SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UNIT–II
8086 Family Assembly Language Programming

UNIT II

- Machine language instruction format-Addressing modes-Data addressing-Program memory and stack addressing modes-Instruction Set: Data Movement Instructions-Arithmetic and Logic Instructions-Program control Instructions-Assembler Directives of 8086

Introduction

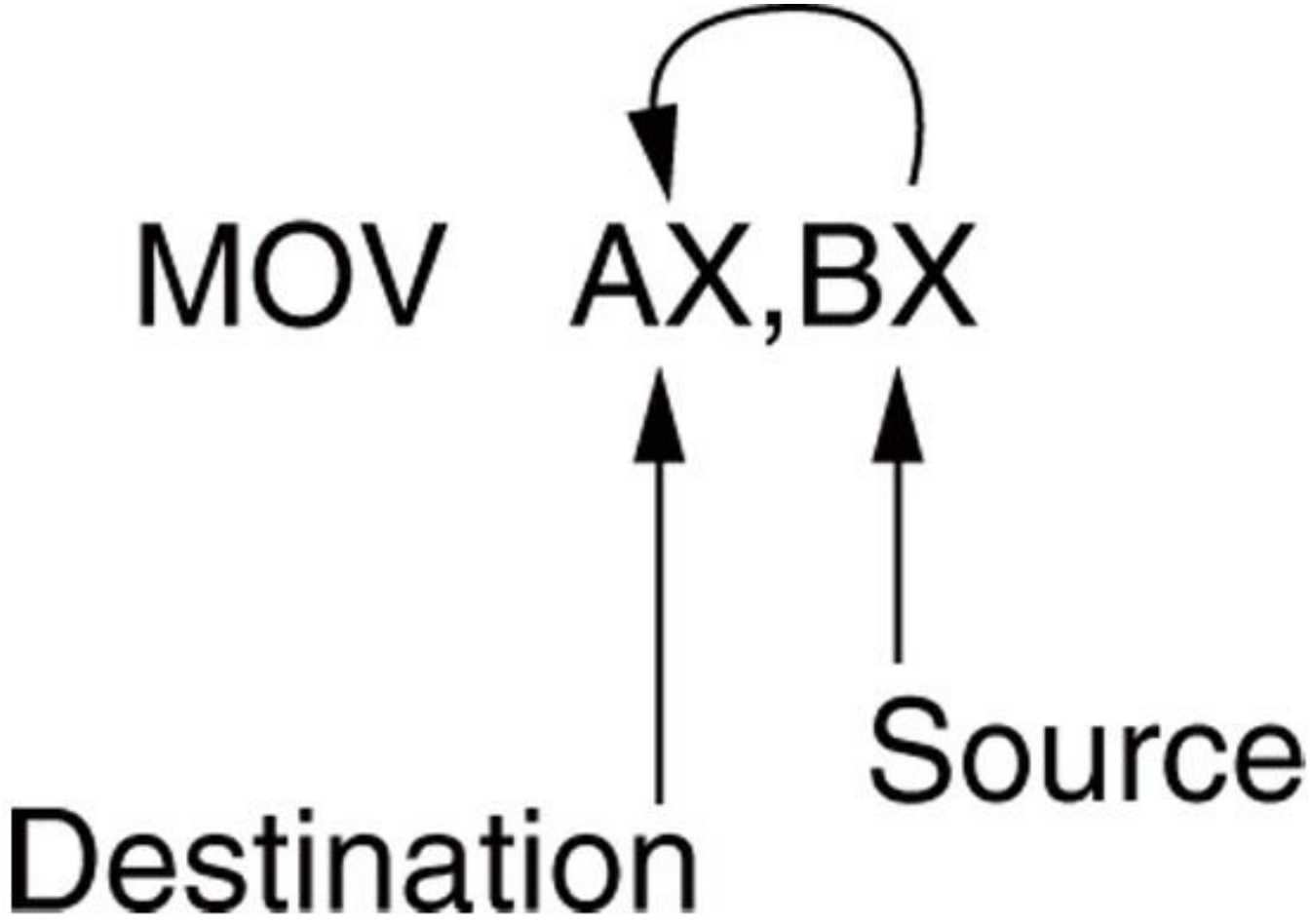
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Efficient software development for the microprocessor requires a complete familiarity with the addressing modes employed by each instruction.
- This chapter explains the operation of the stack memory so that the PUSH and POP instructions and other stack operations will be understood.

3–1 DATA ADDRESSING MODES

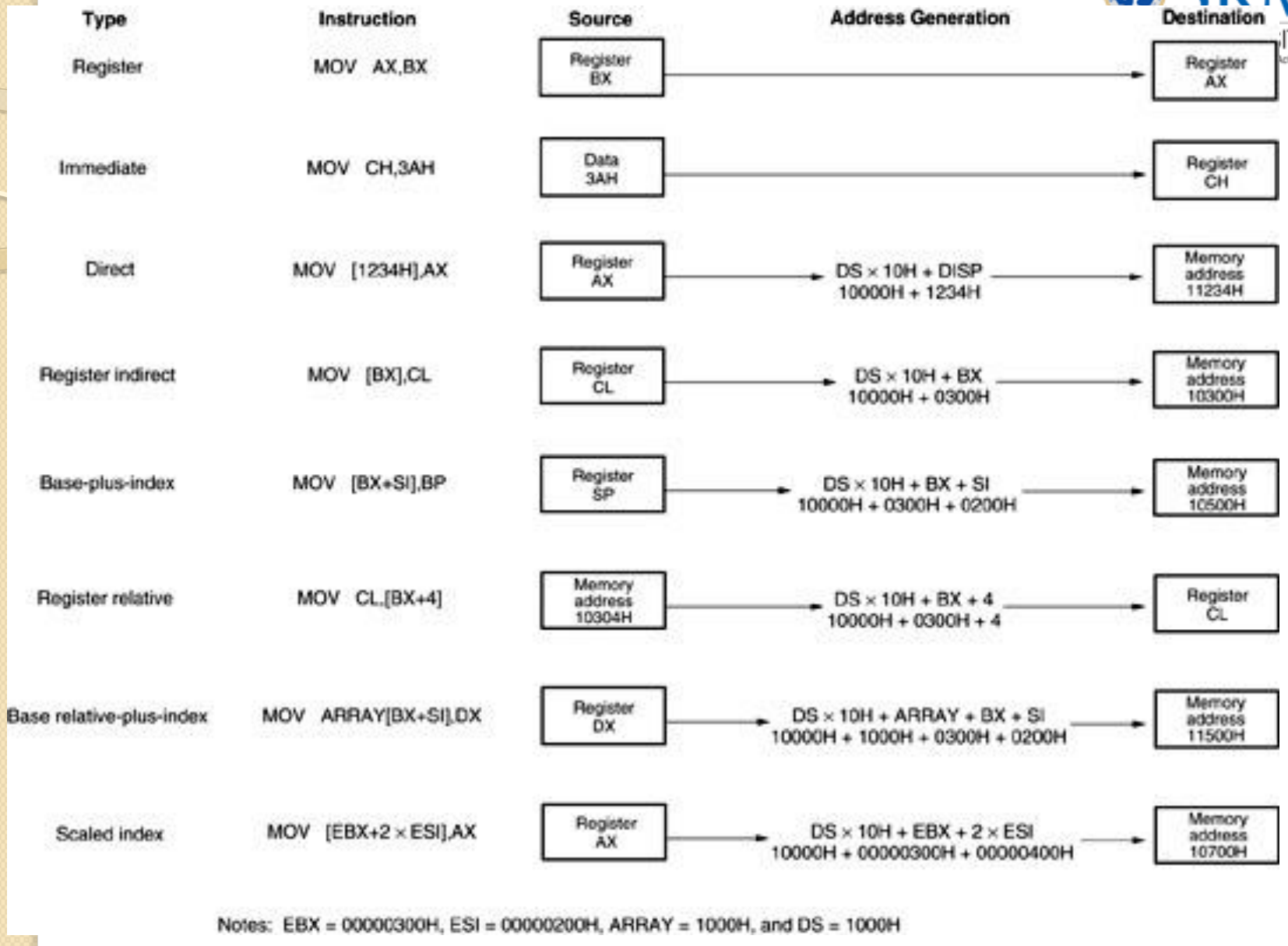
- MOV instruction is a common and flexible instruction.
  - provides a basis for explanation of data-addressing modes
- Figure 3–1 illustrates the MOV instruction and defines the direction of data flow.
- **Source** is to the right and **destination** the left, next to the opcode MOV.
  - an **opcode**, or operation code, tells the microprocessor which operation to perform

Figure 3–1 The MOV instruction showing the source, destination, and direction of data flow.



- Figure 3–2 shows all possible variations of the data-addressing modes using MOV.
- These data-addressing modes are found with all versions of the Intel microprocessor.
  - except for the scaled-index-addressing mode, found only in 80386 through Core2
- RIP relative addressing mode is not illustrated.
  - only available on the Pentium 4 and Core2 in the 64-bit mode

Figure 3–2 8086–Core2 data-addressing modes.



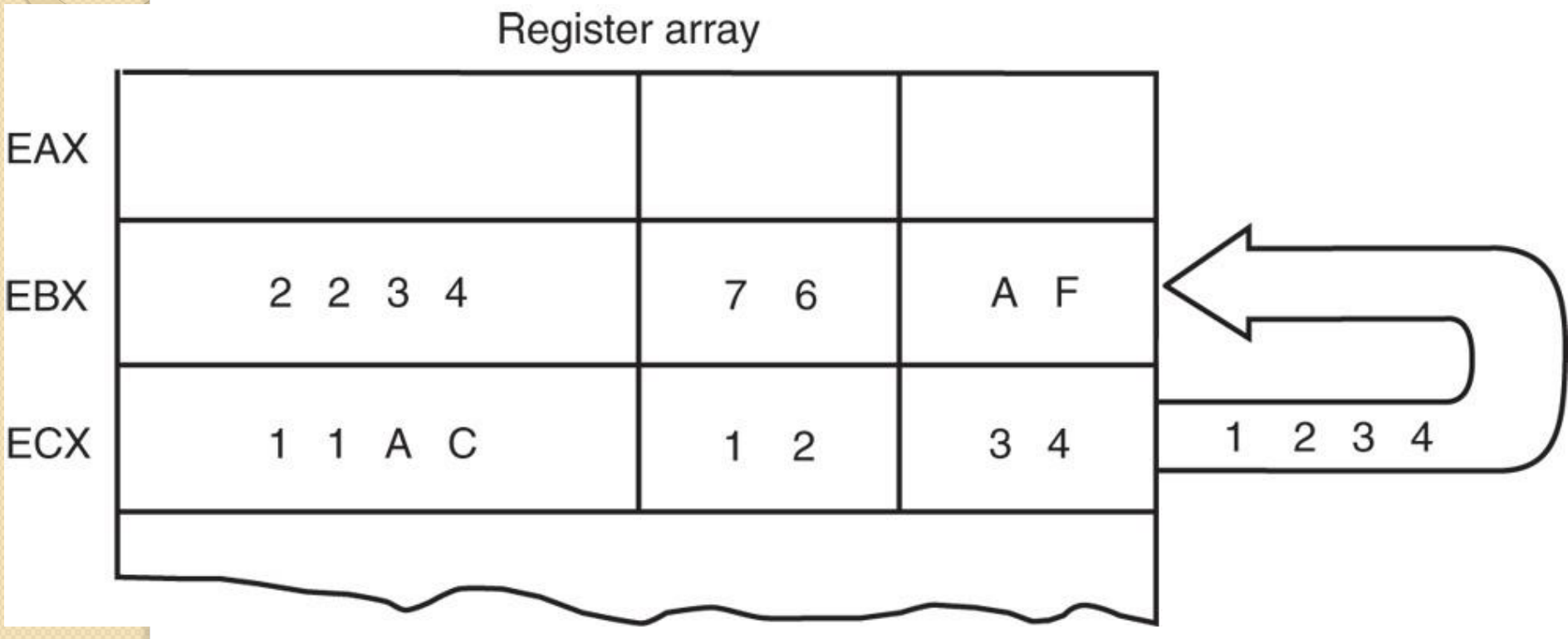
Register Addressing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The most common form of data addressing.
  - once register names learned, easiest to apply.
- The microprocessor contains these 8-bit register names used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL.
- 16-bit register names: AX, BX, CX, DX, SP, BP, SI, and DI.

- In 80386 & above, extended 32-bit register names are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
- 64-bit mode register names are: RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15.
- Important for instructions to use registers that are the same size.
  - *never* mix an 8-bit \with a 16-bit register, an 8- or a 16-bit register with a 32-bit register
  - this is not allowed by the microprocessor and results in an error when assembled

Figure 3–3 The effect of executing the MOV BX, CX instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.

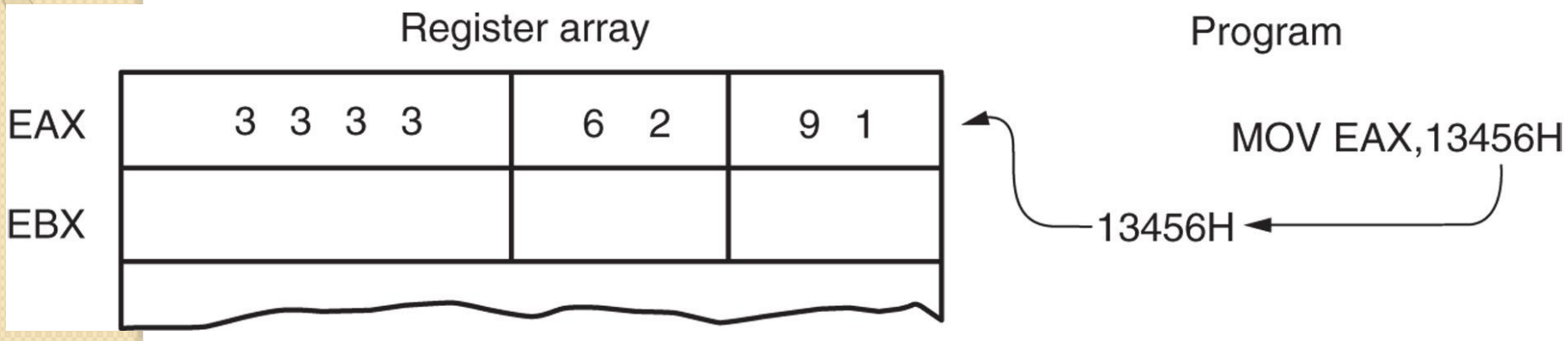


- Figure 3–3 shows the operation of the MOV BX, CX instruction.
- The source register's contents do not change.
  - the destination register's contents do change
- The contents of the destination register or destination memory location change for all instructions except the CMP and TEST instructions.
- The MOV BX, CX instruction does not affect the leftmost 16 bits of register EBX.

Immediate Addressing

- Term *immediate* implies that data immediately follow the hexadecimal opcode in the memory.
  - immediate data are constant data
  - data transferred from a register or memory location are variable data
- Immediate addressing operates upon a byte or word of data.
- Figure 3–4 shows the operation of a MOV EAX,13456H instruction.

Figure 3–4 The operation of the MOV EAX,3456H instruction. This instruction copies the immediate data (13456H) into EAX.



- As with the MOV instruction illustrated in Figure 3–3, the source data overwrites the destination data.

- In symbolic assembly language, the symbol # precedes immediate data in some assemblers.
  - MOV AX,#3456H instruction is an example
- Most assemblers do not use the # symbol, but represent immediate data as in the MOV AX,3456H instruction.
  - an older assembler used with some Hewlett-Packard logic development does, as may others
  - in this text, the # is not used for immediate data

- The symbolic assembler portrays immediate data in many ways.
- The letter **H** appends hexadecimal data.
- If hexadecimal data begin with a letter, the assembler requires the data start with a **0**.
  - to represent a hexadecimal F2, 0F2H is used in assembly language
- Decimal data are represented as is and require no special codes or adjustments.
  - an example is the 100 decimal in the MOV AL,100 instruction

- An ASCII-coded character or characters may be depicted in the immediate form if the ASCII data are enclosed in apostrophes.
  - be careful to use the apostrophe (') for ASCII data and not the single quotation mark (')
- Binary data are represented if the binary number is followed by the letter B.
  - in some assemblers, the letter Y

- Each statement in an assembly language program consists of four parts or fields.
- The leftmost field is called the *label*.
  - used to store a symbolic name for the memory location it represents
- All labels must begin with a letter or one of the following special characters: @, $, -, or ?.
  - a label may any length from 1 to 35 characters
- The label appears in a program to identify the name of a memory location for storing data and for other purposes.

- The next field to the right is the *opcode field*.
  - designed to hold the instruction, or opcode
  - the MOV part of the move data instruction is an example of an opcode
- Right of the opcode field is the *operand field*.
  - contains information used by the opcode
  - the MOV AL,BL instruction has the opcode MOV and operands AL and BL
- The *comment field*, the final field, contains a comment about the instruction(s).
  - comments always begin with a semicolon (;)

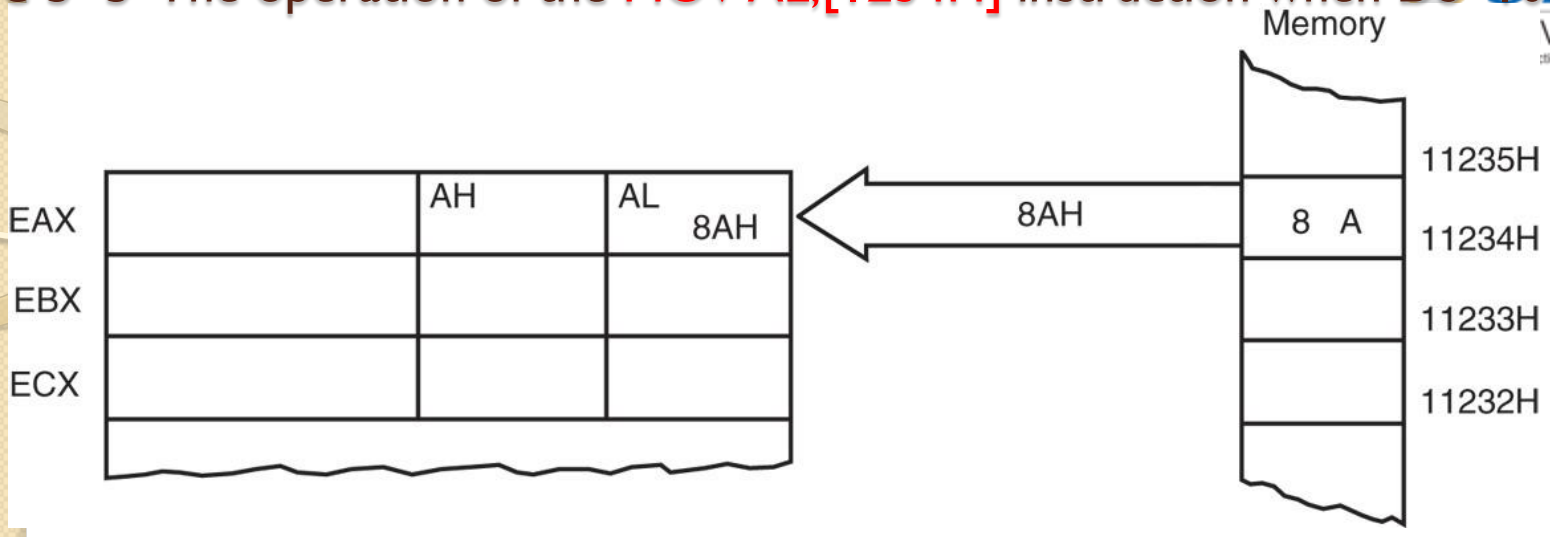
Direct Data Addressing

- Applied to many instructions in a typical program.
- Two basic forms of direct data addressing:
  - direct addressing, which applies to a MOV between a memory location and AL, AX, or EAX
  - displacement addressing, which applies to almost any instruction in the instruction set
- Address is formed by adding the displacement to the default data segment address or an alternate segment address.

