVAE	C = 0	Move if above or equal
CMOVBE	Z = 1 or C = 1	Move if below or equal
CMOVA	Z = 0 and C = 0	Move if above
CMOVE or CMOVZ	Z = 1	Move if equal or move if zero
CMOVNE or CMOVNZ	Z = 0	Move if not equal or move if not zero
CMOVL	S != O	Move if less than
CMOVLE	Z = 1 or S != O	Move if less than or equal
CMOVG	Z = 0 and S = O	Move if greater than
CMOVGE	S = O	Move if greater than or equal
CMOVS	S = 1	Move if sign (negative)
CMOVNS	S = 0	Move if no sign (positive)
CMOVC	C = 1	Move if carry
CMOVNC	C = 0	Move if no carry
CMOVO	O = 1	Move if overflow
CMOVNO	O = 0	Move if no overflow
CMOVPE or CMOVPE	P = 1	Move if parity or move if parity even
CMOVNP or CMOVPO	P = 0	Move if no parity or move if parity odd

- The purpose of CMOV is to avoid a branch.
- When a CPU sees the branch (e.g., JNE) it will take a guess about whether the branch will be taken or not taken, and then start speculatively executing instructions.



- If the guess is wrong there's a performance penalty, because the CPU has to discard any speculatively executed work and then start fetching and executing the correct path.
- For a conditional move (e.g. `CMOVE eax,edx`) the CPU doesn't need to guess which code will be executed and the cost of a mispredicted branch is avoided.



control flow graph for the program a) original b) using CMOV

- Additionally, CMOVcc instructions **convert control dependence to data dependence** and instructions in multiple paths are merged into a basic block. This makes basic blocks contain more instructions and **extends instruction scheduling space**.

## 4-5 SEGMENT OVERRIDE PREFIX

- May be added to almost any instruction in any memory-addressing mode
  - allows the programmer to deviate from the default segment
  - only instructions that cannot be prefixed are jump and call instructions using the code segment register for address generation
  - Additional byte appended to the front of an instruction to select alternate segment register

## instructions that include segments

<i>Assembly Language</i>	<i>Segment Accessed</i>	<i>Default Segment</i>
MOV AX,DS:[BP]	Data	Stack
MOV AX,ES:[BP]	Extra	Stack
MOV AX,SS:[DI]	Stack	Data
MOV AX,CS:LIST	Code	Data
MOV ES:[SI],AX	Extra	Data
LODS ES:DATA1	Extra	Data
MOV EAX,FS:DATA2	FS	Data
MOV GS:[ECX],BL	GS	Data



# 4–6 ASSEMBLER DETAIL

- The assembler can be used in two ways:
  - with models unique to a particular assembler
  - with full-segment definitions that allow complete control over the assembly process and are universal to all assemblers
- This section of the text presents both methods and explains how to organize a program's memory space by using the assembler.
- It also explains the purpose and use of some of the more important directives used with this assembler.

# Directives vs Instructions

- Assembly language statements are either directives or instructions.
- **Directives:** tell assembler how to do
  - generating machine code, allocating storage, etc.
  - only used at assembly time and don't result in any code generation themselves.
- **Instructions:** tell CPU what to do
  - assembled into machine code and eventually linked into the final executable code.
  - executed by the CPU at run time.

# Directives in MASM

- Indicate how an operand or section of a program is to be processed by the assembler.
  - some generate and store information in the memory; others do not
- The **DB** (define byte) directive stores bytes of data in the memory.
- **BYTE PTR** indicates the size of the data referenced by a pointer or index register.

# Directives in MASM

- Data Allocation
  - DB, DW, DD, DQ, DT
- Structure
  - STRUCT, ENDS
- Code Labels
  - ALIGN, ORG
- Segment
  - SEGMENT, ENDS, ASSUME
- Simplified Segment
  - .CODE, .DATA, .STACK, .MODEL, .EXIT
- Procedures
  - PROC, ENDP
- Macros
  - MACRO, ENDM
- Miscellaneous
  - EQU, INCLUDE

Directives Reference: <https://docs.microsoft.com/en-us/cpp/assembly/masm/directives-reference?view=msvc-160>

# ***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.

- The **DUP** directive allows multiple initializations to the same value. e.g., 100 DUP(6) reserves 100 bytes of 6.
- Memory is reserved for use by using a (?) as an operand for a DB, DW, or DD directive. The assembler sets aside a location and does not initialize it to any specific value.
- The **ALIGN** directive aligns the next data element or instruction on an address that is a multiple of its parameter. The parameter must be a power of 2 (e.g., 1, 2, 4, 8) that is less than or equal to the segment alignment.

LIST_SEG	SEGMENT		
DATA1	DB	1, 2, 3	;define bytes
	DB	45H	;hexadecimal
DATA3	DD	300H	;define doubleword
	DD	2.123	;real
	DD	3.34E+12	;real
LISTA	DB	?	;reserve 1 byte
LISTB	DB	10 DUP(?)	;reserve 10 bytes
	ALIGN	2	;set word boundary
LISTC	DW	100H DUP(0)	;reserve 100H words
LISTD	DD	22 DUP(?)	;reserve 22 doublewords

- 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

- **Problem:** Please determine the value in the register AX after the instruction is executed.

.data

DB 33H, 34H, 0AH, 06H

DW 1B7CH, 674CH, 07H, '12', '1'

.code

mov ax, @data

mov ds, ax

xor si, si

mov ax, [si] ax = 3433H

mov ax, [si+4] ax = 1B7CH

mov ax, [si+5] ax = 4C1BH

mov ax, [si+8] ax = 0007H

mov ax, [si+10] ax = 3231H

.exit

Memory: 33, 34, 0A, 06, 7C, 1B, 4C, 67, 07, 00, 31, 32, 31

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12



# ***ASSUME, EQU, and ORG***

- Equate directive (**EQU**) equates a numeric, ASCII, or label to another label. The EQU directive is used for defining constants.
- The syntax of the EQU directive
  - **CONSTANT\_NAME** **EQU** **expression**
- Equates make a program clearer and simplify debugging. For example

```
TEN      EQU    10
```

```
NINE     EQU    9
```

```
MOV AL, TEN
```

```
ADD AL, NINE
```

# ***ASSUME, EQU, and ORG***

- The assembler can only assign either a byte, word, or doubleword address to a label.
- To assign a byte label to a word, use **THIS** directive.
- The THIS directive always appears as THIS BYTE, THIS WORD, THIS DWORD, or THIS QWORD.

- The **ORG** (origin) statement can **change the starting offset address** of the data in the data segment or code in the code segment.
- At times, the origin of data or the code must be assigned to an absolute offset address with the ORG statement. For example, Boot Sector Entry must be assigned to 07c00h.
- **ASSUME** tells the assembler what names have been chosen for the code, data, extra, and stack segments.

*i*

```
ORG      300H
```

← use **ORG** directive to set the location

use **THIS** directive to assign a word address label to a byte label

DATA SEG ENDS

use **SEGMENT** directive to define a program segment

```
ASSUME CS:CODE SEG, DS:DATA SEG
```

MOV BL, DATA1

MOV AX, DATA2

MOV BH, DATA1+1

```
CODE SEG      ENDS
```

# ***PROC and ENDP***

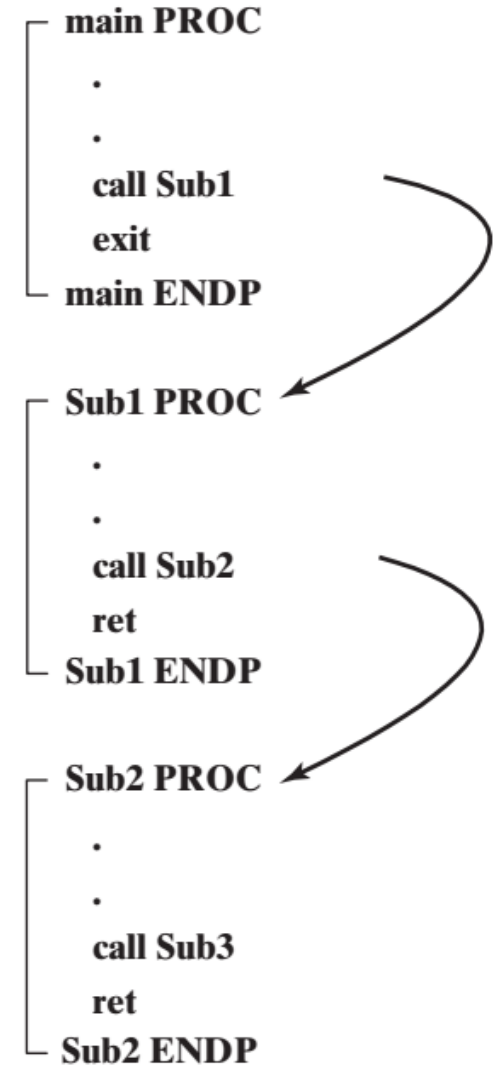
- The **PROC** and **ENDP** directives indicate the start and end of a procedure (subroutine) that must be assigned a name.