Direct Addressing

- Direct addressing with a MOV instruction transfers data between a memory location, located within the data segment, and the AL (8-bit), AX (16-bit), or EAX (32-bit) register.
  - usually a 3-byte long instruction
- MOV AL,DATA loads AL from the data segment memory location DATA (1234H).
  - DATA is a symbolic memory location, while 1234H is the actual hexadecimal location

Figure 3–5 The operation of the MOV AL,[1234H] instruction when DS=1000H.



- This instruction transfers a copy contents of memory location 11234H into AL.
  - the effective address is formed by adding 1234H (the offset address) and 10000H (the data segment address of 1000H times 10H) in a system operating in the real mode

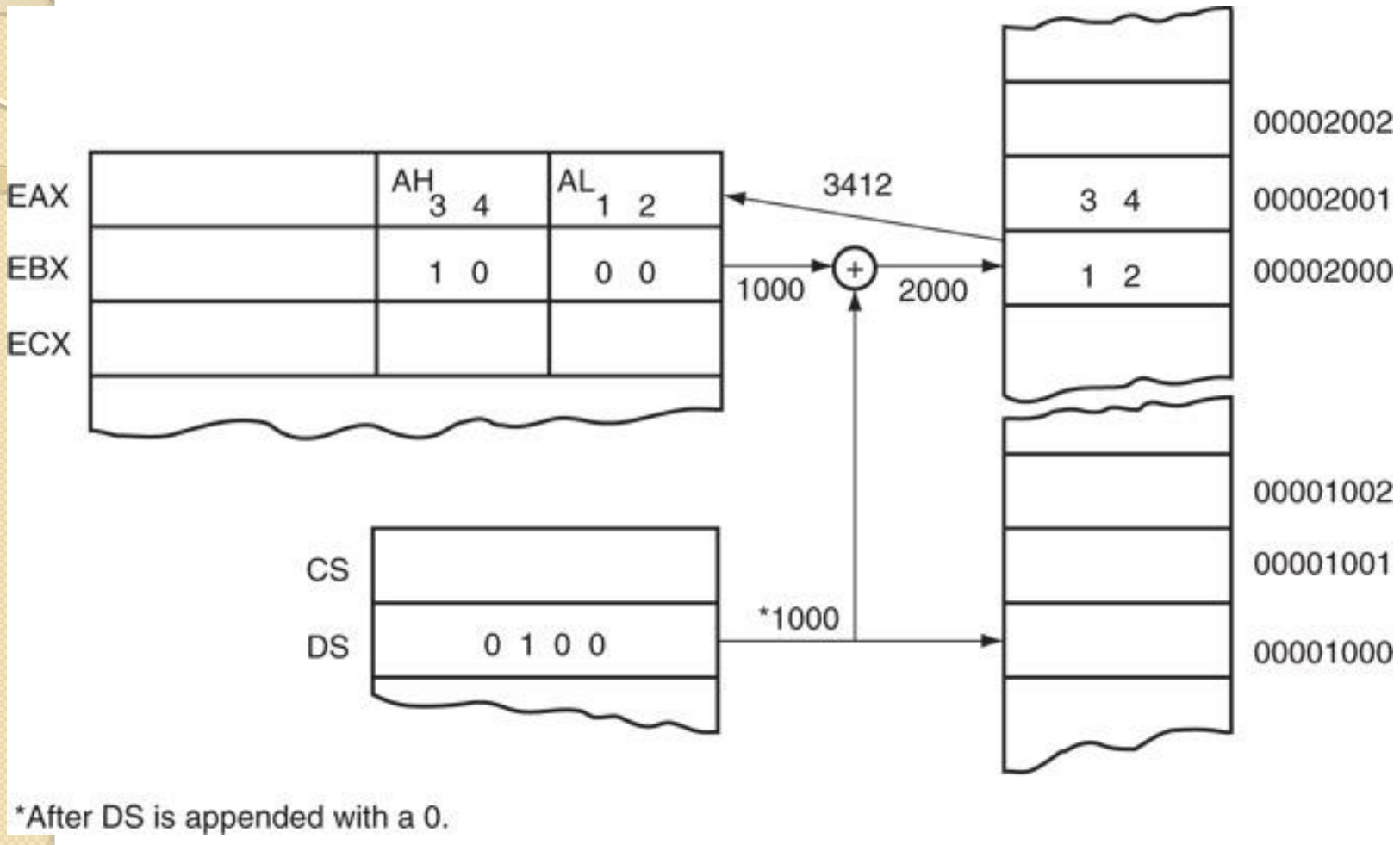
Displacement Addressing

- Almost identical to direct addressing, except the instruction is 4 bytes wide instead of 3.
- In 80386 through Pentium 4, this instruction can be up to 7 bytes wide if a 32-bit register and a 32-bit displacement are specified.
- This type of direct data addressing is much more flexible because most instructions use it.

Register Indirect Addressing

- Allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI, and SI.
- In addition, 80386 and above allow register indirect addressing with any extended register except ESP.
- In the 64-bit mode, the segment registers serve no purpose in addressing a location in the flat model.

Figure 3–6 The operation of the MOV AX, [BX] instruction when BX = 1000H and DS = 0100H. Note that this instruction is shown after the contents of memory are transferred to AX.



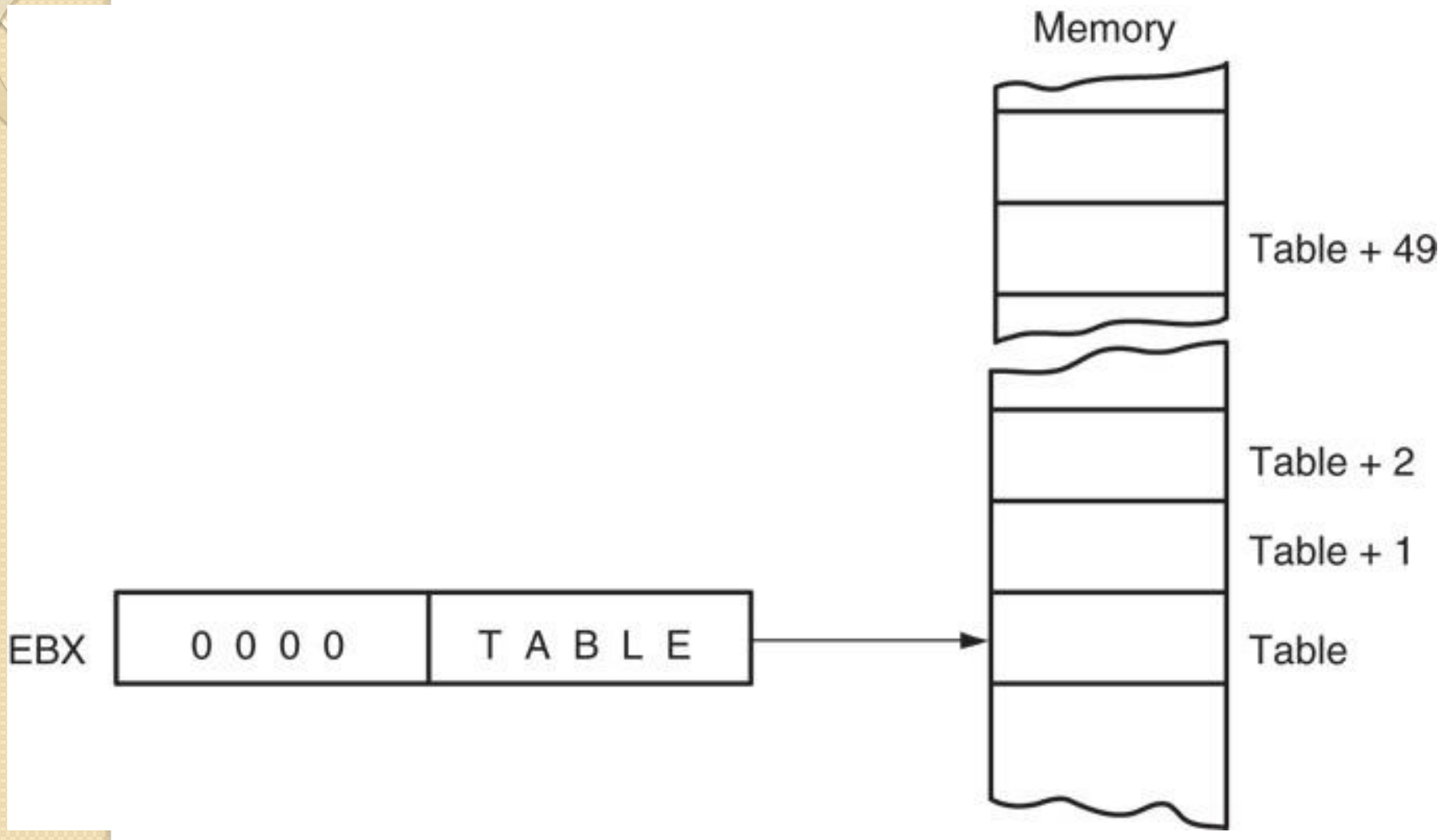
- The **data segment** is used by default with register indirect addressing or any other mode that uses BX, DI, or SI to address memory.
- If the BP register addresses memory, the **stack segment** is used by default.
  - these settings are considered the default for these four index and base registers
- For the 80386 and above, EBP addresses memory in the stack segment by default.
- EAX, EBX, ECX, EDX, EDI, and ESI address memory in the data segment by default.

- When using a 32-bit register to address memory in the real mode, contents of the register must never exceed 0000FFFFH.
- In the protected mode, any value can be used in a 32-bit register that is used to indirectly address memory.
  - as long as it does not access a location outside the segment, dictated by the access rights byte
- In the 64-bit mode, segment registers are not used in address calculation; the register contains the actual linear memory address.

- In some cases, indirect addressing requires specifying the size of the data by the **special assembler directive** BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR.
  - these directives indicate the size of the memory data addressed by the memory **pointer** (PTR)
- The directives are with instructions that address a memory location through a pointer or index register with immediate data.
- With SIMD instructions, the octal OWORD PTR, represents a 128-bit-wide number.

- Indirect addressing often allows a program to refer to tabular data located in memory.
- Figure 3–7 shows the table and the BX register used to sequentially address each location in the table.
- To accomplish this task, load the starting location of the table into the BX register with a MOV immediate instruction.
- After initializing the starting address of the table, use register indirect addressing to store the 50 samples sequentially.

Figure 3–7 An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.



Base-Plus-Index (索引、註標) Addressing

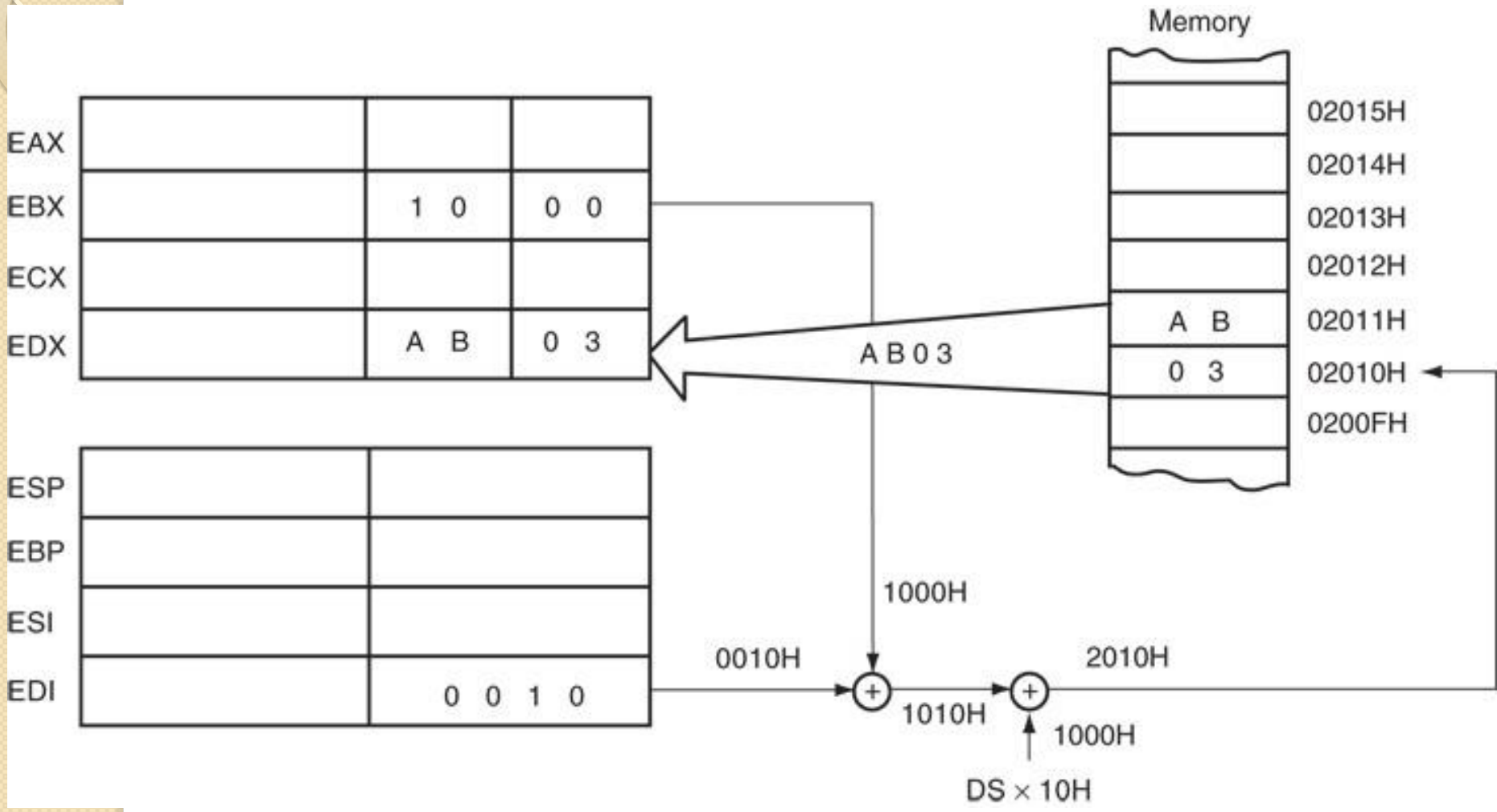
- Similar to indirect addressing because it indirectly addresses memory data.
- The base register often holds the beginning location of a memory array.
  - the index register holds the relative position of an element in the array
  - whenever BP addresses memory data, both the stack segment register and BP generate the effective address

Locating Data with Base-Plus-Index Addressing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Figure 3–8 shows how data are addressed by the MOV DX, [BX + DI] instruction when the microprocessor operates in the real mode.
- The Intel assembler requires this addressing mode appear as [BX][DI] instead of [BX + DI].
- The MOV DX, [BX + DI] instruction is MOV DX,[BX][DI] for a program written for the Intel ASM assembler.

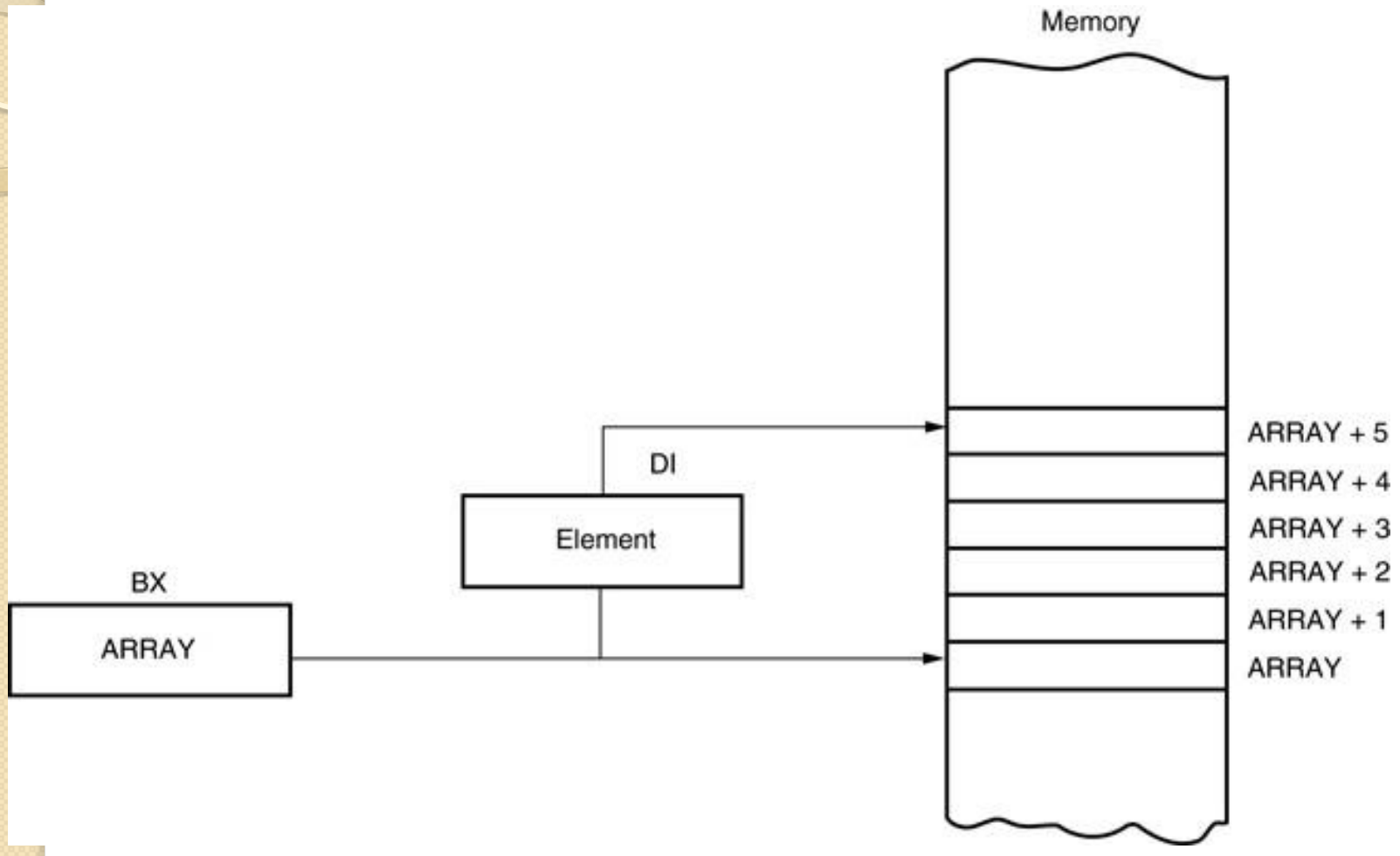
Figure 3–8 An example showing how the base-plus-index addressing mode functions for the MOV DX, [BX + DI] instruction. Notice that memory address 02010H is accessed because DS=0100H, BX=100H and DI=0010H.