*name* **PROC** [*near/far*]

statements

ret

*name* **ENDP**



Nested Procedure Calls

- The PROC directive, which indicates the start of a procedure, must also be followed with a **NEAR** or **FAR** (valid only in 32-bit system).
  - 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*
- **Local** defines a procedure that is only used by the current program.
- The term **global** denotes a procedure that can be used by any program.

- A procedure named *SumOf* calculates the sum of three 32-bit integers by passing register arguments to the procedure.

```
SumOf PROC
    add    eax,ebx
    add    eax,ecx
    ret
SumOf ENDP
```

- Three integers are assigned to EAX, EBX, and ECX.
- The procedure returns the sum in EAX.

```
.data
theSum  DWORD  ?
.code
main PROC
```

```
    mov    eax,10000h
    mov    ebx,20000h
    mov    ecx,30000h
    call   Sumof
    mov    theSum,eax
```



Before calling SumOf, values are assigned to EAX, EBX, and ECX.

```
; argument
; argument
; argument
; EAX = (EAX + EBX + ECX)
; save the sum
```

After the CALL, the sum in EAX are copied to “theSum” variable.

# ***MACRO and ENDM***

- **MACRO** and **ENDM** directives indicate a macro (a named block of assembly language statements).
- When you invoke a macro procedure, a copy of its code is inserted directly into the program at the location where it was invoked.
- This type of automatic code insertion is also known as *inline expansion*.

*name* **MACRO** [*para1,para2,...*]

statements

**ENDM**



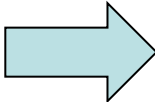
- A macro named `mPuchar` receives a input and displays it on the console by calling `WriteChar`.

```

mPuchar MACRO char
    push    eax
    mov     al,char    ; passing arguments to the procedure
    call    WriteChar
    pop     eax
ENDM

```

- The statement “`mPuchar 'A'`” invokes `mPuchar` and passes it the letter A.
- The assembler’s preprocessor expands the statement into the following code:

.....			
mPuchar 'A'		{	push    eax
	inline		mov     al, 'A'
	expansion		call    WriteChar
.....			pop     eax

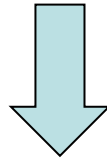
- A macro can also be used in data segment. For example, define a macro for GDT Descriptor.

Descriptor MACRO

Base, Limit, Attr

```
dw Limit & 0FFFFh      ; Limit 1 (2 bytes)
dw Base & 0FFFFh       ; Base 1      (2 bytes)
db (Base >> 16) & 0FFh  ; Base 2      (1 byte)
dw ((Limit >> 8) & 0F00h) | (Attr & 0F0FFh) ; Attr 1 + Limit 2 + Attr 2
db (Base >> 24) & 0FFh   ; Base 3 (1 byte)
```

ENDM ; total 8 bytes



GDT SEGMENT

```
Null_desc: Descriptor 0, 0, 0
Normal_desc: Descriptor 0, 0ffffh, DA_DRW
.....
```

} Allocation of  
GDT Descriptor

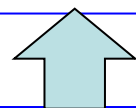
GDT ENDS

# INCLUDE

- **INCLUDE** directive is used to insert source code from the source file given by filename into the current source file during assembly.
- Syntax: **INCLUDE** filename

```
%macro Descriptor 3
    dw  %2 & 0FFFFh          ; 段界限 1
    dw  %1 & 0FFFFh          ; 段基址 1
    db  (%1 >> 16) & 0FFh     ; 段基址 2
    dw  ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh) ; 属性
    db  (%1 >> 24) & 0FFh     ; 段基址 3
%endmacro ; 共 8 字节
```

Macro for GDT  
Descriptor  
(lib.inc)



include lib.inc

```
[SECTION .gdt]
; GDT
;
; 段基址, 段界限, 属性
LABEL_GDT:      Descriptor 0, 0, 0 ; 空描述符
LABEL_DESC_NORMAL: Descriptor 0, 0ffffh, DA_DRW ; Normal
LABEL_DESC_FLAT_C: Descriptor 0, 0ffffh, DA_CR | DA_32 | DA_LIMIT_4K ; 0
LABEL_DESC_FLAT_RW: Descriptor 0, 0ffffh, DA_DRW | DA_LIMIT_4K ; 0
LABEL_DESC_CODE32: Descriptor 0, SegCode32Len - 1, DA_CR | DA_32
LABEL_DESC_CODE16: Descriptor 0, 0ffffh, DA_C ; 非一致代码
LABEL_DESC_DATA:   Descriptor 0, DataLen - 1, DA_DRW ; Da
LABEL_DESC_STACK:  Descriptor 0, TopOfStack, DA_DRWA | DA_32 ; St
LABEL_DESC_VIDEO:  Descriptor 0B8000h, 0ffffh, DA_DRW ; 显存首地址
```

Allocation of  
GDT Descriptor

# Memory Organization

- **Memory organization** defines the memory-related attributes for the software, including size of code, data pointer, instructions encoding, segment combine type, and segment loading order, etc.
- The assembler uses two basic formats to define memory organization:
  - one method uses **full-segment definitions**
  - the other method uses **models** (.MODEL)

# Memory Organization

- The **full-segment definitions** offer better control over the assembly language task and are recommended for complex programs.
- There are many **memory models** available to the MASM assembler, ranging from tiny to huge, which control how the segment registers are used and the default size of pointers.
- The **.MODEL** directive is used to determine the size of code and data pointers.

# ***Full-Segment Definitions***

- Full-segment definitions use **SEGMENT**, **ENDS** and **ASSUME** directives to define segment and inform assembler and linker.

*name* **SEGMENT** [*readonly*] [*align*] [*combine*] [*use*] [*'class'*]

statements

*name* **ENDS**

*cseg* **SEGMENT** readonly word use32 'code'

mov ax, 10

inc ax

ret

*cseg* **ENDS**

# ***Controlling Segments with the ASSUME Directive***

- The segment directive doesn't tell what type of segment. The **assume** directive provides this information to the assembler.
- The assume directive takes the following form:  
**assume** [CS:seg,] [DS:seg,] [ES:seg,]  
          [FS:seg,] [GS:seg,] [SS:seg]
- Examples of valid assume directives:
  - assume DS:DSEG
  - assume CS:CSEG, DS:DSEG, ES:DSEG, SS:SSEG
  - assume CS:CSEG, DS:NOTHING

- When the assembler encounters an instruction (e.g., `mov var,0`), the first thing it does is to determine var's segment.
- If var is not declared in one of the currently assumed segments, then the assembler generates an error claiming that the program cannot access that variable.
- The ideal place to put assume directives is before all procedures in a program. Since segment pointers to declared segments in a program rarely change except at procedure entry and exit.



Example 4–19  
illustrates the same  
program in example  
4–18 using full  
segment definitions.

```

STACK_SEG      SEGMENT      'STACK' ← define stack
                DW          100H DUP(?)      segment

STACK_SEG      ENDS

DATA_SEG       SEGMENT      'DATA' ← define data
LISTA  DB      100 DUP(?)      segment

LISTB  DB      100 DUP(?)

DATA_SEG       ENDS

CODE_SEG       SEGMENT      'CODE' ← define code
                ASSUME CS:CODE_SEG, DS:DATA_SEG ← segment
                ASSUME SS:STACK_SEG

MAIN  PROC     FAR
                MOV     AX,DATA_SEG      ;load DS and ES
                MOV     ES,AX
                MOV     DS,AX
                CLD                     ;save data
                MOV     SI,OFFSET LISTA
                MOV     DI,OFFSET LISTB
                MOV     CX,100
                REP     MOVSB
                MOV     AH,4CH           ;exit to DOS
                INT     21H
MAIN  ENDP
CODE_SEG      ENDS
                END     MAIN

```

Segment name  
DATA\_SEG can be  
used to load segment  
address.

# ***Models*** (32-bit MASM only)

- **Memory models** are unique to MASM.
- Special directives such as **@DATA**, **@STACK**, **@CODE** are used to identify various segments.
- **.MODEL** directive includes six memory models for the real mode in MASM, namely **tiny**, **small**, **compact**, **medium**, **large**, and **huge**.
- One model (flat) is available for the protected model.
- **.MODEL** is not used in MASM for x64.

# Memory Models

Model	Data	Code	Definition
Tiny	near		a single segment, containing both code and data (CS=DS=SS).
Small	near	near	one code segment and one data segment (DS=SS)
Medium	near	far	multiple code segments and a single data segment (DS=SS)
Compact	far	near	single code segment, multiple data segments
Large	far	far	multiple code and data segments
Huge	huge	far	multiple code and data segments; single array may be larger than a single segment (>64 KB)

- Choose the smallest memory model available that can contain the data and code, since **near** references operate more efficiently than **far** references.