Locating Array Data Using Base-Plus-Index Addressing

- A major use is to address elements in a memory array.
- To accomplish this, load the BX register (base) with the beginning address of the array and the DI register (index) with the element number to be accessed.
- Figure 3–9 shows the use of BX and DI to access an element in an array of data.

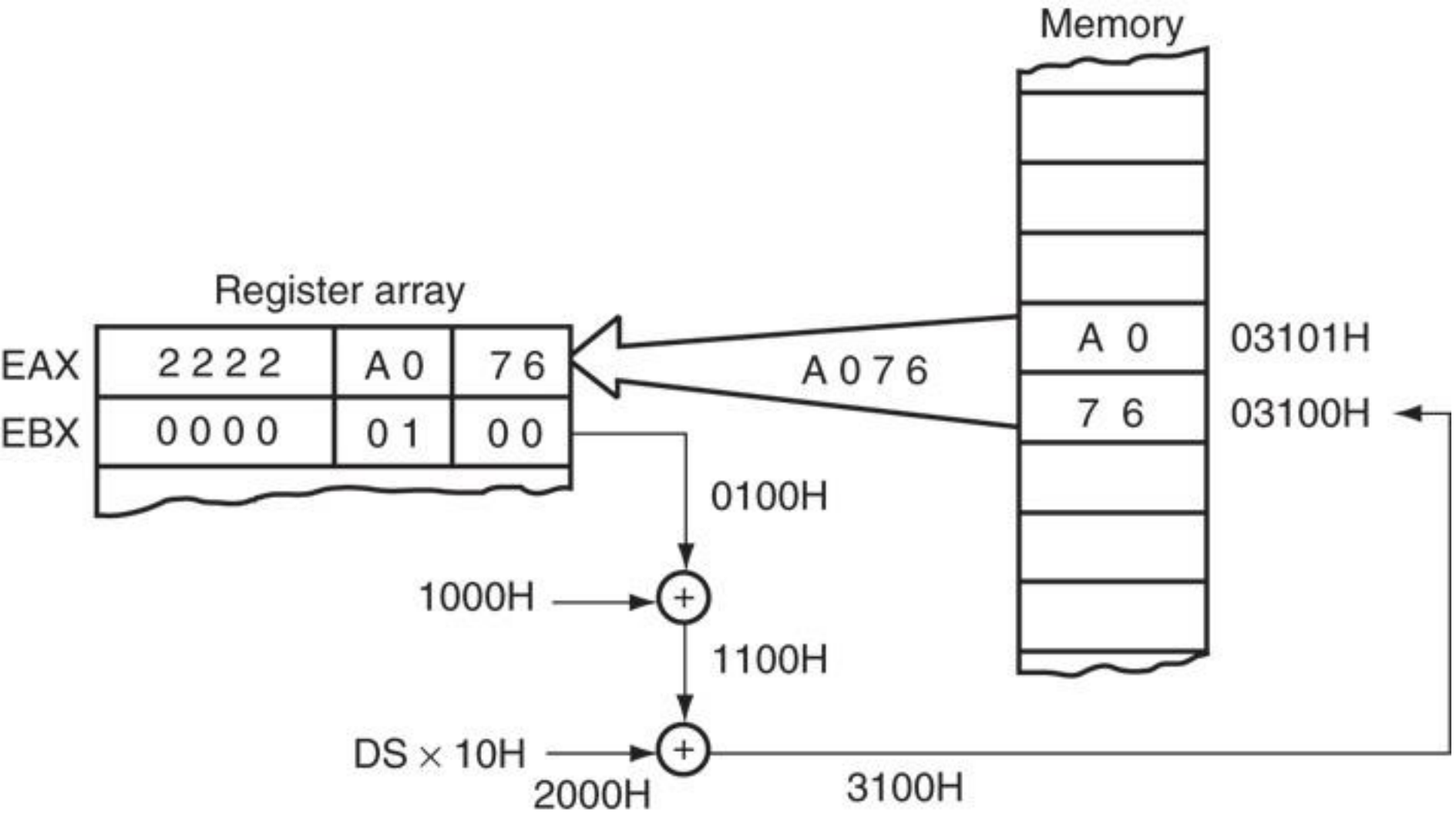
Figure 3–9 An example of the base-plus-index addressing mode. Here an element (DI) of an ARRAY (BX) is addressed.



Register Relative Addressing

- Similar to base-plus-index addressing and displacement addressing.
  - data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register (BP, BX, DI, or SI)
- Figure 3–10 shows the operation of the MOV AX,[BX+1000H] instruction.
- A real mode segment is 64K bytes long.

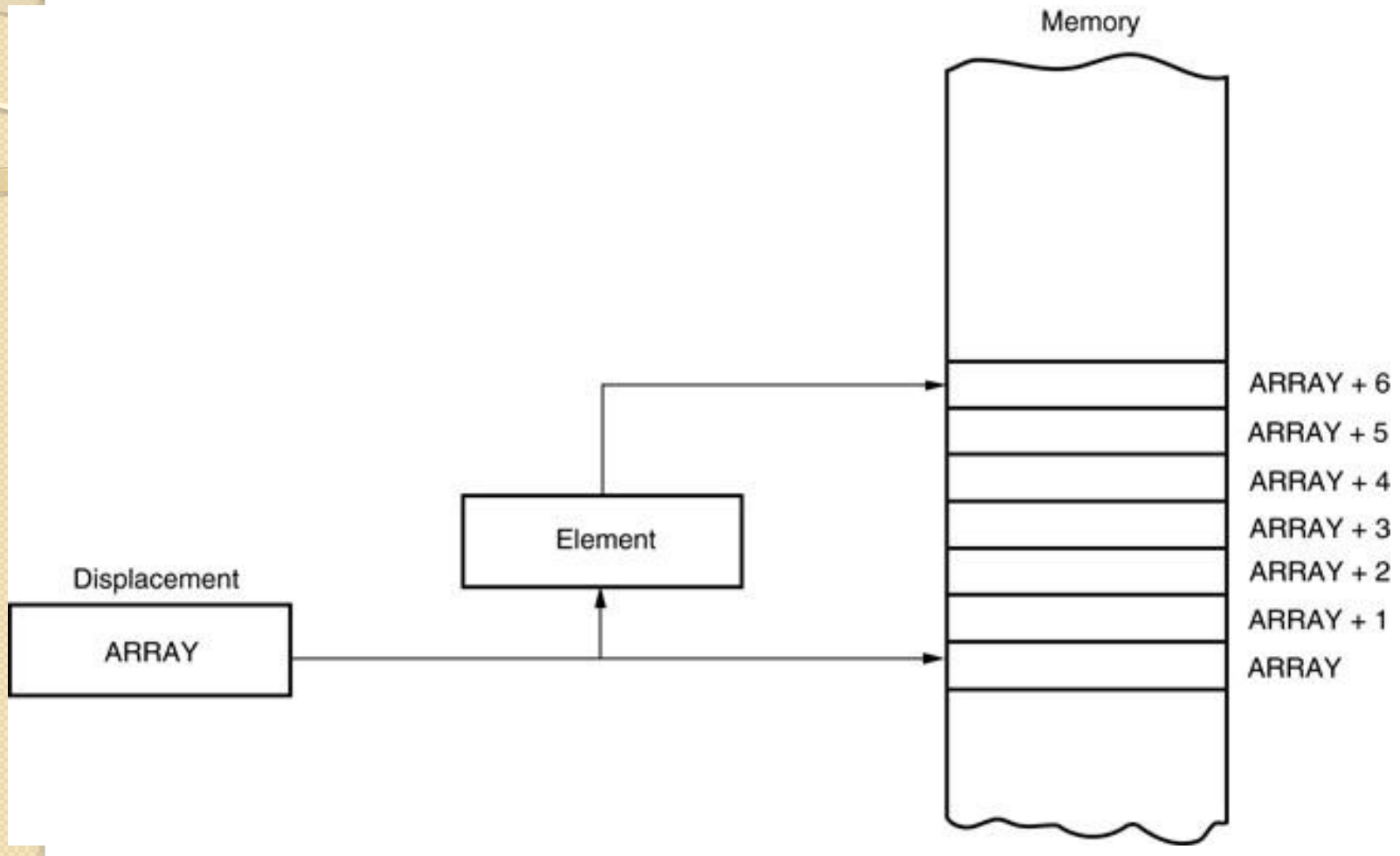
Figure 3–10 The operation of the MOV AX, [BX+1000H] instruction, when BX=1000H and DS=0200H .



Addressing Array Data with Register Relative

- It is possible to address array data with register relative addressing.
  - such as with base-plus-index addressing
- In Figure 3–11, register relative addressing is illustrated with the same example as for base-plus-index addressing.
  - this shows how the displacement ARRAY adds to index register DI to generate a reference to an array element

Figure 3–11 Register relative addressing used to address an element of ARRAY. The displacement addresses the start of ARRAY, and DI accesses an element.



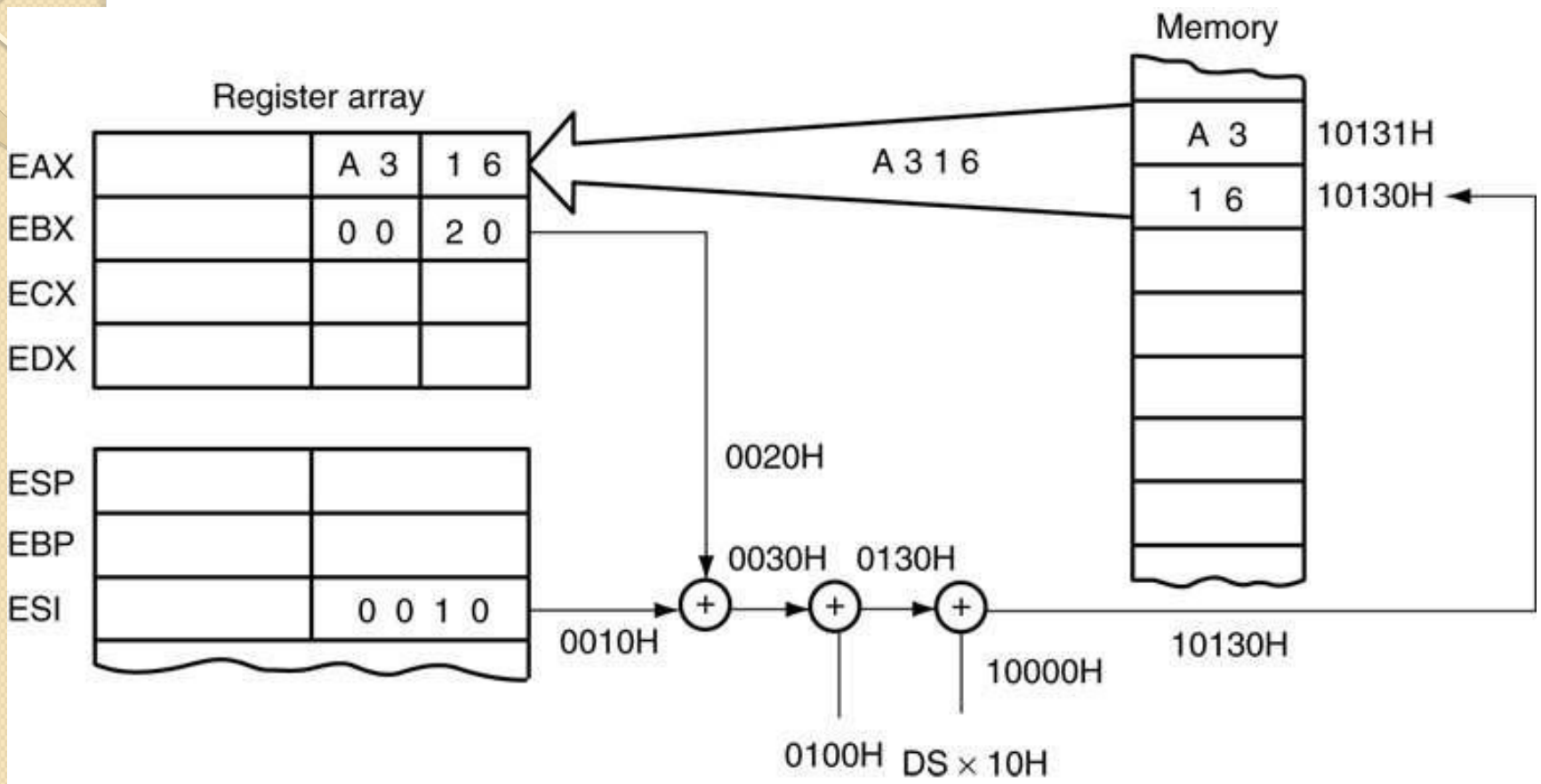
Base Relative-Plus-Index Addressing

- Similar to base-plus-index addressing.
  - adds a displacement
  - uses a base register and an index register to form the memory address
- This type of addressing mode often addresses a two-dimensional array of memory data.

Addressing Data with Base Relative-Plus-Index

- Least-used addressing mode.
- Figure 3–12 shows how data are referenced if the instruction executed by the microprocessor is MOV AX, [BX + SI + 100H].
  - displacement of 100H adds to BX and SI to form the offset address within the data segment
- This addressing mode is too complex for frequent use in programming.

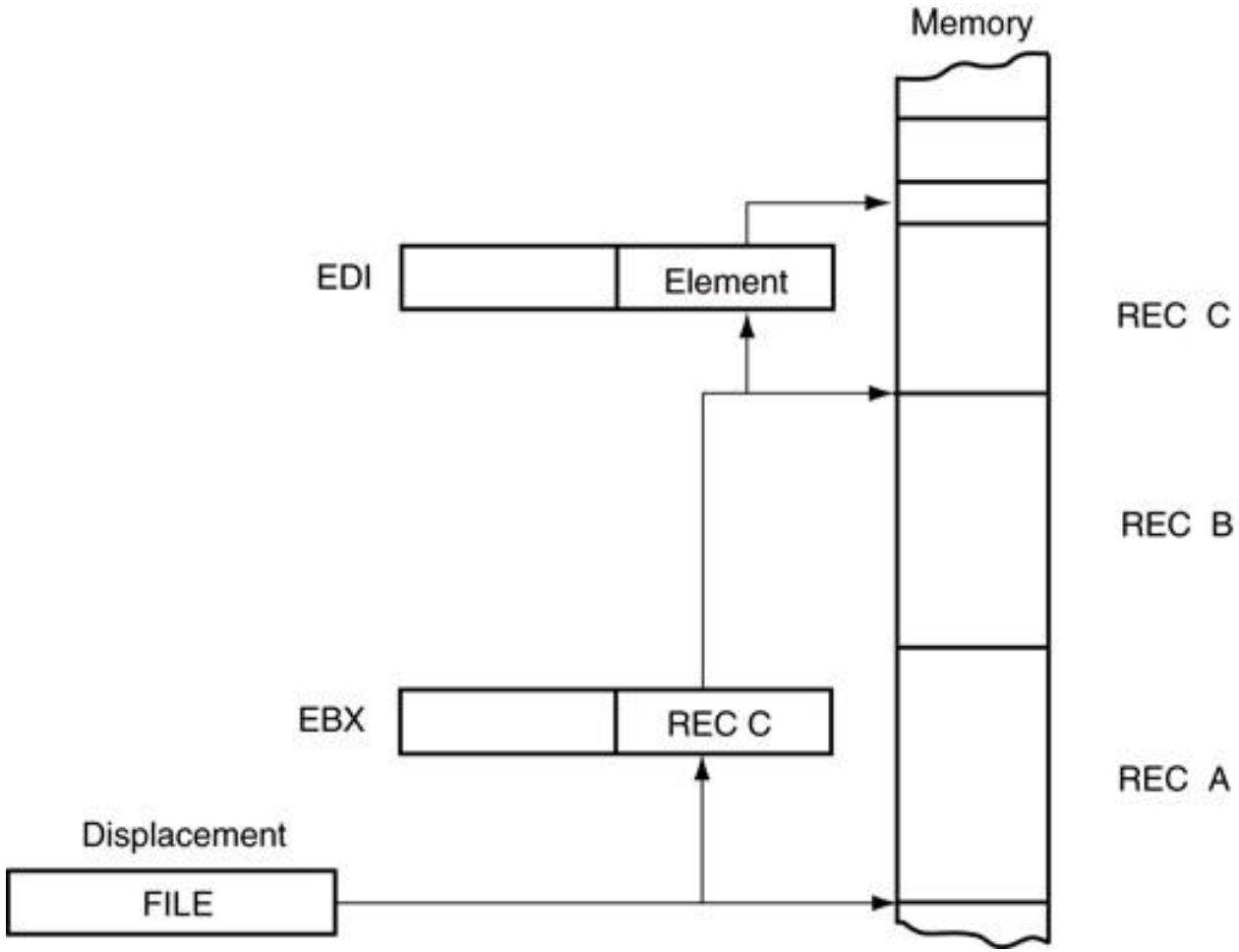
Figure 3–12 An example of base relative-plus-index addressing using a MOV AX,[BX+SI=1000H] instruction. Note: DS=1000H



Addressing Arrays with Base Relative-Plus-Index

- Suppose a file of many records exists in memory, each record with many elements.
  - displacement addresses the file, base register addresses a record, the index register addresses an element of a record
- Figure 3–13 illustrates this very complex form of addressing.

Figure 3–13 Base relative-plus-index addressing used to access a FILE that contains multiple records (REC).



Scaled-Index Addressing

- Unique to 80386 - Core2 microprocessors.
  - uses two 32-bit registers (a base register and an index register) to access the memory
- The second register (index) is multiplied by a scaling factor.
  - the scaling factor can be 1x, 2x, 4x, 8x
- A scaling factor of is implied and need not be included in the assembly language instruction (MOV AL, [EBX + ECX]).

RIP Relative Addressing

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Uses the 64-bit instruction pointer register in the 64-bit mode to address a linear location
  in the flat memory model.
- Inline assembler program available to Visual does not contain any way of using this mode or any other 64-bit addressing mode.
- The Microsoft Visual does not at present support developing 64-bit assembly code.

Data Structures

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Used to specify how information is stored in a memory array.
  - a template for data
- The start of a structure is identified with the STRUC assembly language directive (指導的) and the end with the ENDS statement.

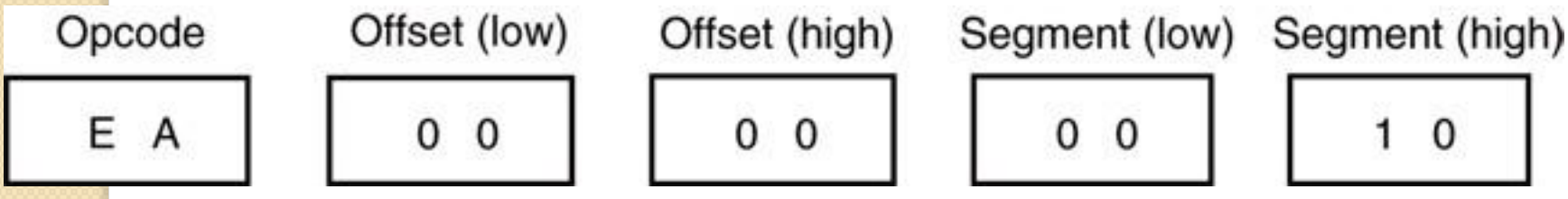
3–2 PROGRAM MEMORY-ADDRESSING MODES

- Used with the JMP (jump) and CALL instructions.
- Consist of three distinct forms:
  - direct, relative, and indirect

Direct Program Memory Addressing

- Used for all jumps and calls by early microprocessor; also used in high-level languages, such as BASIC.
  - GOTO and GOSUB instructions
- The microprocessor uses this form, but not as often as relative and indirect program memory addressing.
- The instructions for direct program memory addressing store the address with the opcode.