Example 4–18 illustrates how the .MODEL statement defines the parameters of a short program that copies the contents of a 100-byte block of memory (LISTA) into a second 100- byte block of memory (LISTB).

### EXAMPLE 4–18

```

                                .MODEL SMALL           ;select small model
                                .STACK 100H           ;define stack
                                .DATA                 ;start data segment

0000 0064 [          LISTA  DB      100 DUP(?)
                        ??
                        ]
0064 0064 [          LISTB  DB      100 DUP(?)
                        ??
                        ]

                                .CODE                 ;start code segment

0000 B9 — ?          HERE:  MOV     AX,@DATA           ;load ES and DS
0003 8E C0           MOV     ES,AX
0005 8E D8           MOV     DS,AX
0007 FC             CLD                                ;move data
0008 BE 0000 R       MOV     SI,OFFSET LISTA
000B BF 0064 R       MOV     DI,OFFSET LISTB
000E B9 0064         MOV     CX,100
0011 F3/A4          REP     MOVSB

0013                .EXIT 0                          ;exit to DOS
                                END HERE

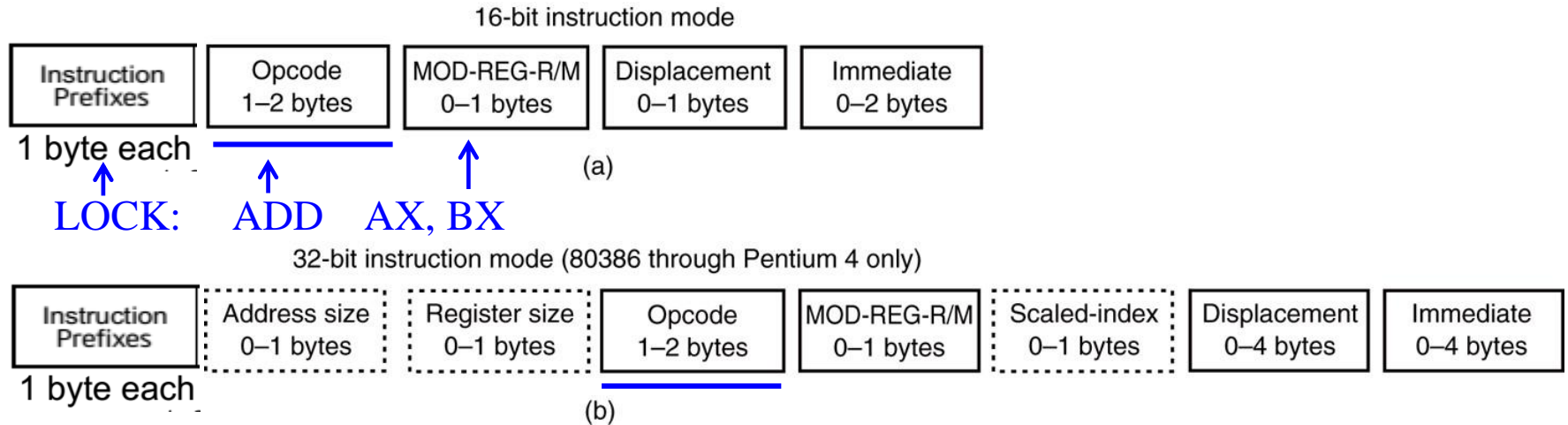
```

uses @DATA to load segment address.

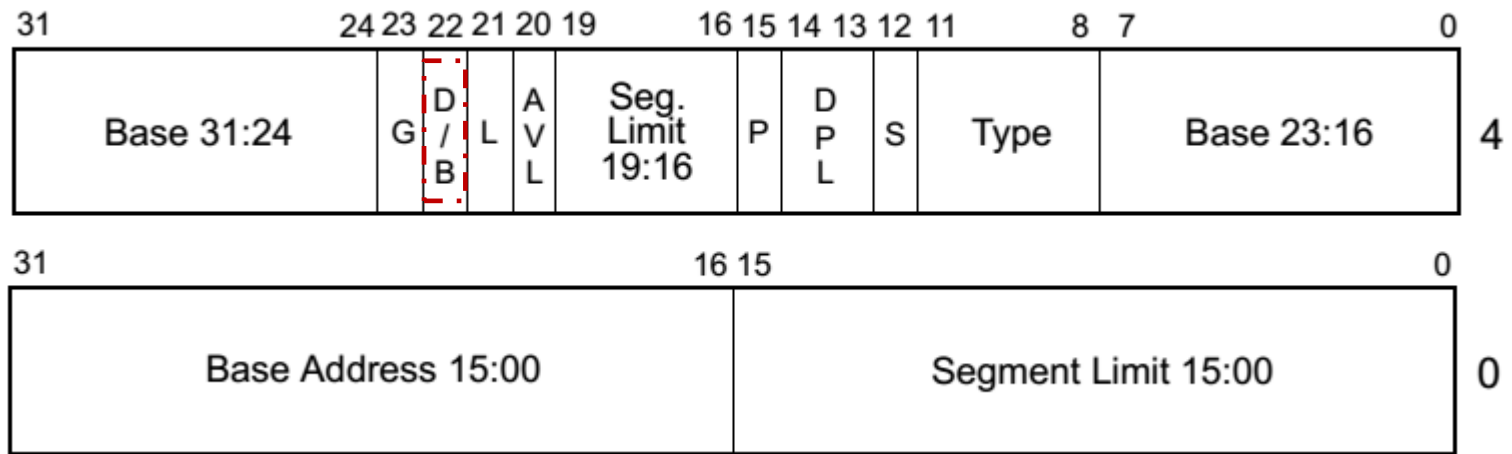
# Assignment

4-2, 3, 12, 13, 16, 21, 25-27, 31, 36, 38,  
40, 43, 49-51, 56, 60, 62

**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



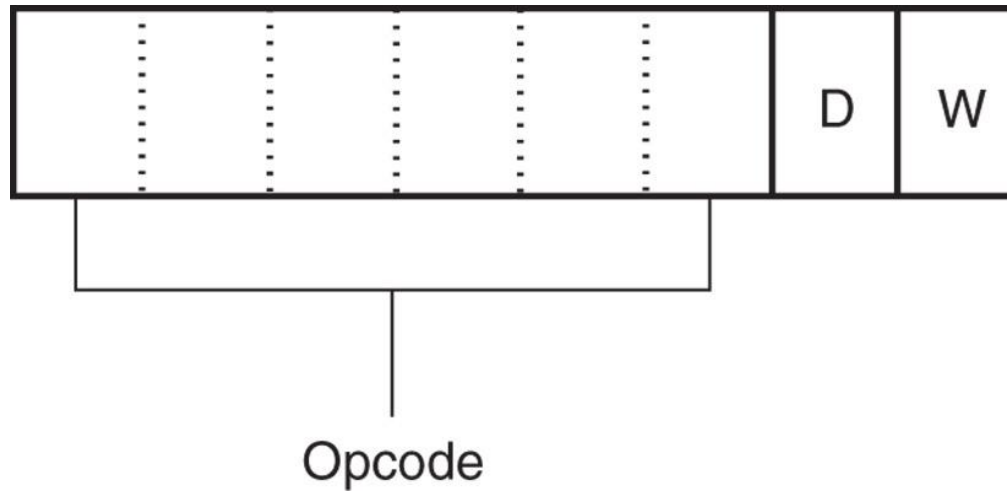
- D flag indicates the default length for effective addresses and operands referenced by instructions. If the flag is set, 32-bit addresses and 32 or 8-bit operands are assumed; if it is clear, 16-bit addresses and 16 or 8-bit operands are assumed.
- The instruction prefix 66H can be used to select an operand size other than the default, and the prefix 67H can be used select an address size other than the default.

# The Opcode-Byte 1

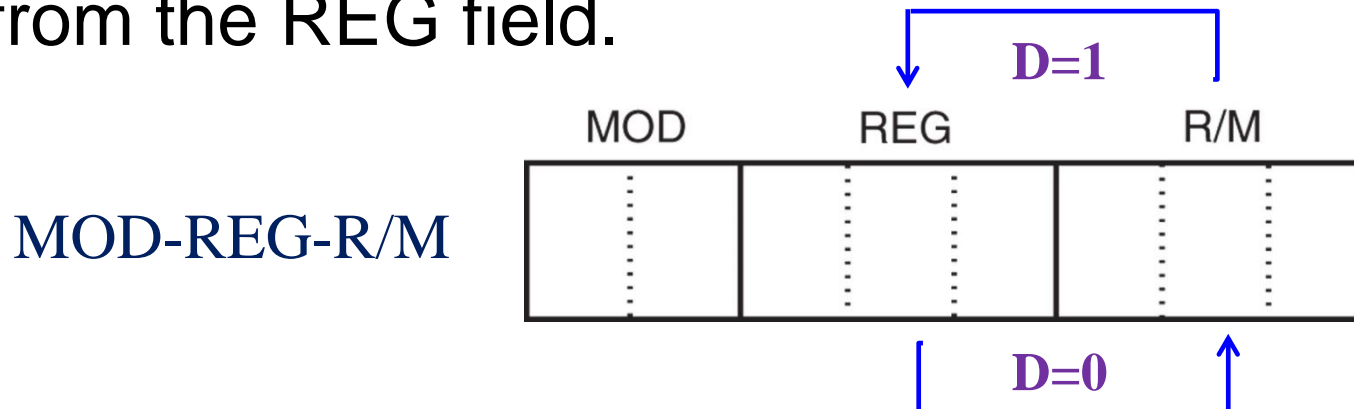
- Selects the operation (addition, subtraction, etc.,) performed by the microprocessor.
  - either 1 or 2 bytes long for most instructions
- Figure 4–2 illustrates the general form of the first opcode byte of many instructions.
  - first 6 bits of the first byte are the binary opcode
  - remaining 2 bits indicate the **direction** (D) of the data flow, and indicate the **data size** (W) (a byte or a word)



**Figure 4–2** Byte 1 of many machine language instructions, showing the position of the D- and W-bits.

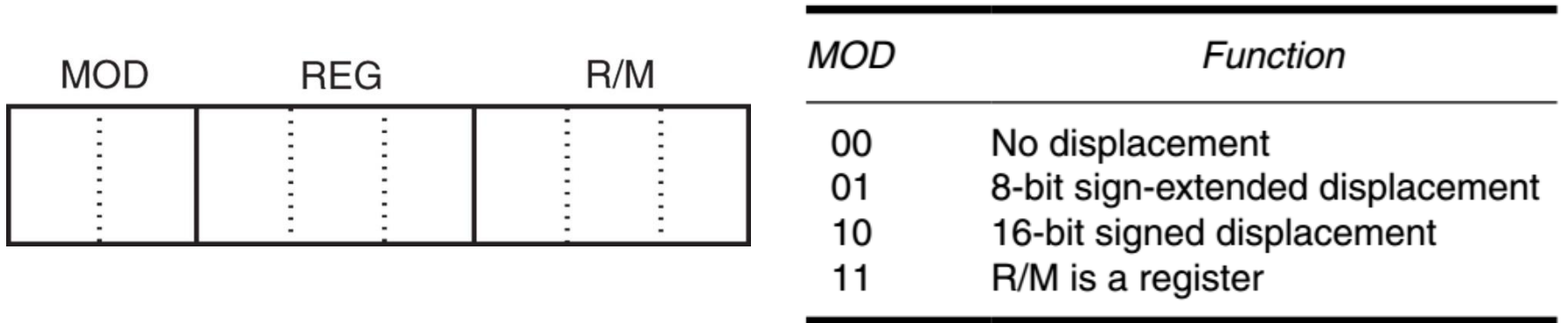


- If the direction bit  $D=1$ , data flow to the register REG field from the R/M field.
- If the direction bit  $D=0$ , data flow to the R/M field from the REG field.



# MOD Field-Byte 2

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



MOD field for the 16-bit instruction mode

- The **MOD field specifies the addressing mode** (MOD) for the selected instruction.
- The MOD field selects the type of addressing and whether a displacement is present with the selected type.

# ***MOD Field-Byte 2***

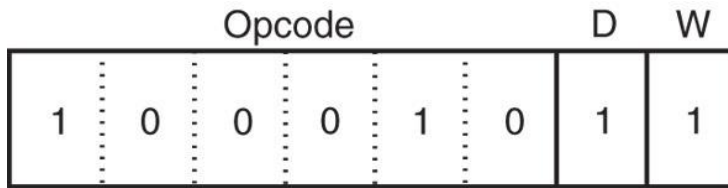
- Specifies addressing mode (MOD) and whether a displacement is present with the selected type.
  - If MOD field contains an 11, it selects the register-addressing mode
  - Register addressing specifies a register instead of a memory location, using the R/M field
- If the MOD field contains a 00, 01,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

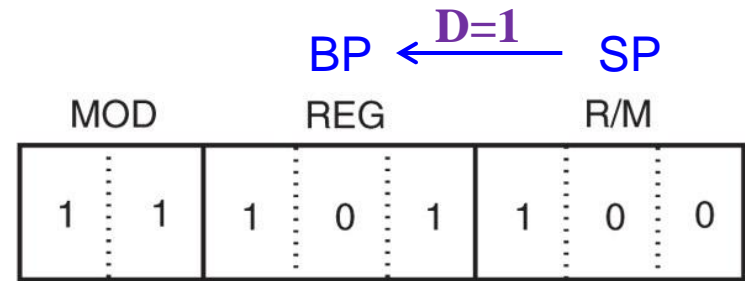
D = Transfer to register (REG)

W = Word

MOD = R/M is a register

REG = BP

R/M = SP



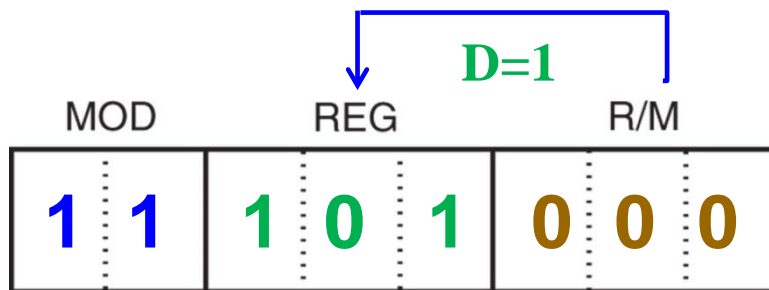
— opcode 100010: MOV

REG Code	W=1 (Word)
101	BP
100	SP

- 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

- 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.

- Suppose that a **668BE8H** instruction appears in an 80386, operated in the 16-bit instruction mode.
  - The first byte (66H) is the register-size **override prefix** that selects 32-bit register operands for the 16-bit instruction.
  - **10001011** (8BH): **opcode: 100010 (MOV); D=1; W=1**
  - **11101000** (E8H): **MOD=11; REG=101; R/M=000**

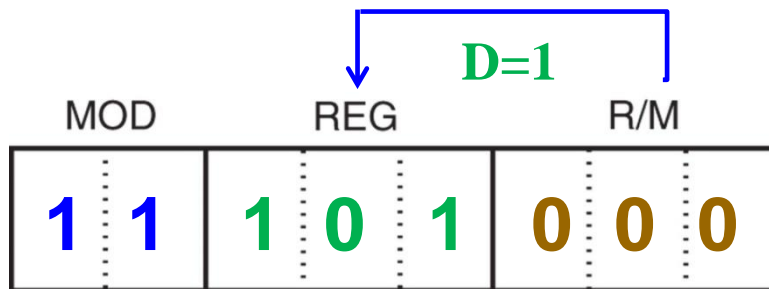


REG Code	W=1 (Doubleword)
000	EAX
101	EBP

- This instruction is a **MOV EBP,EAX**.



- The same instruction (**668BE8H**) becomes a **MOV BP,AX** instruction in the 80386 and above if it is operated in the 32-bit instruction mode.
  - The first byte (66H) is the register-size **override prefix** that selects 16-bit register operands for the 32-bit instruction.
  - 10001011** (8BH): **opcode: 100010 (MOV); D=1; W=1**
  - 11101000** (E8H): **MOD=11; REG=101; R/M=000**