Figure 3–14 The 5-byte machine language version of a JMP [10000H] instruction.



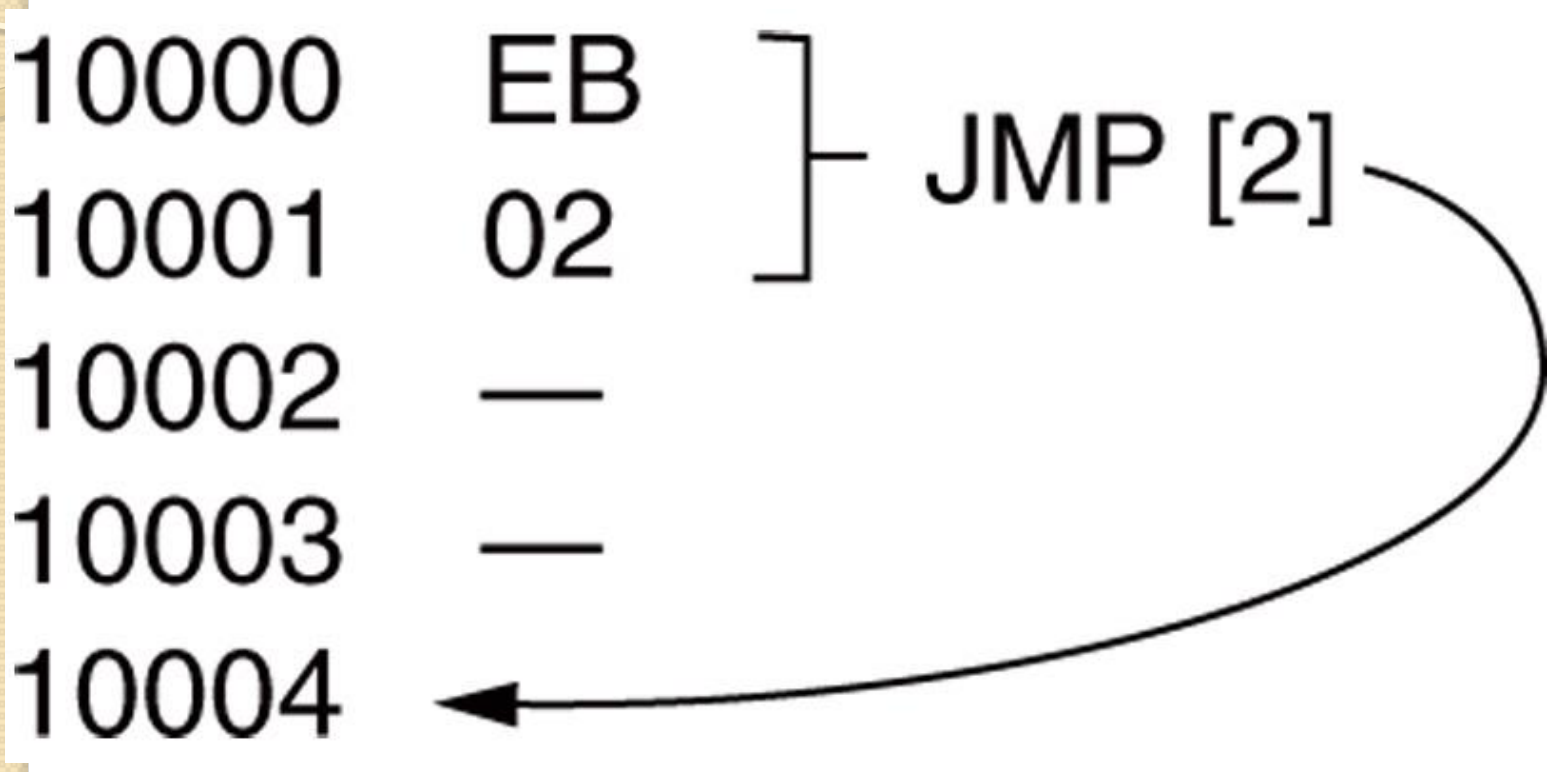
- This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction.
  - an **intersegment jump** is a jump to any memory location within the entire memory system
- Often called a *far jump* because it can jump to any memory location for the next instruction.
  - in real mode, any location within the first 1M byte
  - In protected mode operation, the far jump can jump to any location in the 4G-byte address range in the 80386 - Core2 microprocessors

- The only other instruction using direct program addressing is the intersegment or far CALL instruction.
- Usually, the name of a memory address, called a *label*, refers to the location that is called or jumped to instead of the actual numeric address.
- When using a label with the CALL or JMP instruction, most assemblers select the best form of program addressing.

Relative Program Memory Addressing

- Not available in all early microprocessors, but it is available to this family of microprocessors.
- The term *relative* means "relative to the instruction pointer (IP)".
- The JMP instruction is a 1-byte instruction, with a 1-byte or a 2-byte displacement that adds to the instruction pointer.
- An example is shown in Figure 3–15.

Figure 3–15 A JMP [2] instruction. This instruction skips over the 2 bytes of memory that follow the JMP instruction.



Indirect Program Memory Addressing

- The microprocessor allows several forms of program indirect memory addressing for the JMP and CALL instructions.
- In 80386 and above, an extended register can be used to hold the address or indirect address of a relative JMP or CALL.
  - for example, the JMP EAX jumps to the location address by register EAX

- If a relative register holds the address, the jump is considered to be an indirect jump.
- For example, JMP [BX] refers to the memory location within the data segment at the offset address contained in BX.
  - at this offset address is a 16-bit number used as the offset address in the intrasegment jump
  - this type of jump is sometimes called an *indirect-indirect* or *double-indirect jump*
- Figure 3–16 shows a jump table that is stored, beginning at memory location TABLE.

Figure 3–16 A jump table that stores addresses of various programs. The exact address chosen from the TABLE is determined by an index stored with the jump instruction.


```
TABLE   DW   LOC0
        DW   LOC1
        DW   LOC2
        DW   LOC3
```

3–3 STACK MEMORY-ADDRESSING MODES

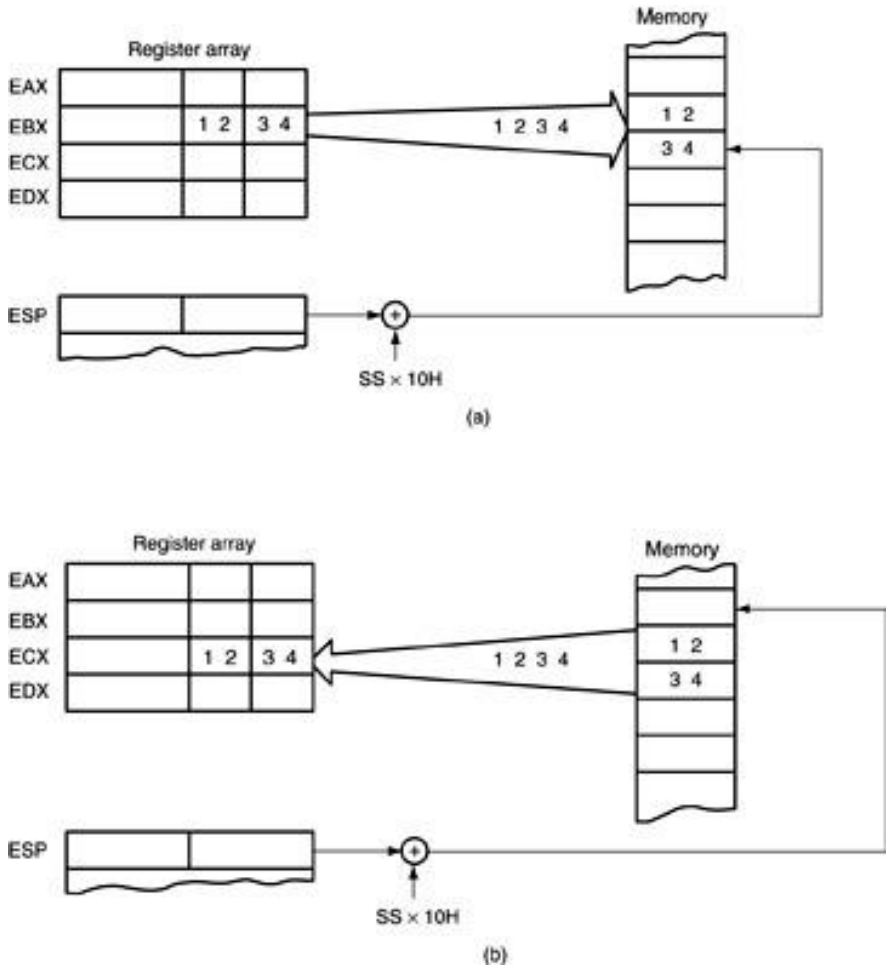
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The stack plays an important role in all microprocessors.
  - holds data temporarily and stores return addresses used by procedures
- Stack memory is LIFO (**last-in, first-out**) memory
  - describes the way data are stored and removed from the stack

- Data are placed on the stack with a **PUSH instruction**; removed with a **POP instruction.**
- Stack memory is maintained by two registers:
  - the stack pointer (SP or ESP)
  - the stack segment register (SS)
- Whenever a word of data is pushed onto the stack, the high-order 8 bits are placed in the location addressed by SP – 1.
  - low-order 8 bits are placed in the location addressed by SP – 2

- The SP is decremented by 2 so the next word is stored in the next available stack location.
  - the SP/ESP register always points to an area of memory located within the stack segment.
- In protected mode operation, the SS register holds a selector that accesses a descriptor for the base address of the stack segment.
- When data are popped from the stack, the low-order 8 bits are removed from the location addressed by SP.
  - high-order 8 bits are removed; the SP register is incremented by 2

Figure 3–17 The PUSH and POP instructions: (a) PUSH BX places the contents of BX onto the stack; (b) POP CX removes data from the stack and places them into CX. Both instructions are shown after execution.



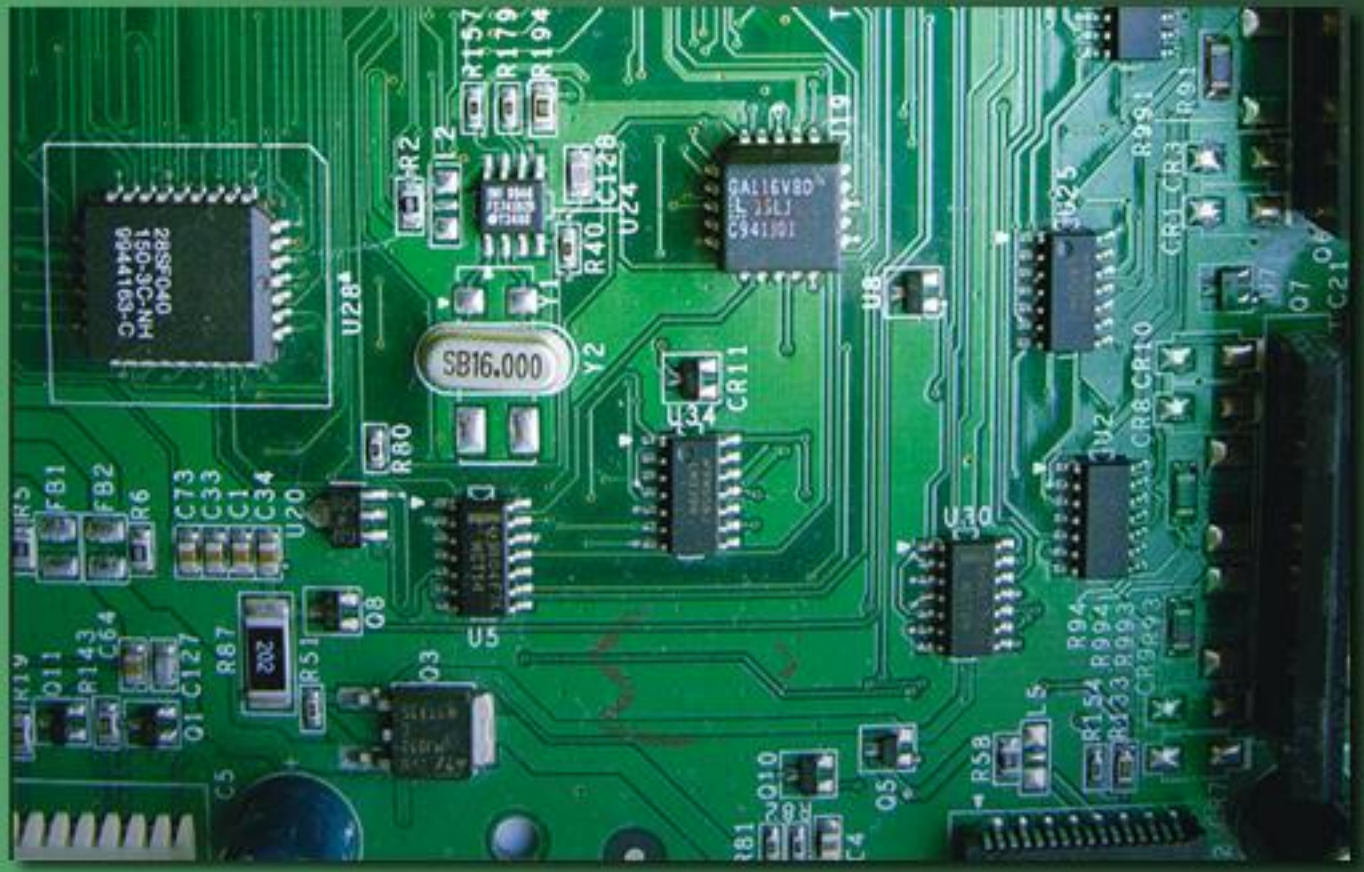
- Note that PUSH and POP store or retrieve words of data—never bytes—in 8086 - 80286.
- 80386 and above allow words or doublewords to be transferred to and from the stack.
- Data may be pushed onto the stack from any 16-bit register or segment register.
  - in 80386 and above, from any 32-bit extended register
- Data may be popped off the stack into any register or any segment register except CS.

- PUSHA and POPA instructions push or pop all except segment registers, on the stack.
- Not available on early 8086/8088 processors.
- 80386 and above allow extended registers to be pushed or popped.
  - 64-bit mode for Pentium and Core2 does not contain a PUSHA or POPA instruction

The Intel Microprocessors

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

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 4: Data Movement Instructions

4–1 MOV Revisited

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

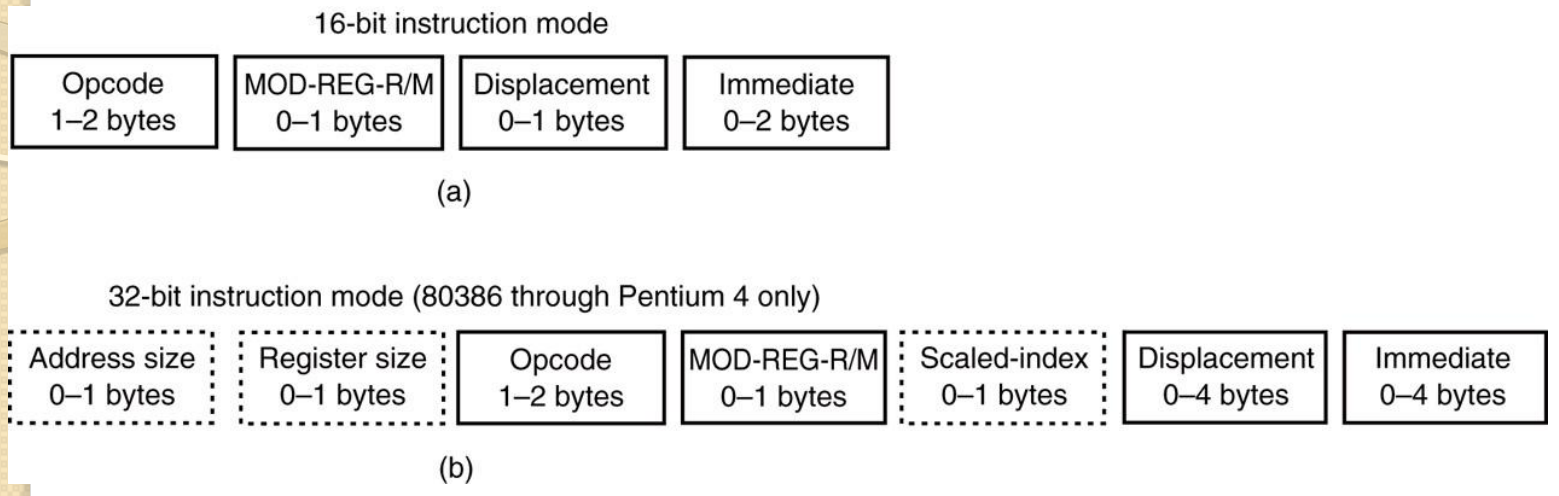
- In this chapter, the MOV instruction introduces machine language instructions available with various addressing modes and instructions.
- It may be necessary to interpret machine language programs generated by an assembler.
- Occasionally, machine language patches are made by using the DEBUG program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