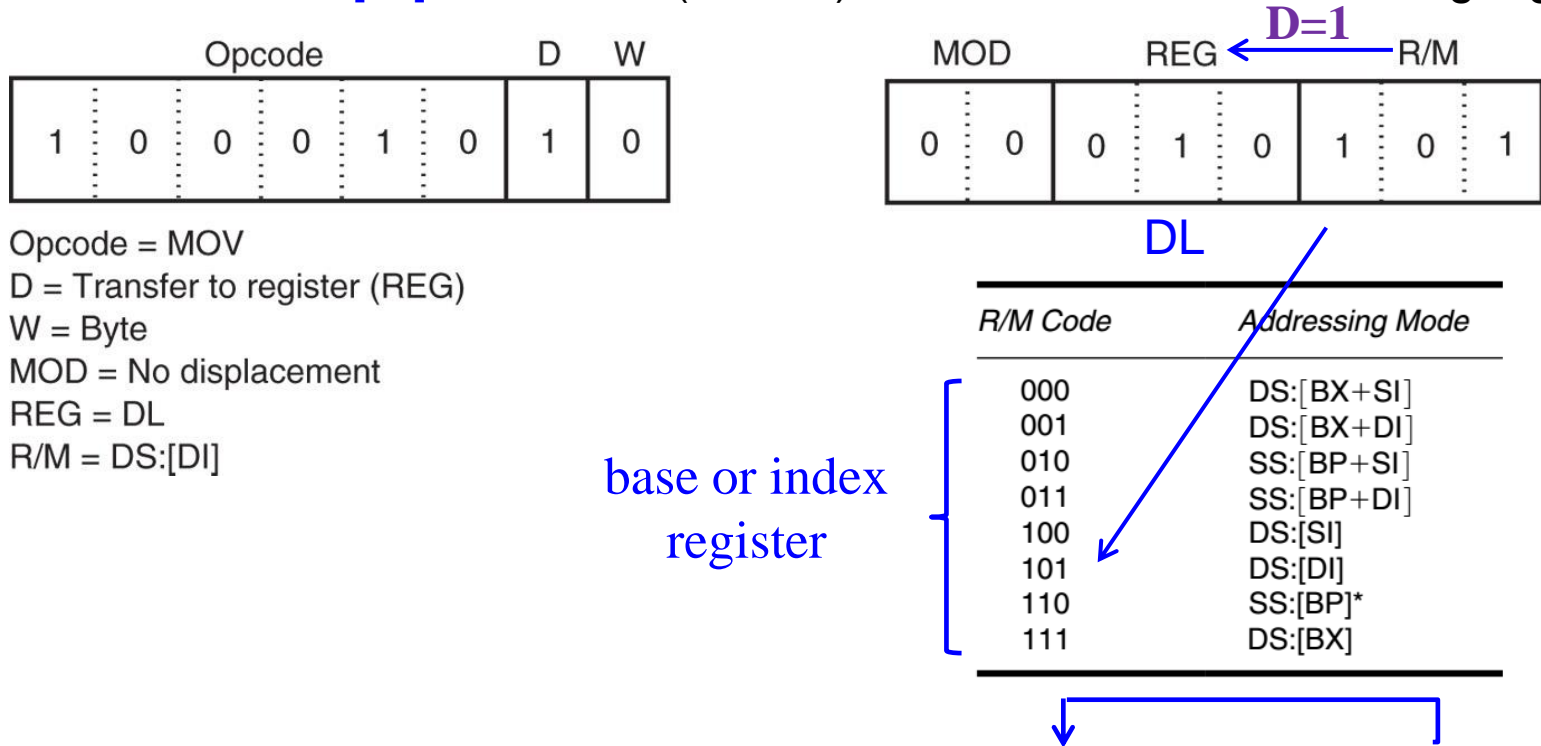
REG Code	W=1 (Word)
000	AX
101	BP

- This instruction is a **MOV BP,AX**.

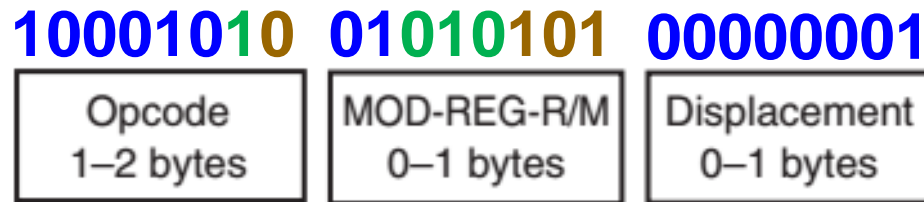
# ***R/M Memory Addressing***

- 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 (8A15H) converted to its machine language form.



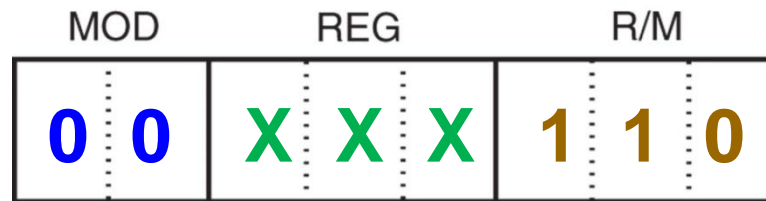
- If the instruction becomes 8A5501H
- first 1 byte of the instruction remains the same
- the MOD field changes to 01 for 8-bit displacement, the instruction changes to **MOV DL, [DI+1]**



# ***Special Addressing Mode***

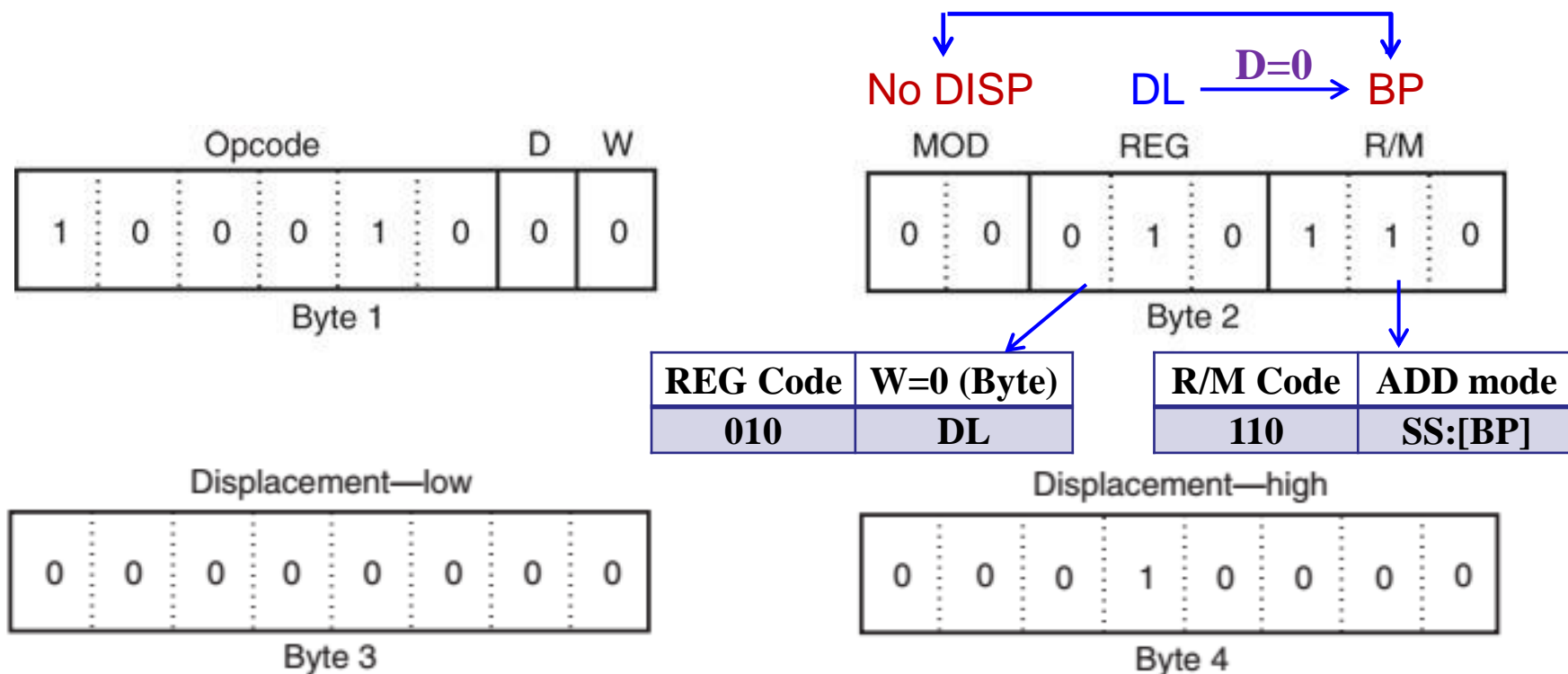
- A special addressing mode occurs when memory data are referenced by only the displacement mode of addressing for 16-bit instructions.
- Examples are the **MOV [1000H],DL** and **MOV NUMB,DL** instructions.
  - first instruction moves contents of register DL into data segment memory location 1000H
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

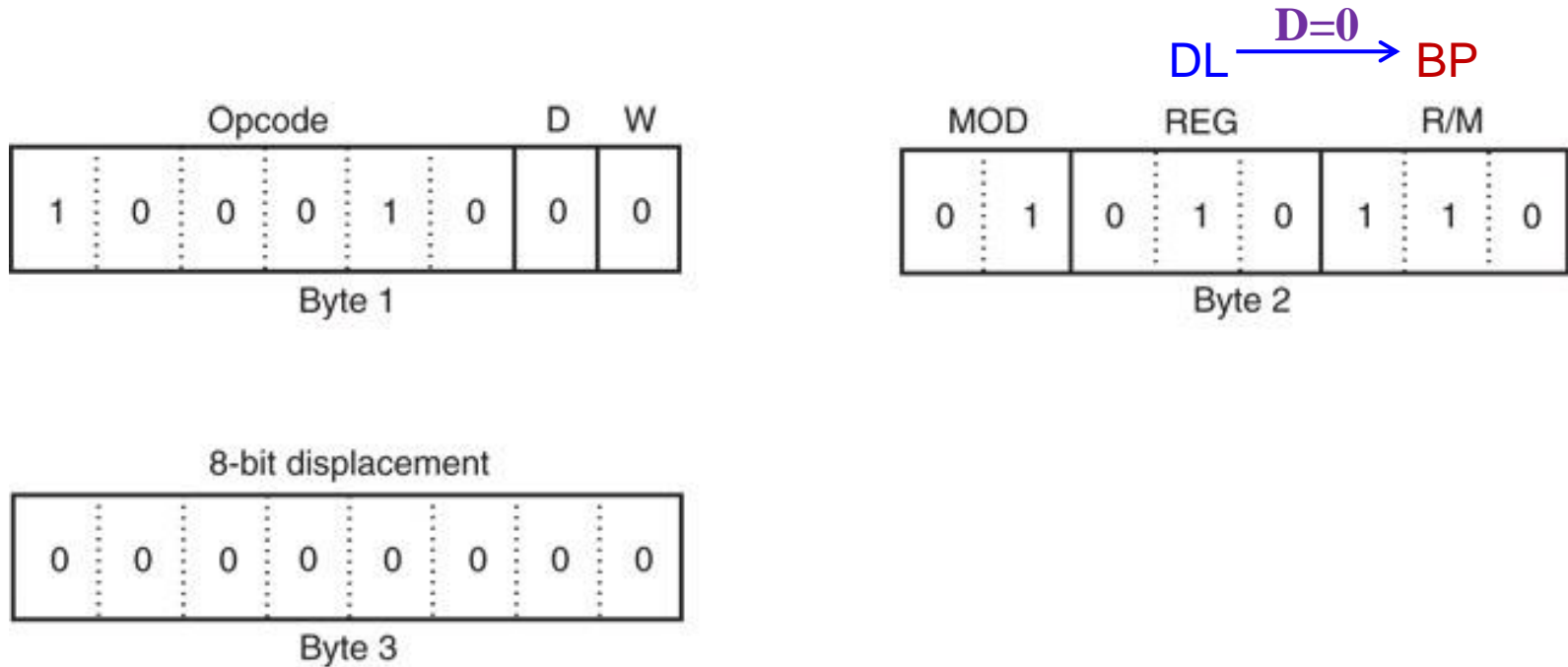
**Figure 4–6** The `MOV [1000H],DL` instruction uses the special addressing mode.



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]

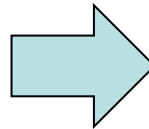
Displacement = 00H

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

# 32-Bit Addressing Modes

- Found in 80386 and above.
  - By running in 32-bit instruction mode or
  - in 16-bit mode by using **address-size prefix 67H**

<i>R/M Code</i>	<i>Addressing Mode</i>
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]*
111	DS:[BX]



<i>R/M Code</i>	<i>Function</i>
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	Uses scaled-index byte
101	SS:[EBP]*
110	DS:[ESI]
111	DS:[EDI]

16-bit instruction mode uses base or index register to address the memory, e.g., **MOV DX,[BX + DI]**

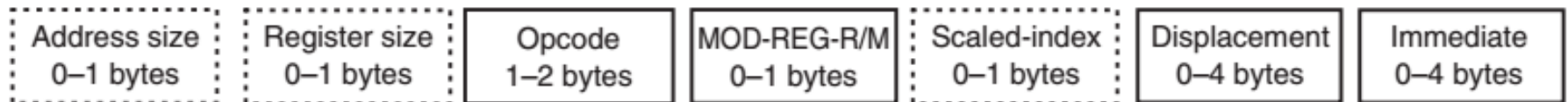
32-bit instruction mode allows the combination of any two 32-bit registers except ESP, e.g., **MOV DL,[EAX+EBX]**



# 32-Bit Addressing Modes

- When R/M=100, an additional byte called a **scaled-index byte** appears in the instruction.
- A scaled-index byte indicates additional forms of scaled-index addressing.

<i>R/M Code</i>	<i>Function</i>
000	DS:[EAX]
001	DS:[ECX]
010	DS:[EDX]
011	DS:[EBX]
100	Uses scaled-index byte
101	SS:[EBP]*
110	DS:[ESI]
111	DS:[EDI]



- Over 32,000 variations of the MOV instruction alone in the 80386 - Core2 microprocessors.
- Figure 4–8 shows the format of the scaled-index byte as selected by a value of 100 in the R/M field of an instruction when the 80386 and above use a 32-bit address.
- The leftmost 2 bits select a scaling factor (multiplier) of 1x, 2x, 4x, 8x.
- Scaled-index addressing can also use a single register multiplied by a scaling factor.

**Figure 4–8** The scaled-index byte.



SS

00 =  $\times 1$

01 =  $\times 2$

10 =  $\times 4$

11 =  $\times 8$

— the index and base fields both contain register numbers

- If the microprocessor operates in the 32-bit instruction mode, The instruction

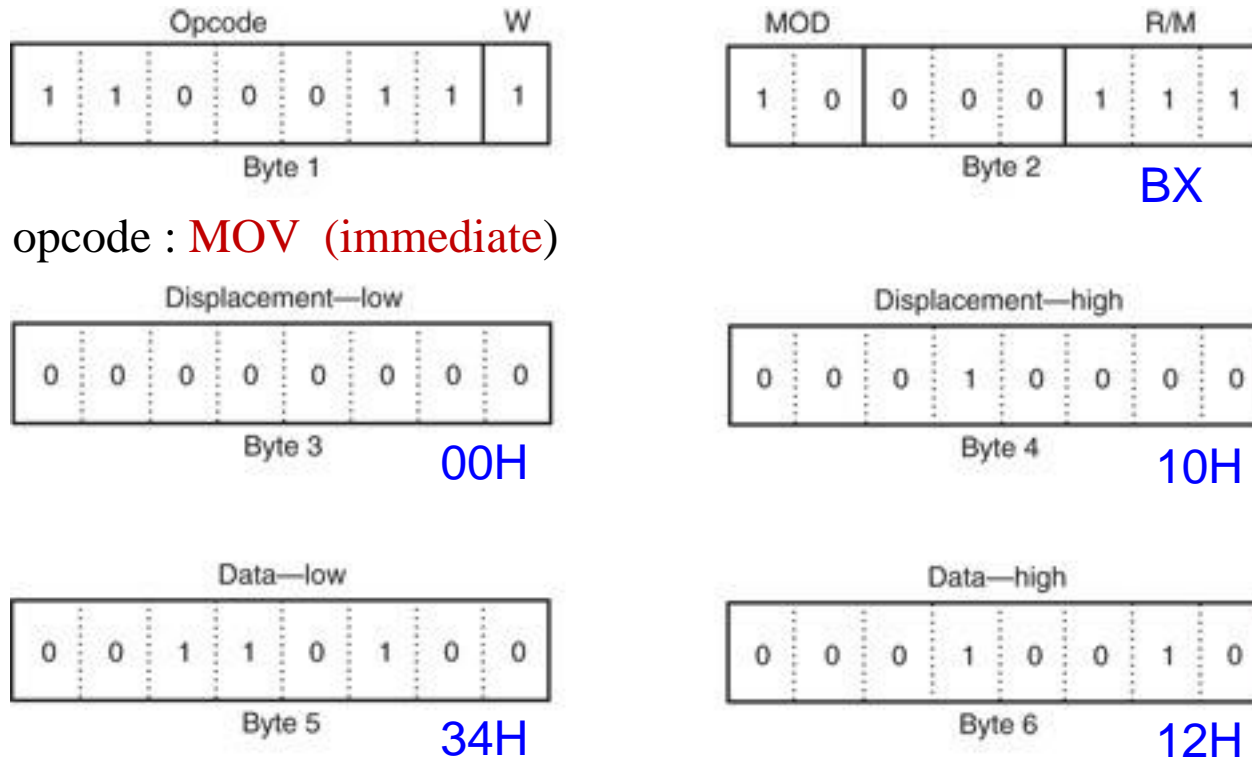
**MOV EAX,[EBX+4\*ECX]** is encoded as **8B048BH**.

- First byte (8BH): **100010 11** → **MOV**
- Second byte (04H): **00 000 100** → **EAX, scaled-index**
- Third byte (8BH): **10 001 011** →  **$\times 4$ , ECX, EBX**

# ***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  $DS \times 10H$
- 6-byte instruction
  - 2 bytes for the opcode, W, MOD, and R/M fields; 2 bytes are the displacement of 1000H; 2 bytes are the data of 1234H
- 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.