Machine Language

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

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



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

The Opcode

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

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

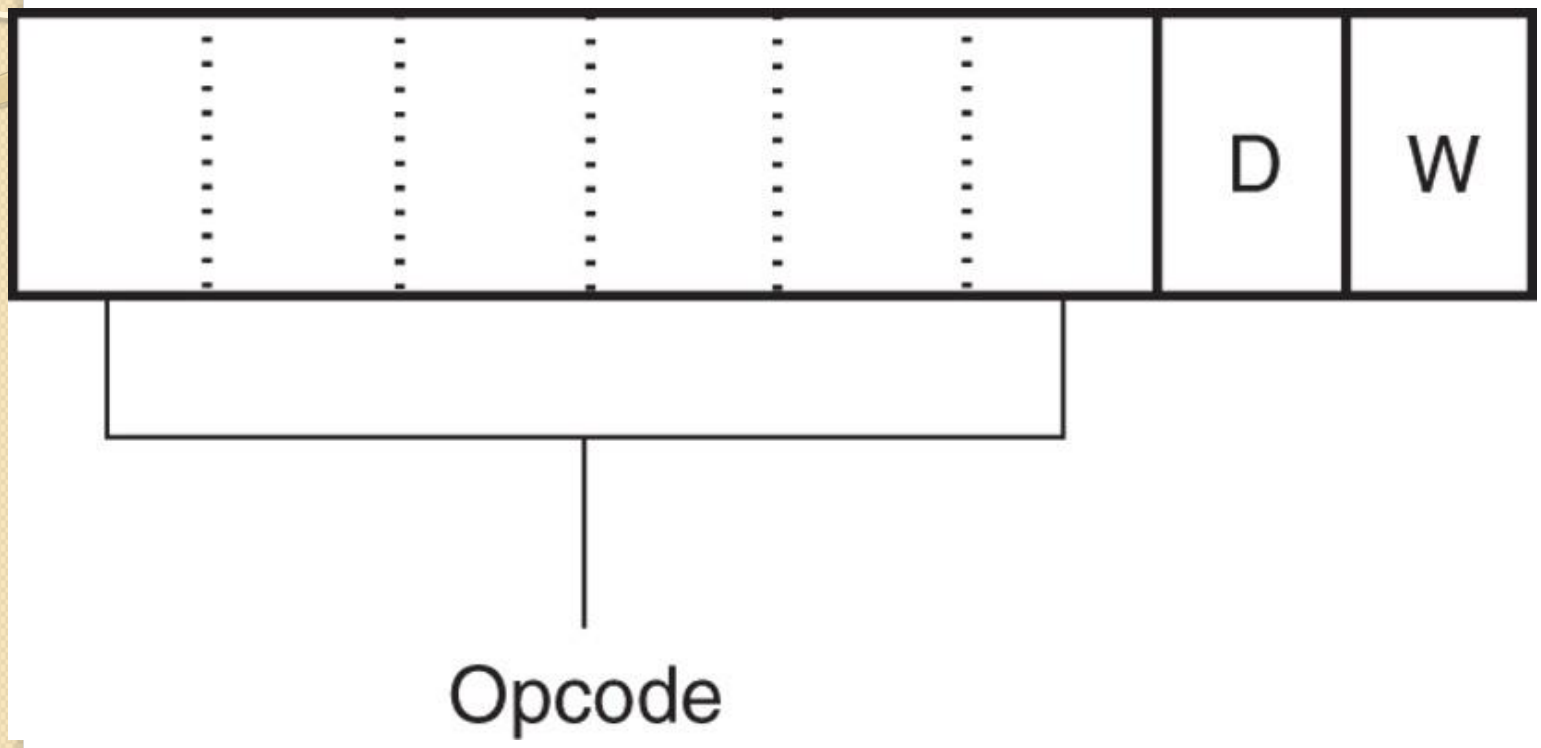
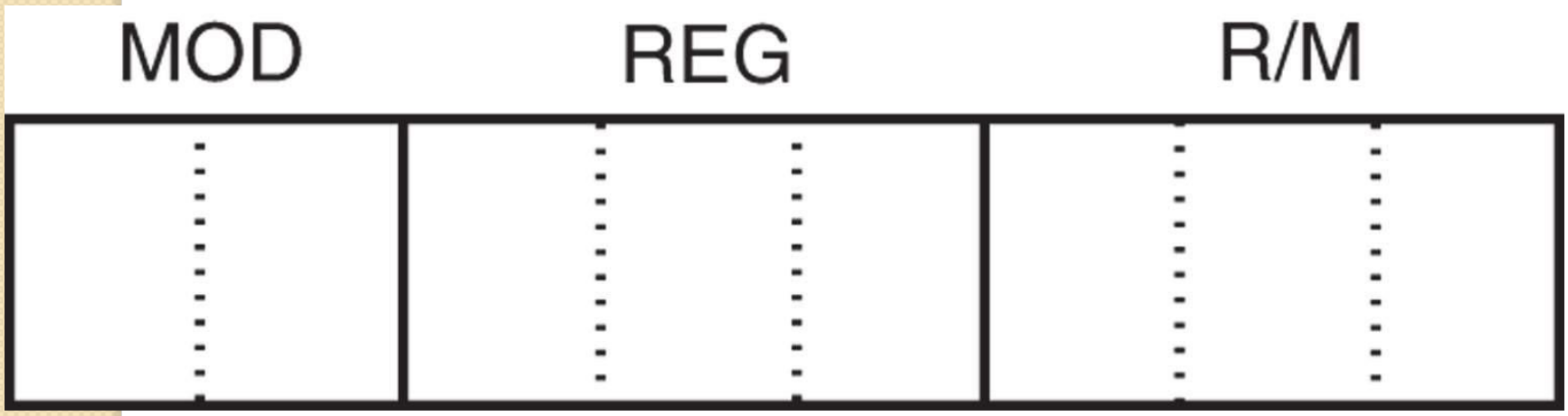


Figure 4–3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.



MOD Field

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

- All 8-bit displacements are sign-extended into 16-bit displacements when the processor executes the instruction.
  - if the 8-bit displacement is 00H–7FH (positive), it is sign-extended to 0000H–007FH before adding to the offset address
  - if the 8-bit displacement is 80H–FFH (negative), it is sign-extended to FF80H–FFFFH
- Some assembler programs do not use the 8-bit displacements and in place default to all 16-bit displacements.

Register Assignments

- Suppose a 2-byte instruction, 8BECH, appears in a machine language program.
  - neither a 67H (operand address-size override prefix) nor a 66H (register-size override prefix) appears as the first byte, thus the first byte is the opcode
- In 16-bit mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2, as illustrated in Figure 4–4.

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

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

- the opcode is 100010, a MOV instruction
- D and W bits are a logic 1, so a word moves into the destination register specified in the REG field
- REG field contains 101, indicating register BP, so the MOV instruction moves data into register BP

R/M Memory Addressing

- If the MOD field contains a 00, 01, or 10, the R/M field takes on a new meaning.
- Figure 4–5 illustrates the machine language version of the 16-bit instruction MOV DL, [DI] or instruction (8AI5H).
- This instruction is 2 bytes long and has an opcode 100010, D=1 (to REG from R/M), W=0 (byte), MOD=00 (no displacement), REG=010 (DL), and R/M=101 ([DI]).

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

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

- If the instruction changes to MOV DL, [DI+1], the MOD field changes to 01 for 8-bit displacement
- first 2 bytes of the instruction remain the same
- instruction now becomes 8A5501H instead of 8A15H

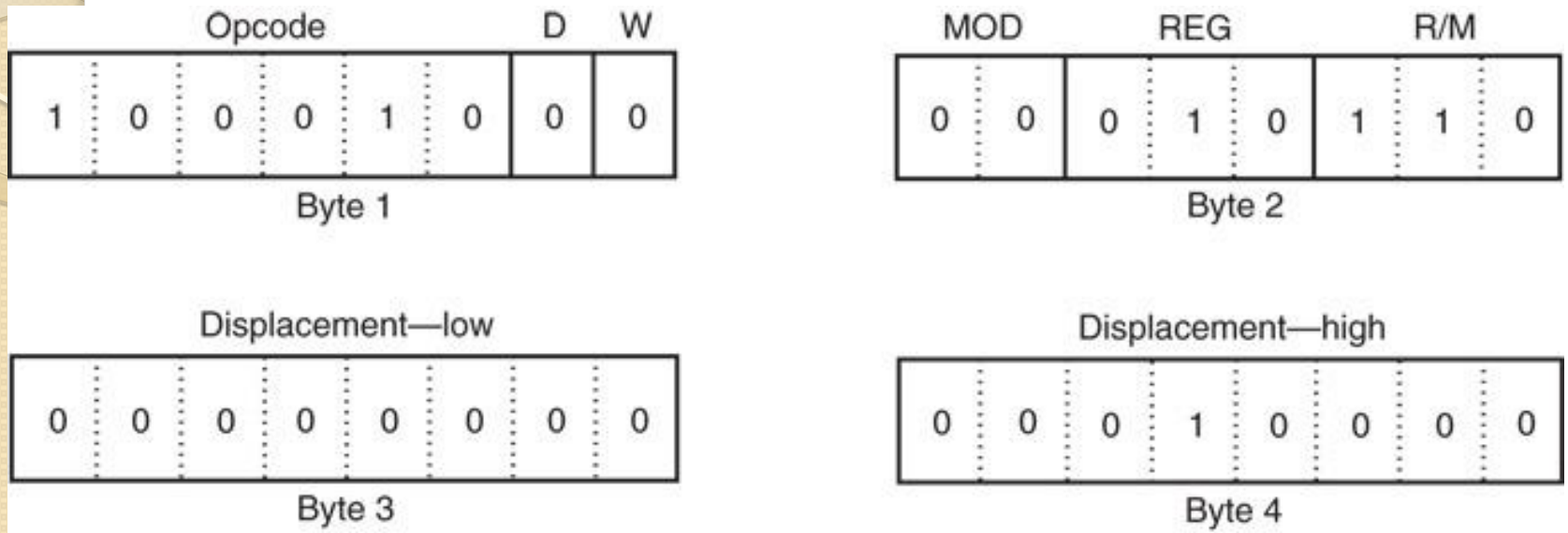
- Because the MOD field contains a 11, the R/M field also indicates a register.
- = 100(SP); therefore, this instruction moves data from SP into BP.
  - written in symbolic form as a MOV BP,SP instruction
- The assembler program keeps track of the register- and address-size prefixes and the mode of operation.

Special Addressing Mode

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

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

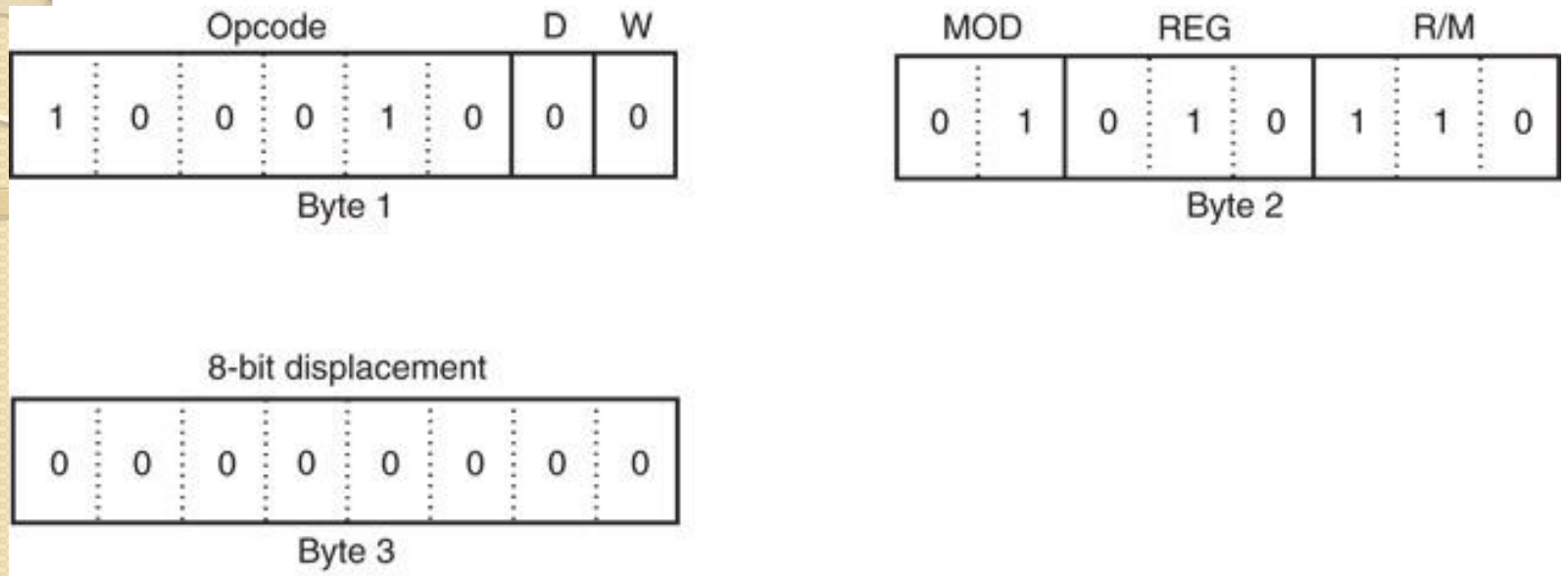
Figure 4–6 The MOV [1000H],DI instruction uses the special addressing mode.



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

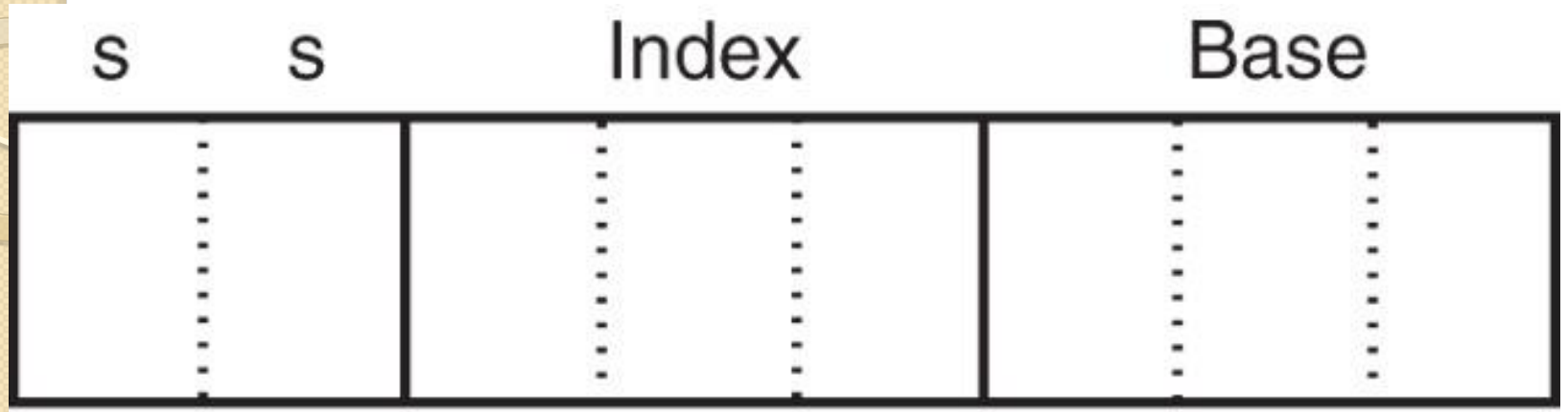
- actual form of the MOV [BP],DL instruction
- a 3-byte instruction with a displacement of 00H

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
- A scaled-index byte indicates additional forms of scaled-index addressing.
  - mainly used when two registers are added to specify the memory address in an instruction
- A scaled-index instruction has 2^{15} (32K) possible combinations.

- 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 = × 1

01 = × 2

10 = × 4

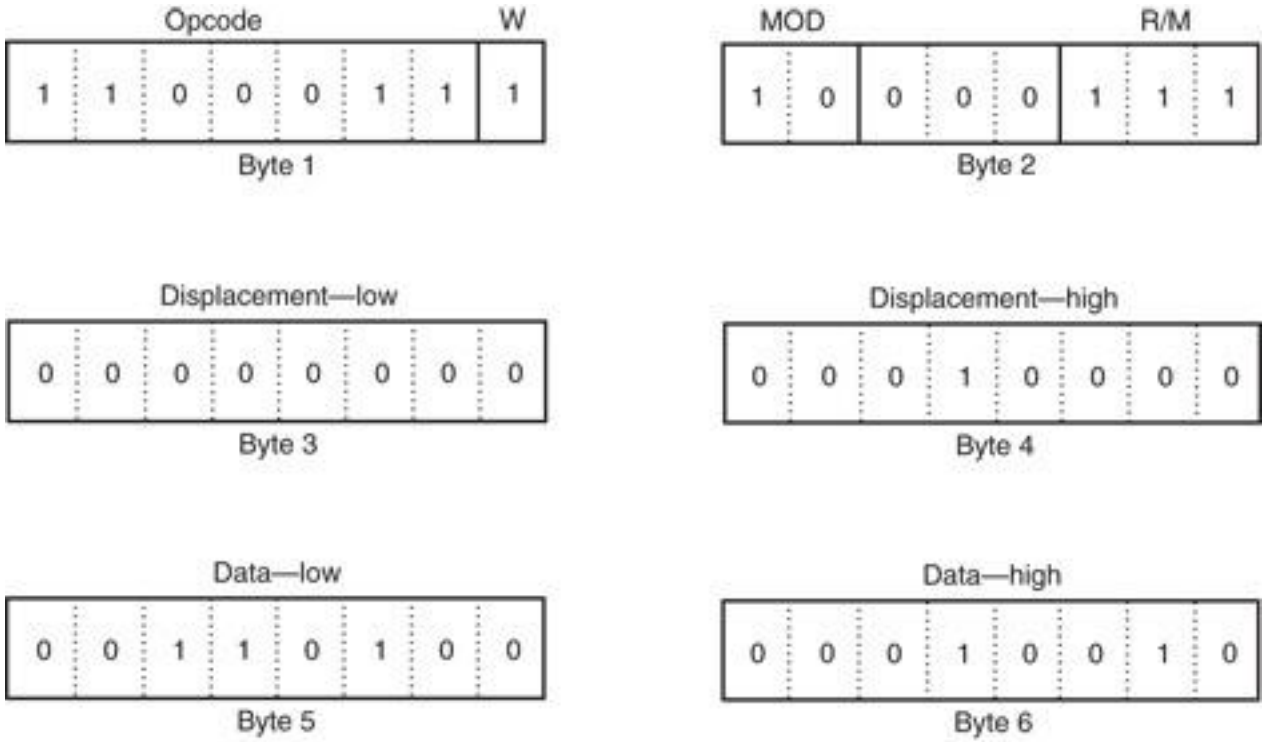
11 = × 8

– the index and base fields both contain register numbers

An Immediate Instruction

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

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



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

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

Segment MOV Instructions

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

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

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

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

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

The 64-Bit Mode for the Pentium 4 and Core2

- In 64-bit mode, a prefix called REX (*register extension*) is added.
  - encoded as a 40H–4FH, follows other prefixes; placed immediately 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

- 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.

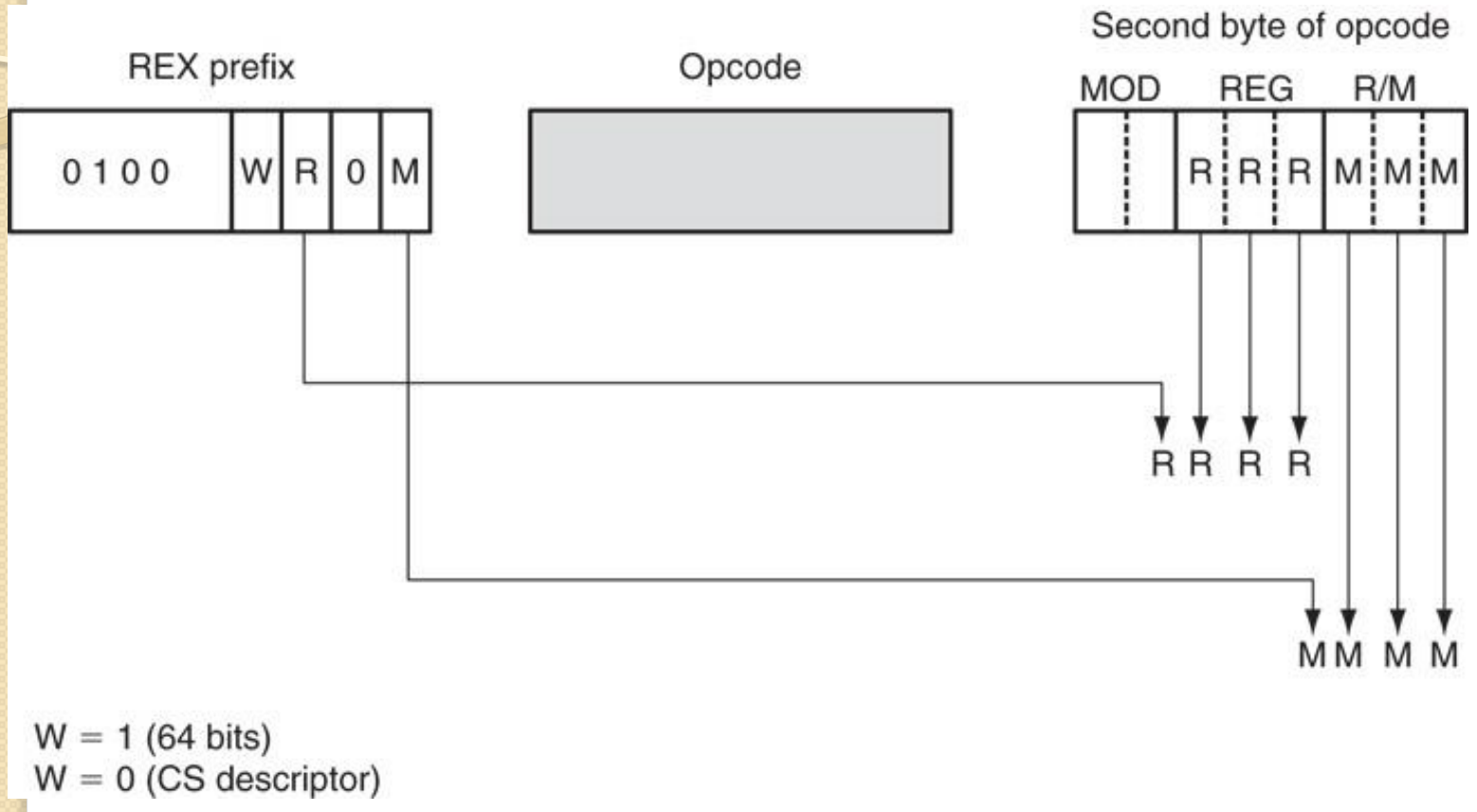
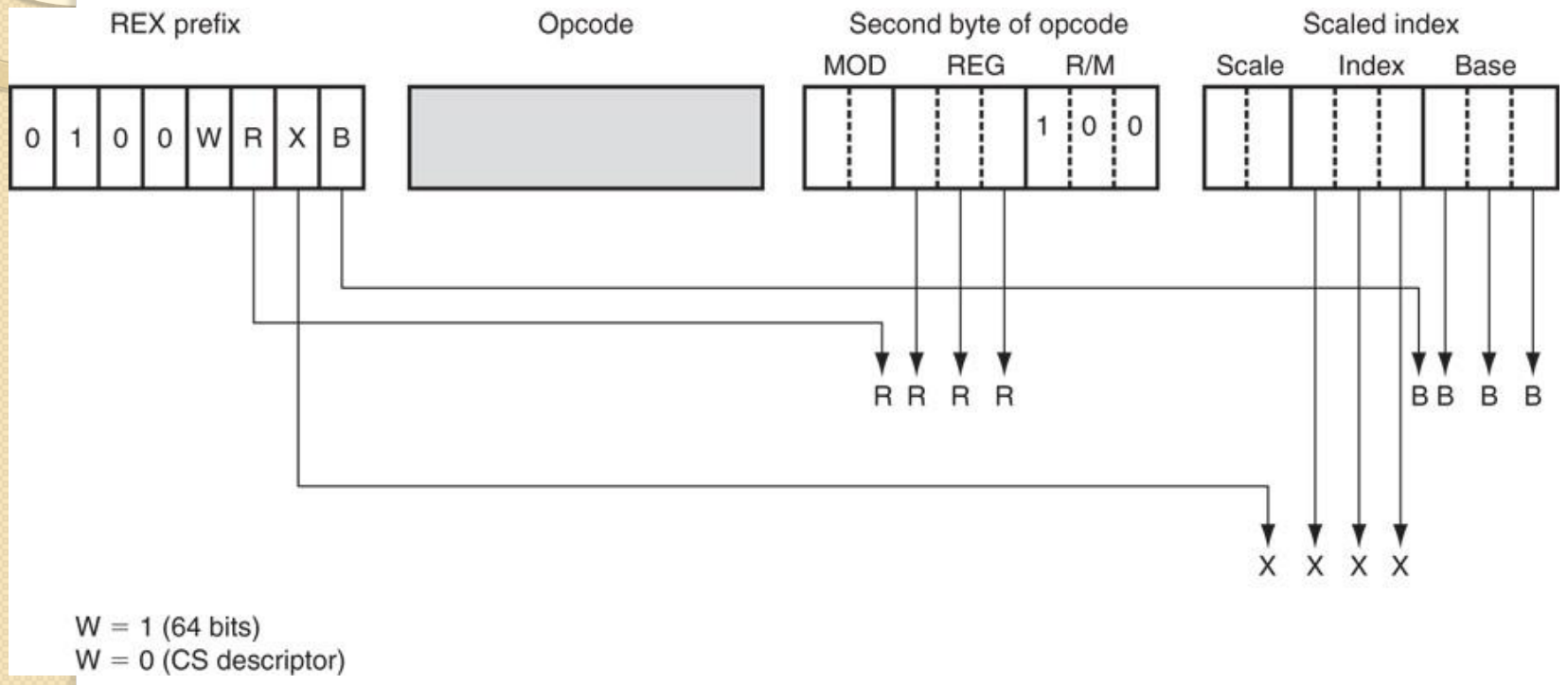


Figure 4–12 The scaled-index byte and REX prefix for 64-bit operations.



4–2 PUSH/POP

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

PUSH and POP instructions

TABLE 3–11 Example PUSH and POP instructions.

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

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

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

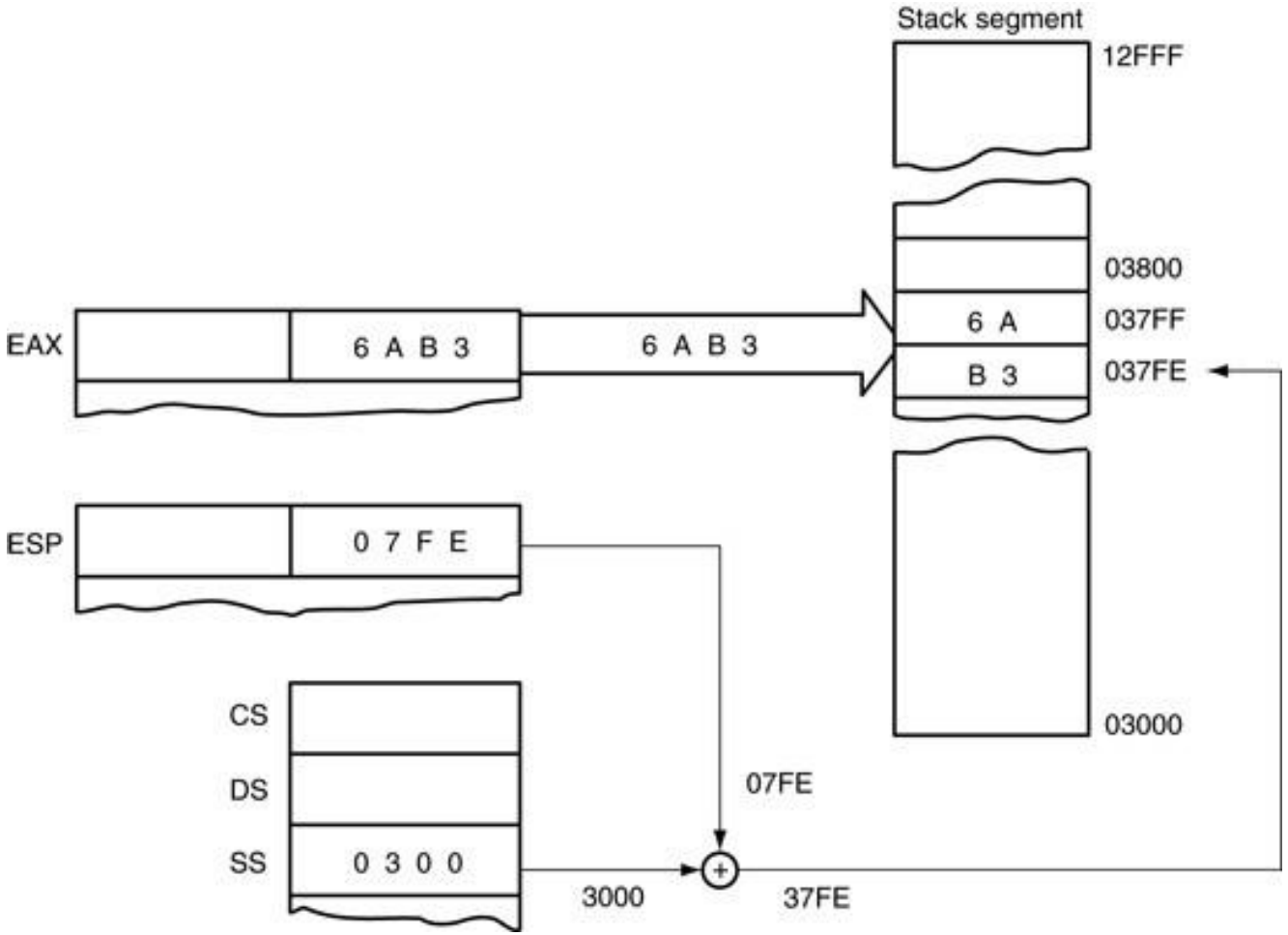
PUSH

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

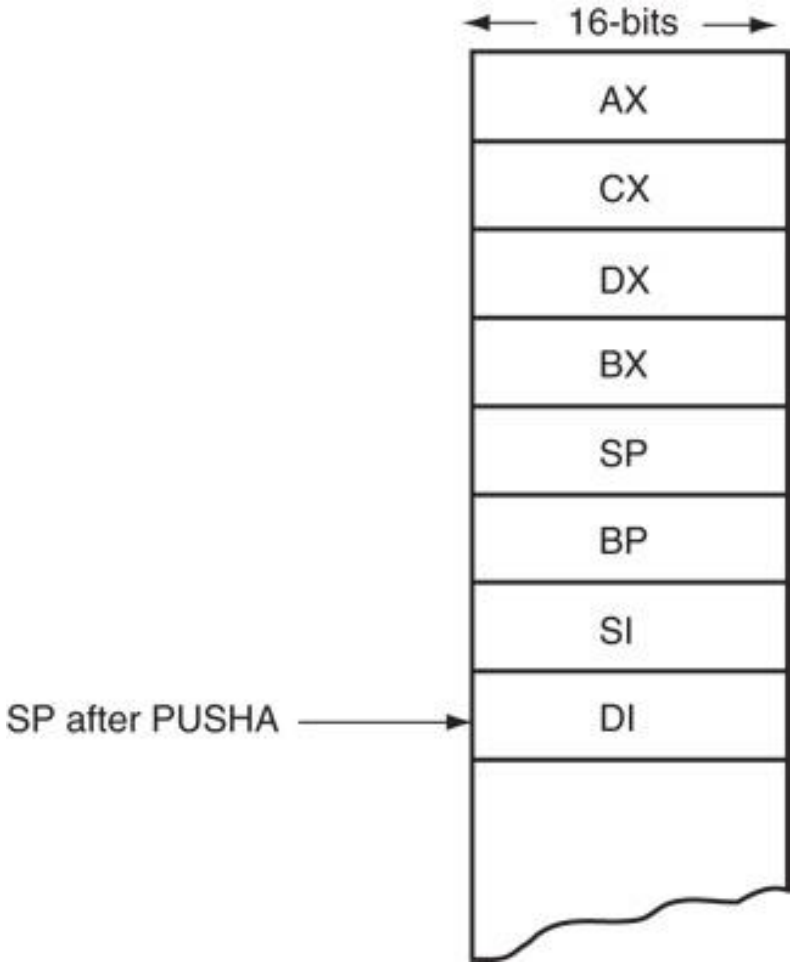
- PUSHF (**push flags**) instruction copies the contents of the flag register to the stack.
- PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set in 80386 - Pentium 4.
  - PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4

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



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

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

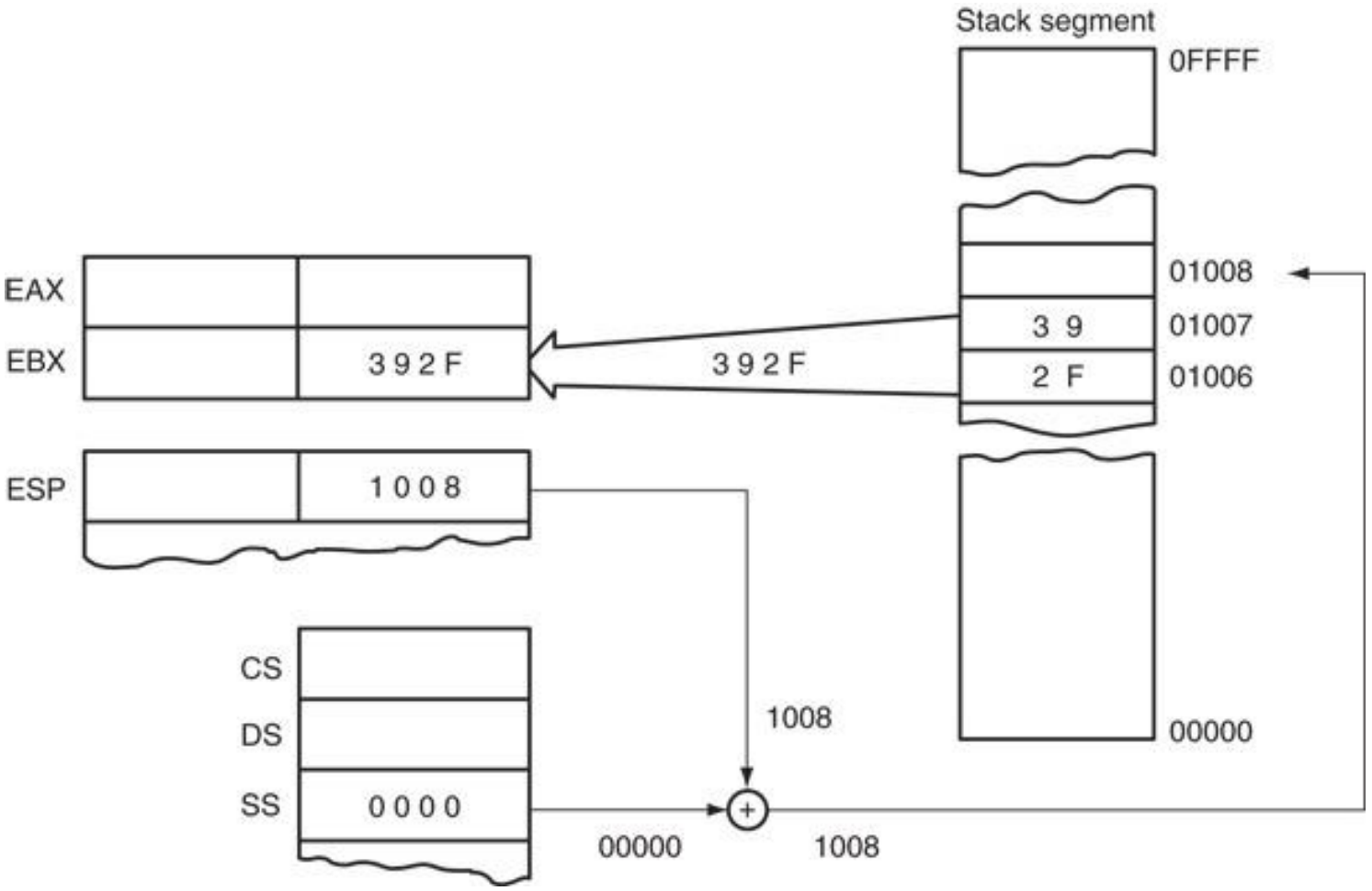


POP

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

- POPA (pop all) removes 16 bytes of data from the stack and places them into the following registers, in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX.
  - reverse order from placement on the stack by PUSHA instruction, causing the same data to return to the same registers
- Figure 4–15 shows how the POP BX instruction removes data from the stack and places them into register BX.

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

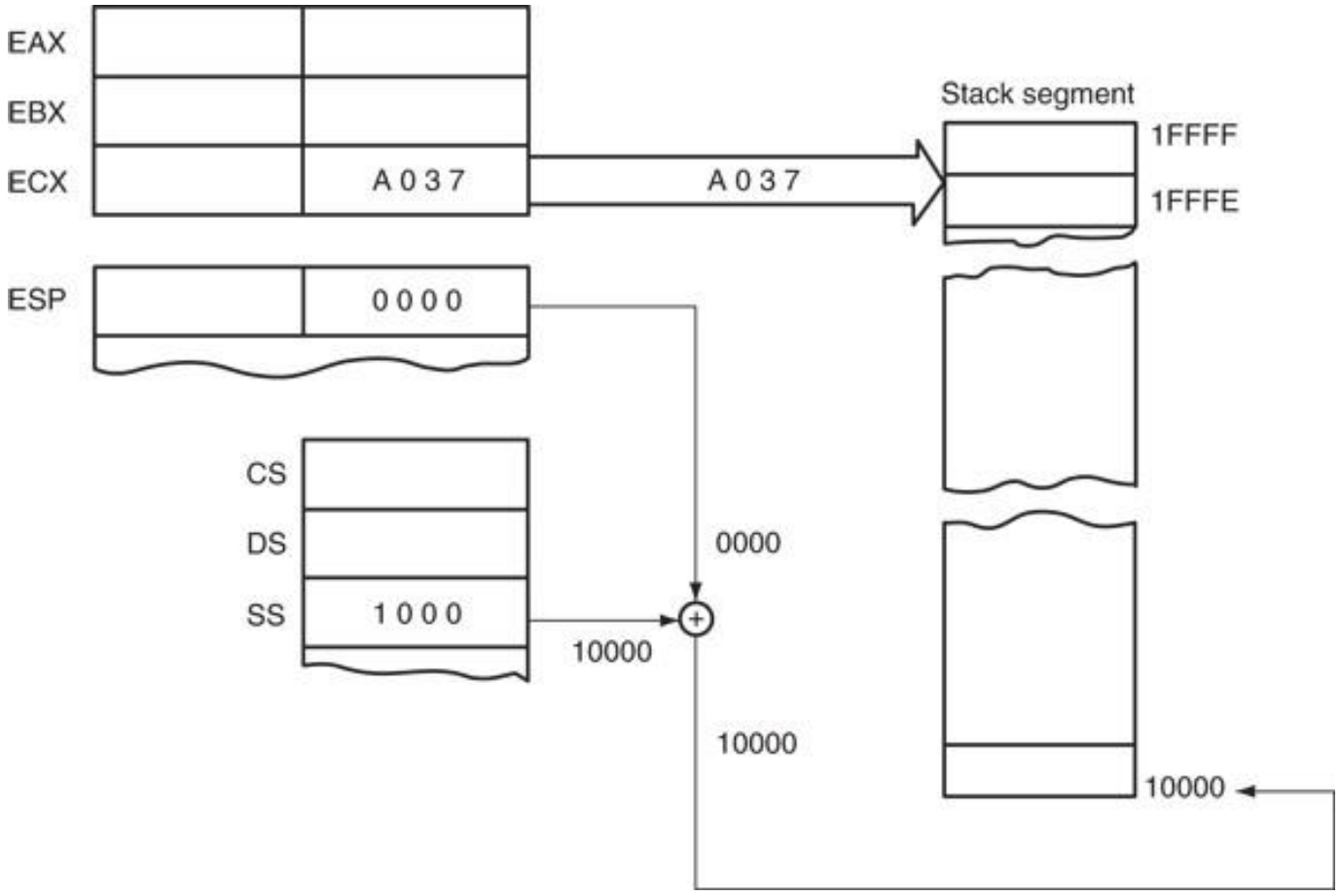


Initializing the Stack

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

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



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

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

LEA

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

- Why is LEA (**LOAD EFFECTIVE ADDRESS**) instruction available if OFFSET accomplishes the same task?
  - OFFSET functions with, and is more efficient than LEA instruction, for simple operands such as LIST
  - Microprocessor takes longer to execute the LEA BX,LIST instruction the MOV BX,OFFSET LIST
- The MOV BX,OFFSET LIST instruction is actually assembled as a move immediate instruction and is more efficient.