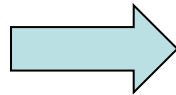
Opcode = MOV (immediate)  
 W = Word  
 MOD = 16-bit displacement  
 REG = 000 (not used in immediate addressing)  
 R/M = DS:[BX]  
 Displacement = 1000H  
 Data = 1234H

- 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.

segment register  
selection



<i>Code</i>	<i>Segment Register</i>
000	ES
001	CS*
010	SS
011	DS
100	FS
101	GS

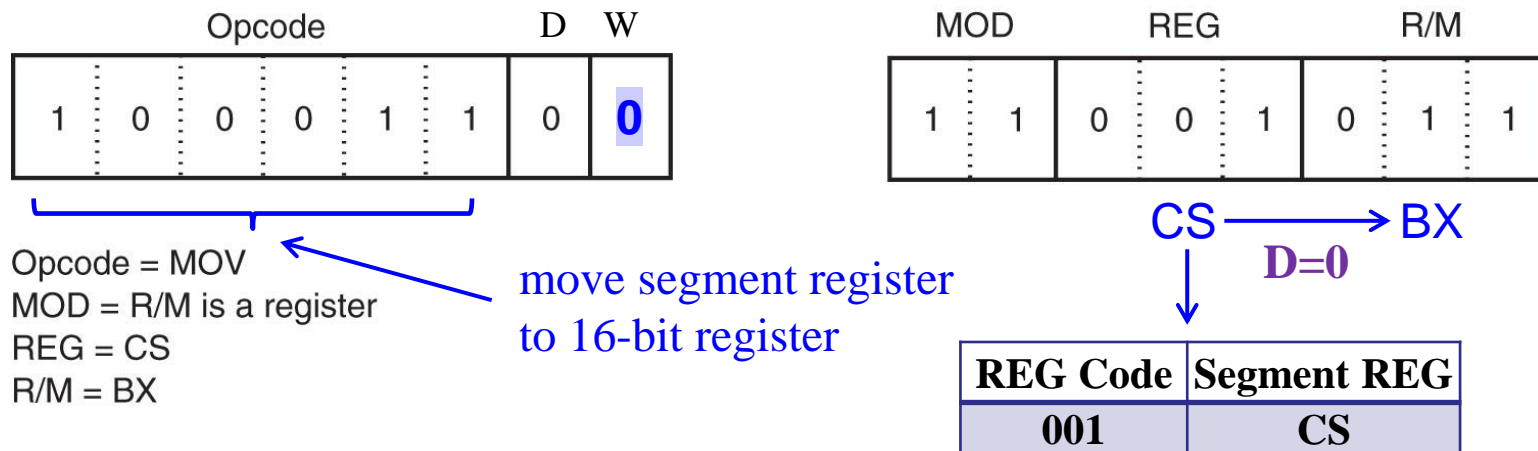
- The opcode for this type of MOV instruction is different for the prior MOV instructions.

- It is important to understand the **limitations** of the segment register MOV instruction:
  - **Immediate data** cannot be moved into a segment register.
  - **CS** cannot successfully be loaded with a segment register MOV.
  - **MOV CS, R/M** and **POP CS** are not allowed.



- 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.



- 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.

## 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.

- **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.
  - CS may be pushed, but data from the stack may never be popped into CS
- The **flags** may be pushed or popped from that stack.
  - contents of all registers may be pushed or popped

# Initializing the Stack

- Assembly language stack segment setup, e.g.,:

```
STACK_SEG SEGMENT STACK
```

```
    DW  100H DUP(?)
```

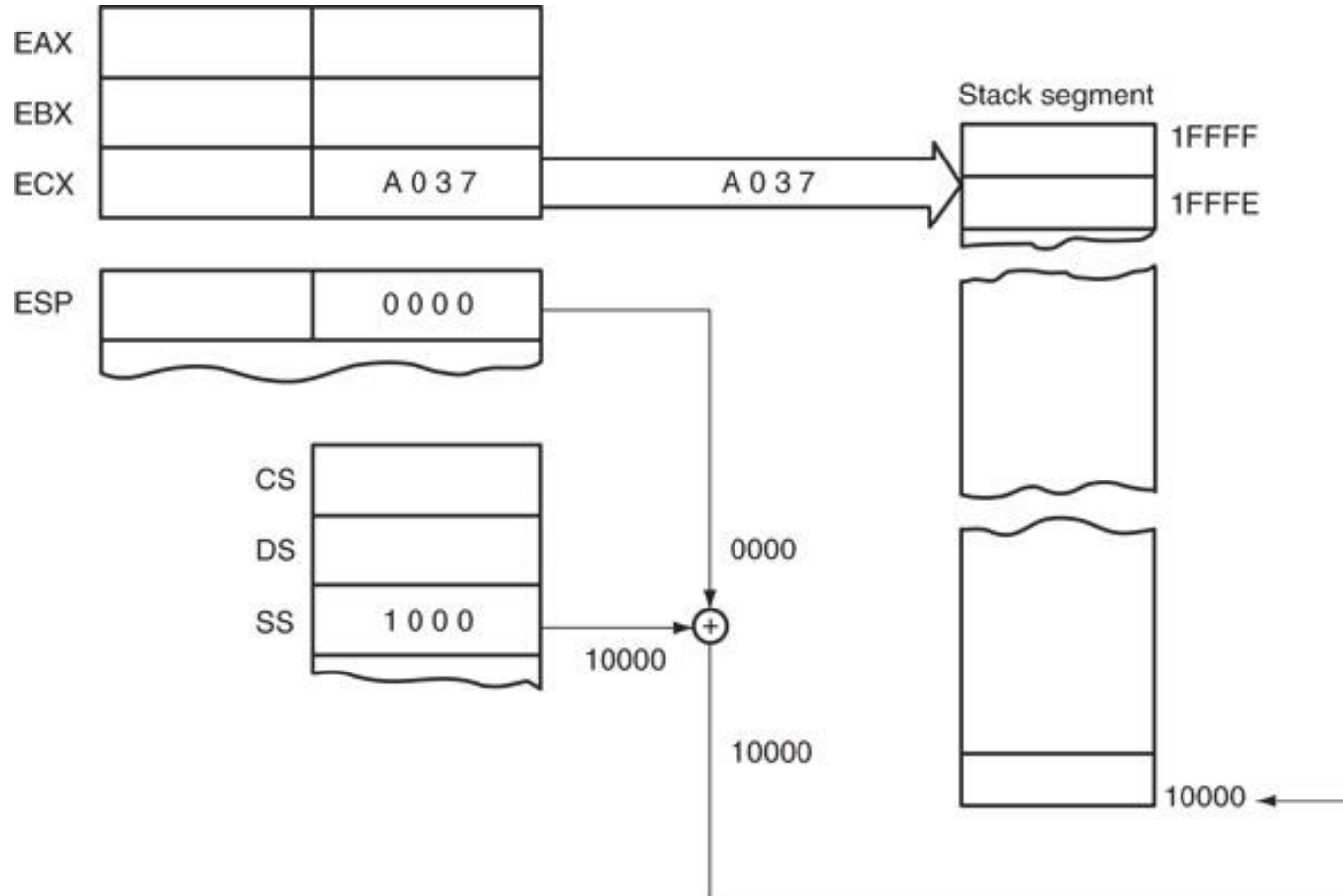
```
STACK_SEG ENDS
```

- 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.

- 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.

- 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.