4–3 LOAD EFFECTIVE ADDRESS

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

Load-effective address instructions

END

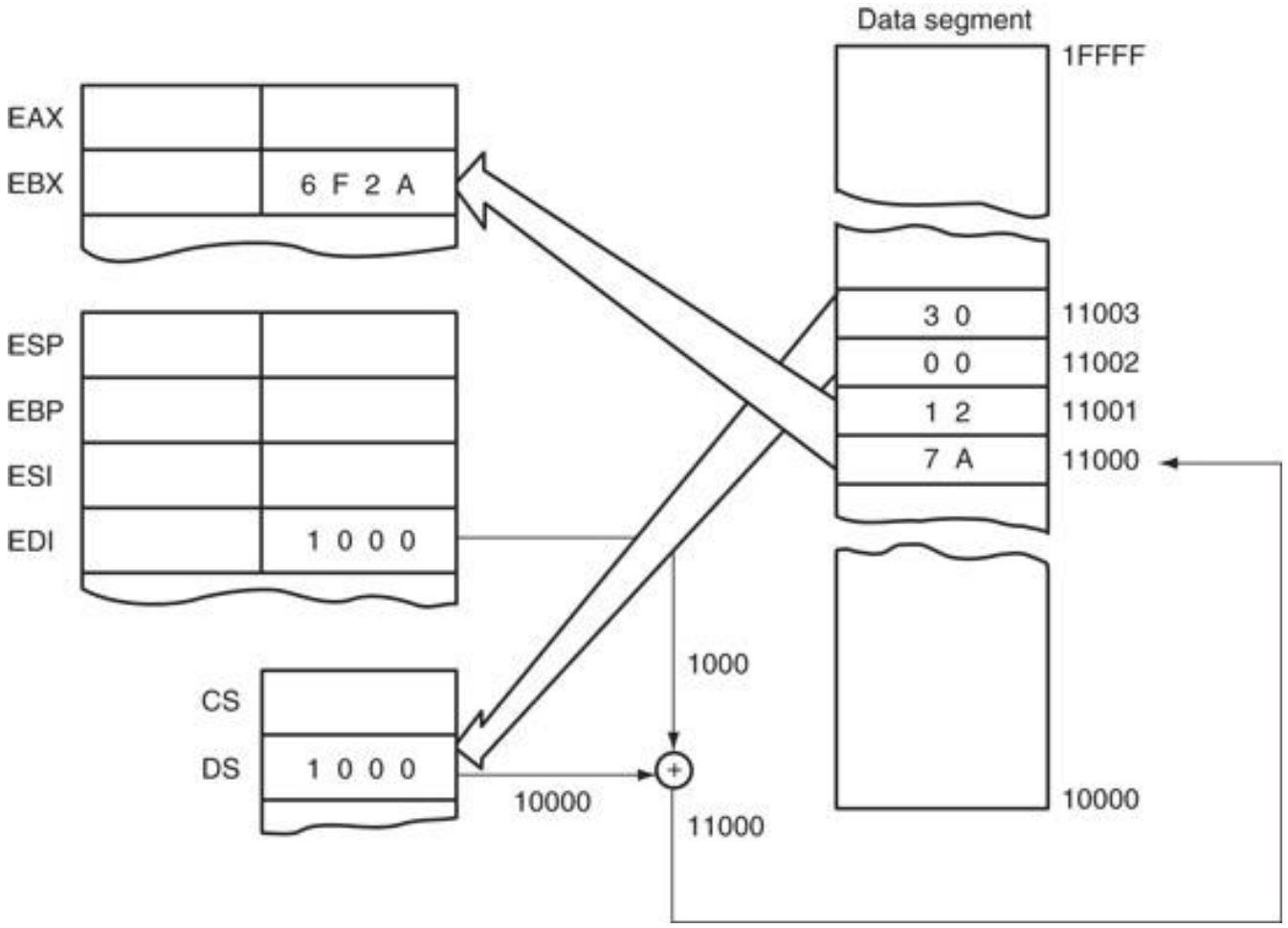
TABLE 4–10 Load-effective address instructions.

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

LDS, LES, LFS, LGS, and LSS

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

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



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

- A far address can be stored in memory by the assembler.
- The most useful of the load instructions is the LSS instruction.
  - after executing some dummy instructions, the old stack area is reactivated by loading both SS and SP with the LSS instruction
- CLI (**disable interrupt**) and STI (**enable interrupt**) instructions must be included to disable interrupts.

4–4 STRING DATA TRANSFERS

- Five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS.
- Each allows data transfers as a single byte, word, or doubleword.
- Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions.

The Direction Flag

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

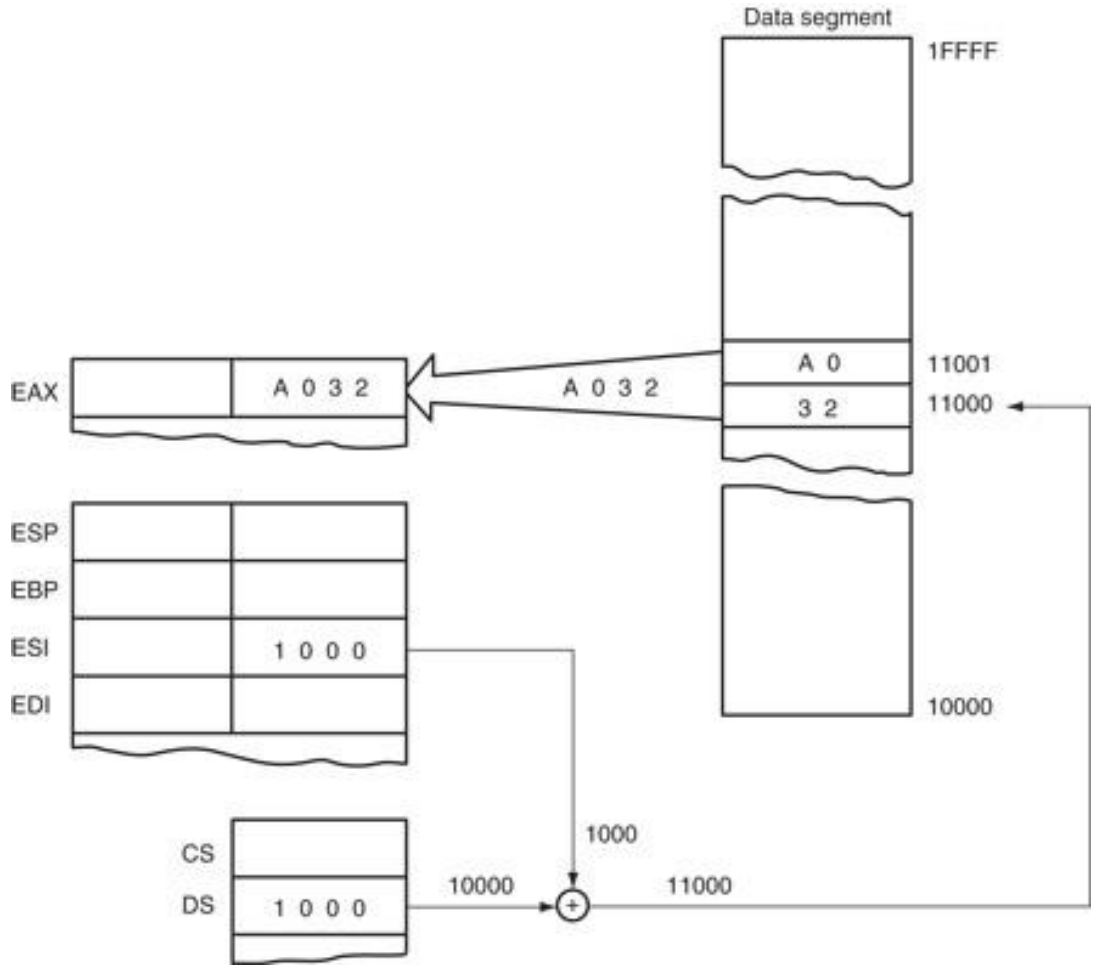
DI and SI

- During execution of string instruction, memory accesses occur through DI and SI registers.
  - DI offset address accesses data in the extra segment for all string instructions that use it
  - SI offset address accesses data by default in the data segment
- Operating in 32-bit mode EDI and ESI registers are used in place of DI and SI.
  - this allows string using any memory location in the entire 4G-byte protected mode address space

LODS

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

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



STOS

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

STOS with a REP

- The **repeat prefix** (REP) is added to any string data transfer instruction except LODS.
  - REP prefix causes CX to decrement by 1 each time the string instruction executes; after CX decrements, the string instruction repeats
- 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.

MOVS

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

INS

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

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

OUTS

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

4–5 MISCELLANEOUS DATA TRANSFER INSTRUCTIONS

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

XCHG

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

LAHF and SAHF

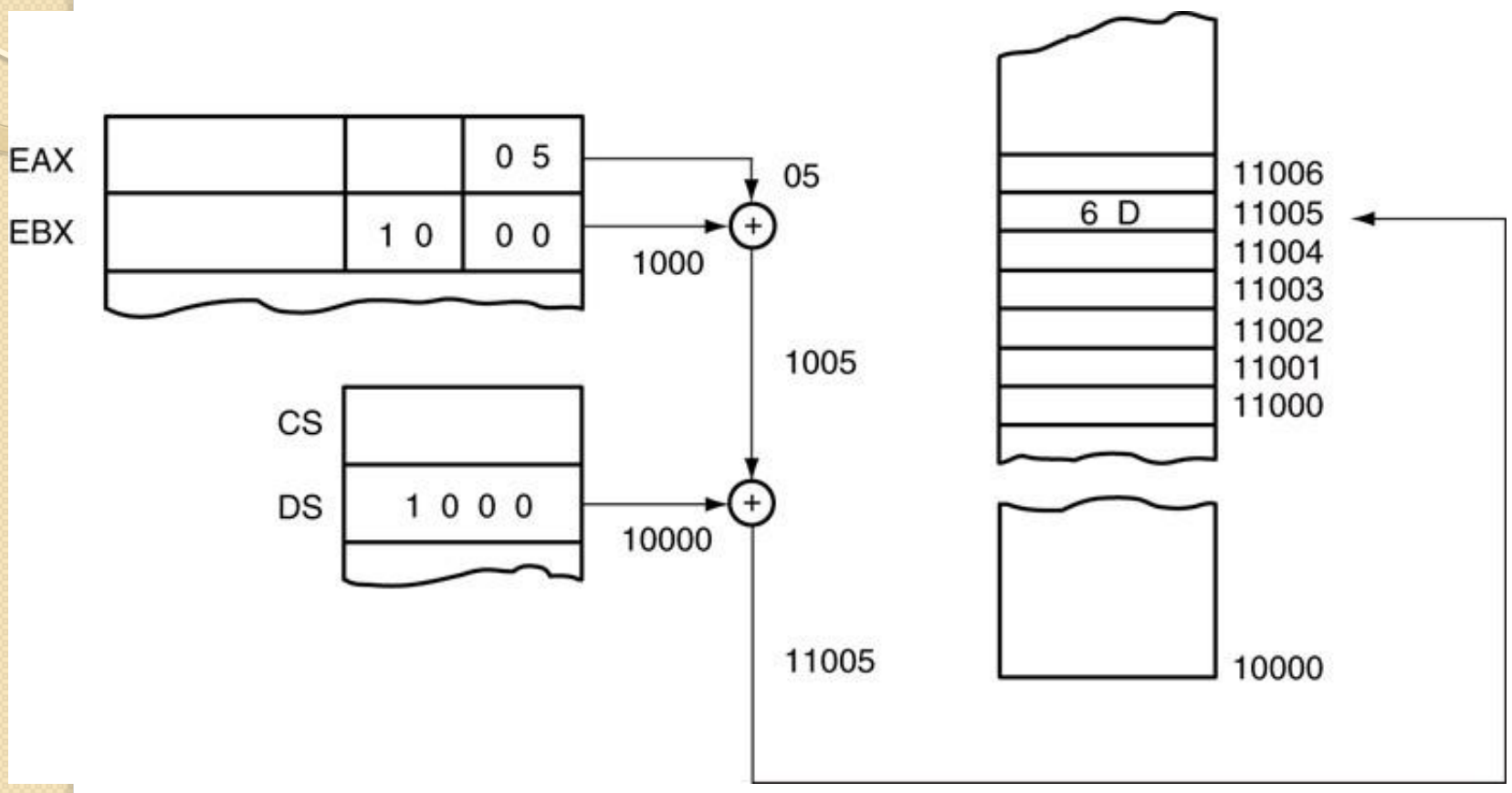
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

XLAT

- Converts the contents of the AL register into a number stored in a memory table.
  - performs the direct table lookup technique often used to convert one code to another
- An XLAT instruction first adds the contents of AL to BX to form a memory address within the data segment.
  - copies the contents of this address into AL
  - only instruction adding an 8-bit to a 16-bit number

Figure 4–19 The operation of the XLAT instruction at the point just before 6DH is loaded into AL.



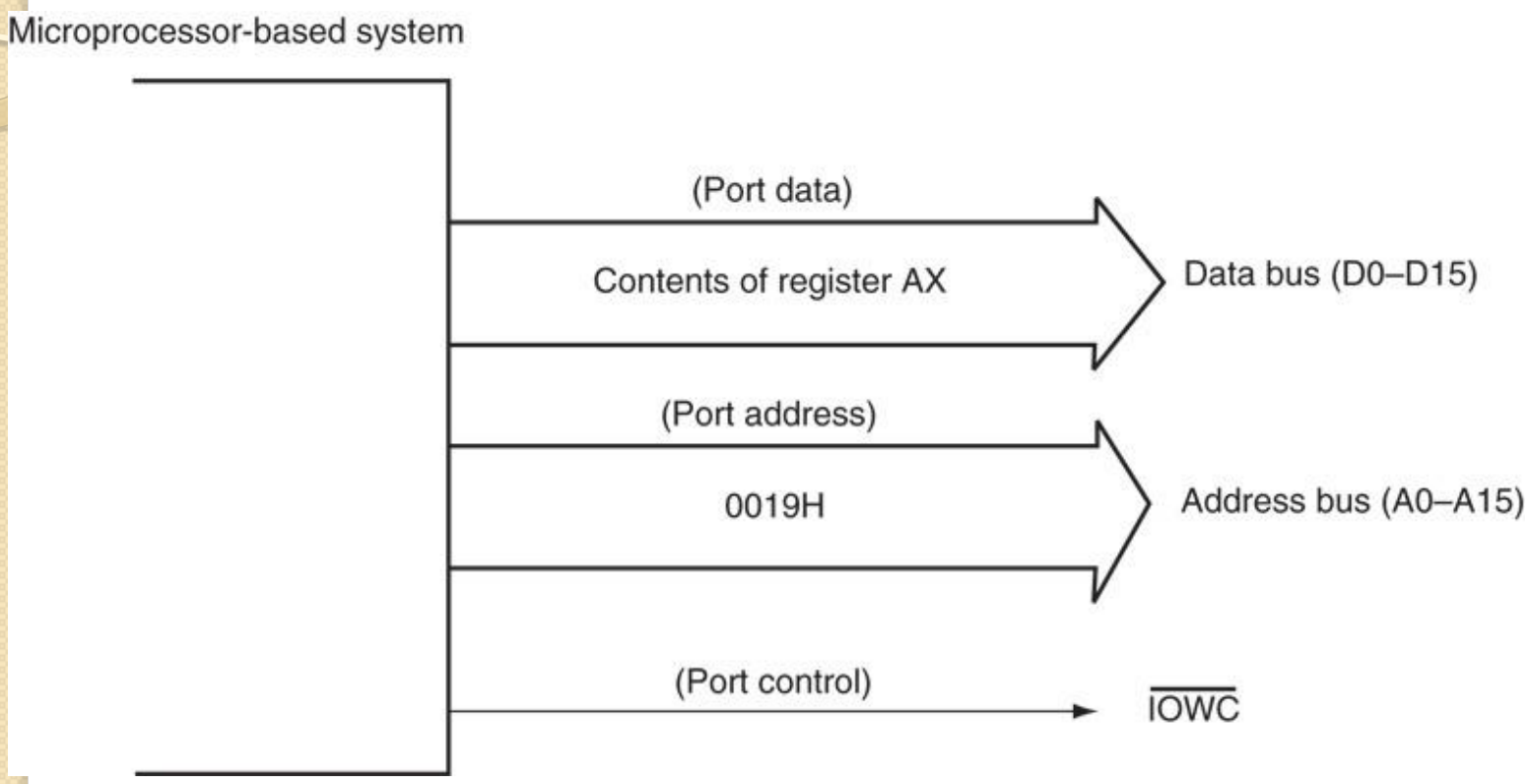
IN and OUT

- IN & OUT instructions perform I/O operations.
- Contents of AL, AX, or EAX are transferred only between I/O device and microprocessor.
  - an IN instruction transfers data from an external I/O device into AL, AX, or EAX
  - an OUT transfers data from AL, AX, or EAX to an external I/O device
- Only the 80386 and above contain EAX

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

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

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



BSWAP

- Takes the contents of any 32-bit register and swaps the first byte with the fourth, and the second with the third.
  - BSWAP (**byte swap**) is available only in 80486–Pentium 4 microprocessors
- This instruction is used to convert data between the big and little endian forms.
- In 64-bit operation for the Pentium 4, all 8 bytes in the selected operand are swapped.

CMOV

- Many variations of the CMOV instruction.
  - these move the data only if the condition is true
- CMOVZ instruction moves data only if the result from some prior instruction was a zero.
  - destination is limited to only a 16- or 32-bit register, but the source can be a 16- or 32-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

4–6 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

4–7 ASSEMBLER DETAIL

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The assembler can be used 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
- In most cases, the inline assembler found in Visual is used for developing assembly code for use in a program
  - occasions require separate assembly modules using the assembler

Directives

- 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.
- Complex sections of assembly code are still written using MASM.