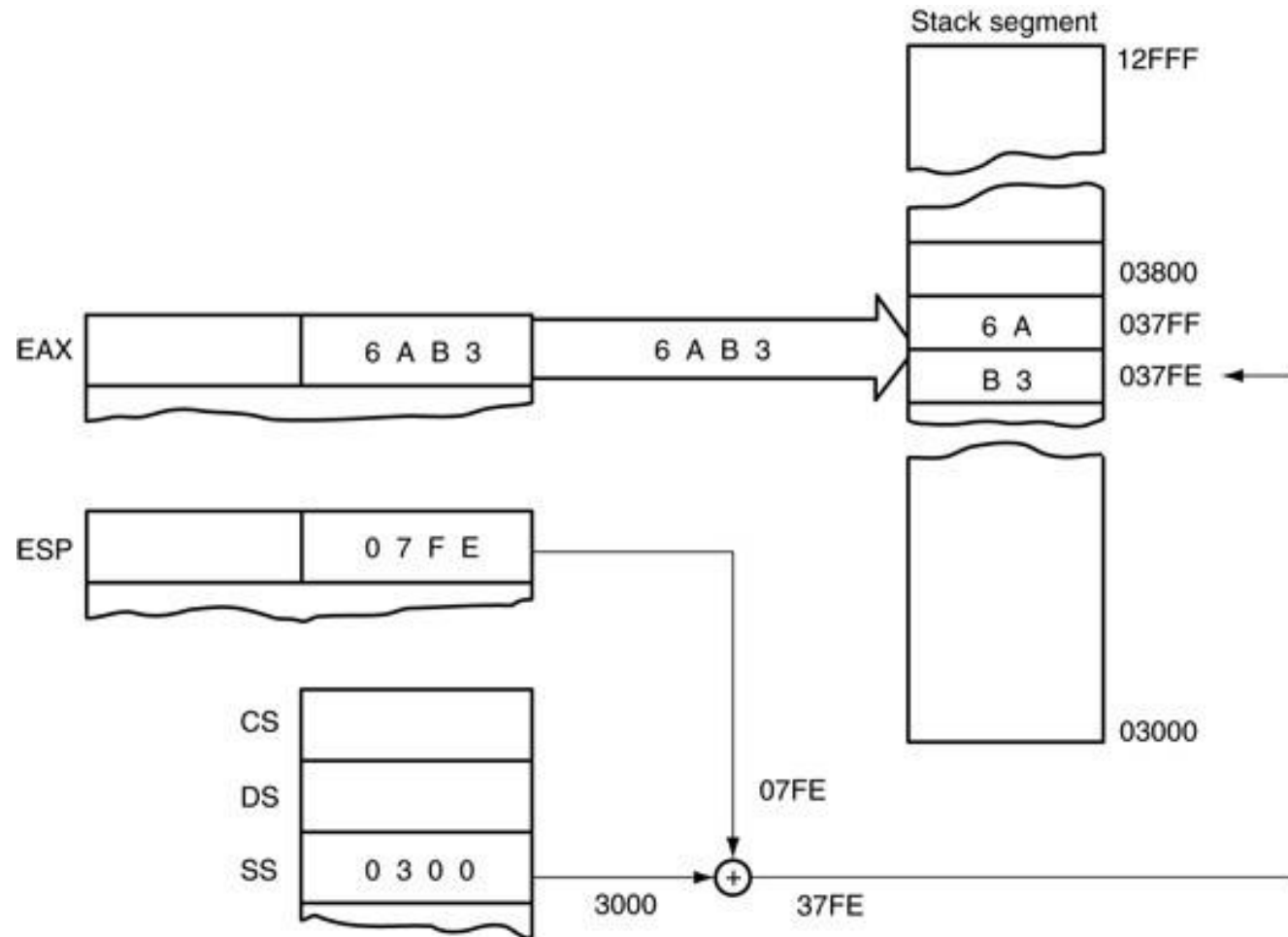


# PUSH

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

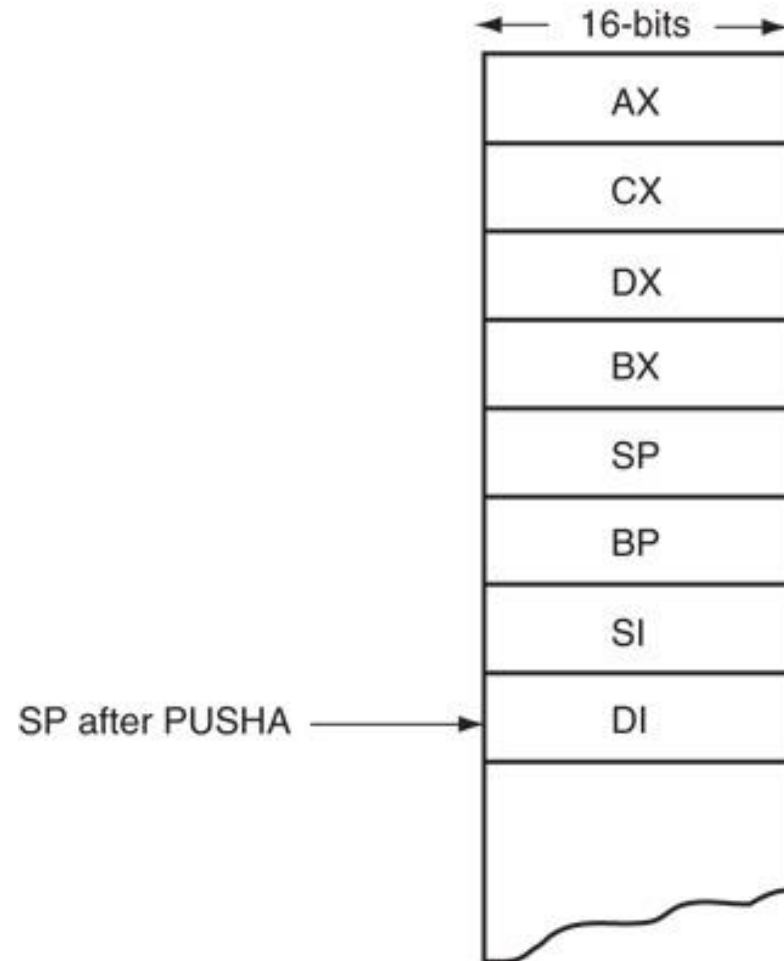
- PUSHF (**push flags**) instruction copies the contents of the flag register to the stack.
- PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set in 80386 - Pentium 4.
  - PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4

**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 PUSHAD instruction, showing the location and order of stack data.

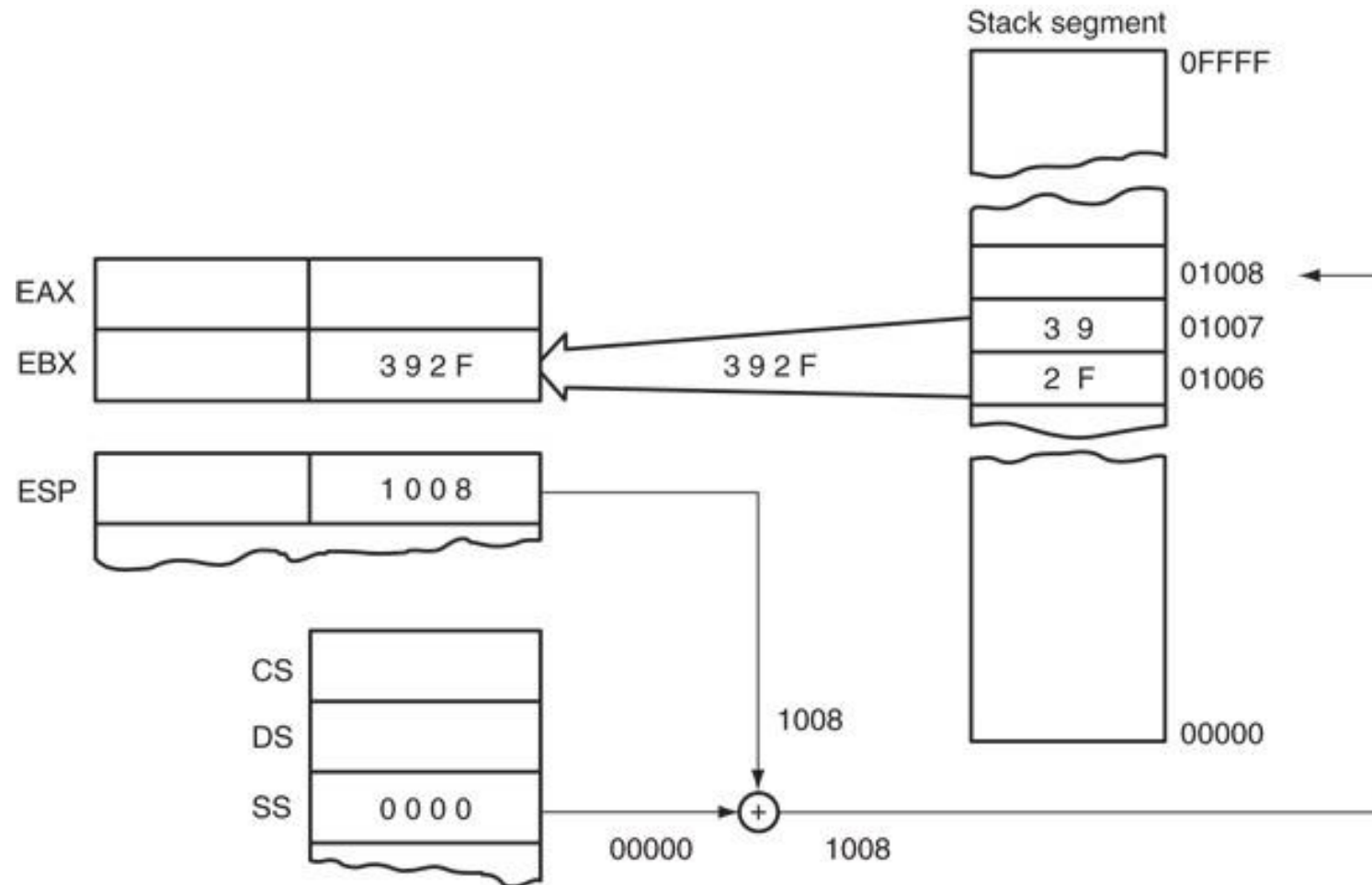


# POP

- 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 PUSHAD 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.





# ***x86-64 Code Models***

- Various programs differ in addressing (absolute versus position independent), code size, data size and address range.
- Code models define constraints for compiler to generate better code.
- The AMD provides seven different code models for programs running in 64-bit mode.
  - Small code model, Kernel code model
  - Medium code model, Large code model
  - Small/Medium/Large position independent code model (PIC)

# ***x86-64 Code Models***

- The small code model is the default code model, which uses the fast RIP-relative addressing modes.
- And the program and its statically defined symbols must be within 4GB of each other.

Understanding the x64 code models:

<https://eli.thegreenplace.net/2012/01/03/understanding-the-x64-code-models>