Storing Data in a Memory Segment

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

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

ASSUME, EQU, and ORG

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

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

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

PROC and ENDP

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

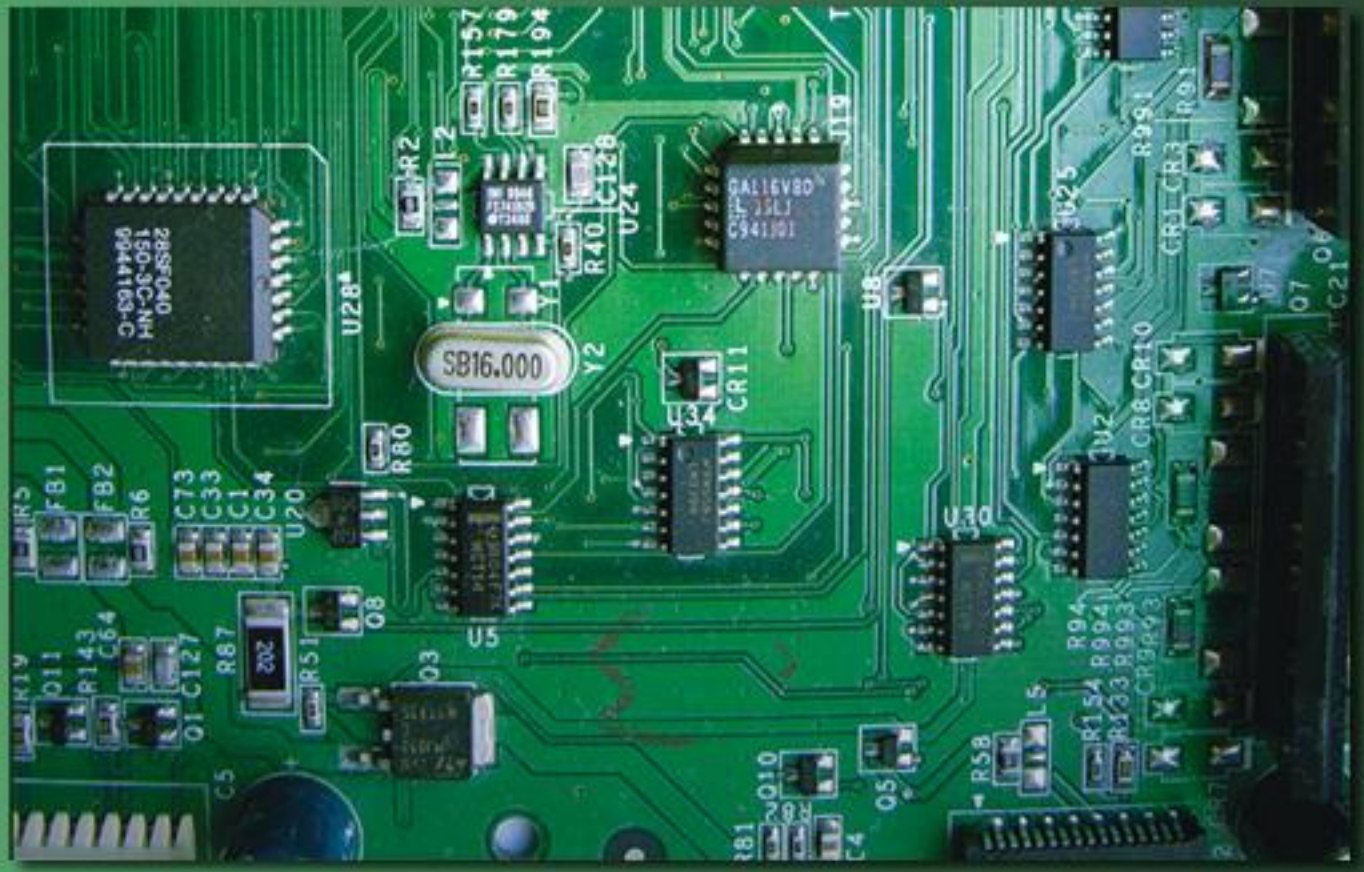
- Indicate start and end of a procedure (subroutine).
  - they *force structure* because the procedure is clearly defined
- If structure is to be violated for whatever reason, use the CALLF, CALLN, RETF, and RETN instructions.
- Both the PROC and ENDP directives require a label to indicate the name of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

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

The Intel Microprocessors

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

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 5: Arithmetic and Logic Instructions

Introduction

- We examine the arithmetic and logic instructions. The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement.
- The logic instructions include AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST).

5-1 ADDITION, SUBTRACTION AND COMPARISON

- The bulk of the arithmetic instructions found in any microprocessor include addition, subtraction, and comparison.
- Addition, subtraction, and comparison instructions are illustrated.
- Also shown are their uses in manipulating register and memory data.

Addition

- Addition (ADD) appears in many forms in the microprocessor.
- A second form of addition, called **add-with-carry**, is introduced with the ADC instruction.
- The only types of addition *not* allowed are memory-to-memory and segment register.
  - segment registers can only be moved, pushed, or popped
- Increment instruction (INC) is a special type of addition that adds 1 to a number.

Register Addition

- When arithmetic and logic instructions execute, contents of the flag register change.
  - interrupt, trap, and other flags do not change
- Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags.

Immediate Addition

- Immediate addition is employed whenever constant or known data are added.

Memory-to-Register Addition

- Moves memory data to be added to the AL (and other) register.

Array Addition

- Memory arrays are sequential lists of data.

Array Addition

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Sequential lists of data.
- A sequence of instructions written 80386 shows scaled-index form addressing to add elements 3, 5, and 7 of an area of memory called ARRAY.
- EBX is loaded with the address ARRAY, and ECX holds the array element number.
- The scaling factor is used to multiply the contents of the ECX register by 2 to address words of data.

Increment Addition

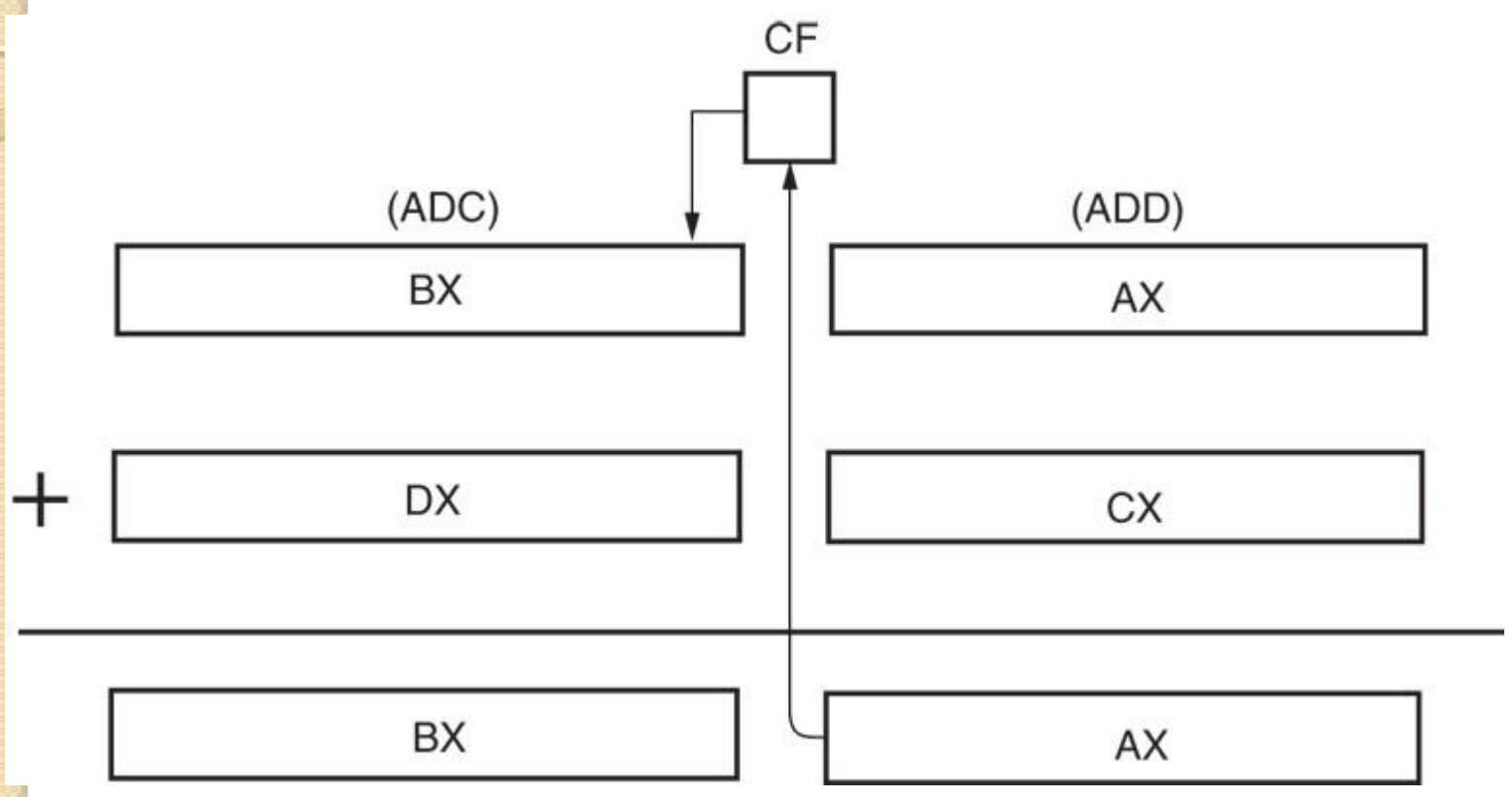
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The INC instruction adds 1 to any register or memory location, except a segment register.
- The size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives.
- The assembler program cannot determine if the INC [DI] instruction is a byte-, word-, or doubleword-sized increment.

Addition-with-Carry

- ADC) adds the bit in the carry flag (C) to the operand data.
  - mainly appears in software that adds numbers wider than 16 or 32 bits in the 80386–Core2
  - like ADD, ADC affects the flags after the addition
- Figure 5–1 illustrates this so placement and function of the carry flag can be understood.
  - cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers

Figure 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.



Exchange and Add for the 80486–Core2 Processors

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Exchange and add (XADD) appears in 80486 and continues through the Core2.
- XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition.
  - after the addition takes place, the original value of the destination is copied into the source operand
- One of the few instructions that change the source.

Subtraction

- Many forms of subtraction (SUB) appear in the instruction set.
  - these use any addressing mode with 8-, 16-, or 32-bit data
  - a special form of subtraction (decrement, or DEC) subtracts 1 from any register or memory location
- Numbers that are wider than 16 bits or 32 bits must occasionally be subtracted.
  - the **subtract-with-borrow instruction** (SBB) performs this type of subtraction

Register Subtraction

- After each subtraction, the microprocessor modifies the contents of the flag register.
  - flags change for most arithmetic/logic operations

Immediate Subtraction

- The microprocessor also allows immediate operands for the subtraction of constant data.

Decrement Subtraction

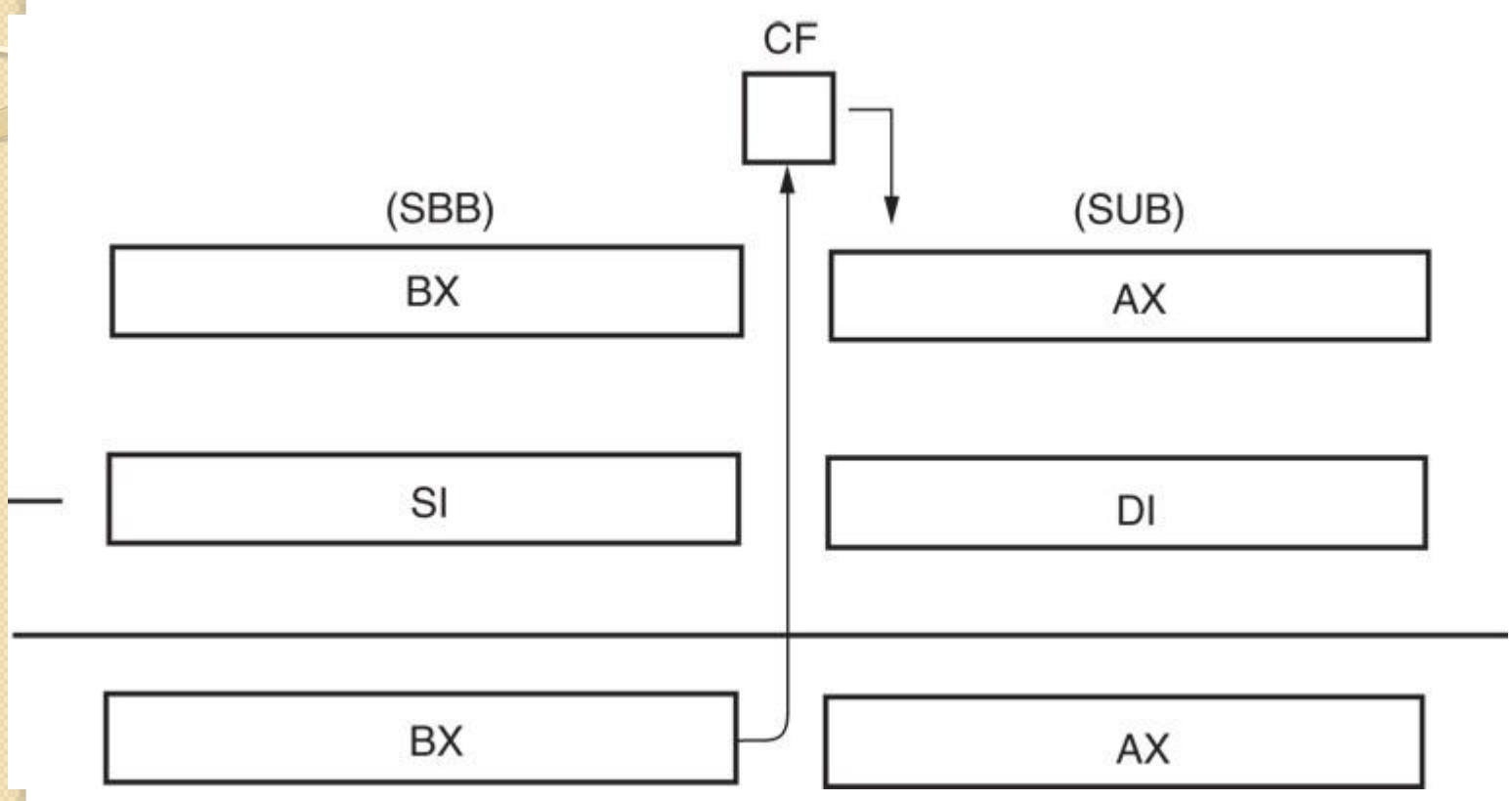
- Subtracts 1 from a register/memory location.

Subtraction-with-Borrow

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference.
  - most common use is subtractions wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2.
  - wide subtractions require borrows to propagate through the subtraction, just as wide additions propagate the carry

Figure 5–2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.



Comparison

- The comparison instruction (CMP) is a subtraction that changes only the flag bits.
  - destination operand never changes
- Useful for checking the contents of a register or a memory location against another value.
- A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

Compare and Exchange (80486–Core2 Processors Only)

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Compare and exchange instruction (CMPXCHG) compares the destination operand with the accumulator.
  - found only in 80486 - Core2 instruction sets
- If they are equal, the source operand is copied to the destination; if not equal, the destination operand is copied into the accumulator.
  - instruction functions with 8-, 16-, or 32-bit data

- CMPXCHG CX,DX instruction is an example of the compare and exchange instruction.
  - this compares the contents of CX with AX
  - if CX equals AX, DX is copied into AX; if CX is not equal to AX, CX is copied into AX
  - also compares AL with 8-bit data and EAX with 32-bit data if the operands are either 8- or 32-bit
- This instruction has a bug that will cause the operating system to crash.
  - more information about this flaw can be obtained at www.intel.com

5-2 MULTIPLICATION AND DIVISION

- Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions.
  - manufacturers were aware of this inadequacy, they incorporated multiplication and division into the instruction sets of newer microprocessors.
- Pentium–Core2 contains special circuitry to do multiplication in as few as one clocking period.
  - over 40 clocking periods in earlier processors

8-Bit Multiplication

- With 8-bit multiplication, the multiplicand is always in the AL register, signed or unsigned.
  - multiplier can be any 8-bit register or memory location
- Immediate multiplication is not allowed unless the special signed immediate multiplication instruction appears in a program.
- The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL.

Multiplication

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Performed on bytes, words, or doublewords,
  - can be signed (IMUL) or unsigned integer (MUL)
- Product after a multiplication always a double-width product.
  - two 8-bit numbers multiplied generate a 16-bit product; two 16-bit numbers generate a 32-bit; two 32-bit numbers generate a 64-bit product
  - in 64-bit mode of Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product

16-Bit Multiplication

- Word multiplication is very similar to byte multiplication.
- AX contains the multiplicand instead of AL.
  - 32-bit product appears in DX–AX instead of AX
- The DX register always contains the most significant 16 bits of the product; AX contains the least significant 16 bits.
- As with 8-bit multiplication, the choice of the multiplier is up to the programmer.

A Special Immediate 16-Bit Multiplication

- 80186 - Core2 processors can use a special version of the multiply instruction.
  - immediate multiplication must be signed;
  - instruction format is different because it contains three operands
- First operand is 16-bit destination register; the second a register/memory location with16-bit multiplicand; the third 8- or 16-bit immediate data used as the multiplier.

32-Bit Multiplication

- In 80386 and above, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers.
  - can be signed or unsigned by using IMUL and MUL instructions
- Contents of EAX are multiplied by the operand specified with the instruction.
- The 64 bit product is found in EDX–EAX, where EAX contains the least significant 32 bits of the product.

64-Bit Multiplication

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The result of a 64-bit multiplication in the Pentium 4 appears in the RDX:RAX register pair as a 128-bit product.
- Although multiplication of this size is relatively rare, the Pentium 4 and Core2 can perform it on both signed and unsigned numbers.

Division

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Occurs on 8- or 16-bit and 32-bit numbers depending on microprocessor.
  - signed (IDIV) or unsigned (DIV) integers
- Dividend is always a double-width dividend, divided by the operand.
- There is no immediate division instruction available to any microprocessor.
- In 64-bit mode Pentium 4 & Core2, divide a 128-bit number by a 64-bit number.

- A division can result in two types of errors:
  - attempt to divide by zero
  - other is a divide overflow, which occurs when a small number divides into a large number
- In either case, the microprocessor generates an interrupt if a divide error occurs.
- In most systems, a divide error interrupt displays an error message on the video screen.

8-Bit Division

- Uses AX to store the dividend divided by the contents of any 8-bit register or memory location.
- Quotient moves into AL after the division with AH containing a whole number remainder.
  - quotient is positive or negative; remainder always assumes sign of the dividend; always an integer

- Numbers usually 8 bits wide in 8-bit division .
  - the dividend must be converted to a 16-bit wide number in AX ; accomplished differently for signed and unsigned numbers
- CBW (**convert byte to word**) instruction performs this conversion.
- In 80386 through Core2, MOVSX sign-extends a number.

16-Bit Division

- Sixteen-bit division is similar to 8-bit division
  - instead of dividing into AX, the 16-bit number is divided into DX–AX, a 32-bit dividend
- As with 8-bit division, numbers must often be converted to the proper form for the dividend.
  - if a 16-bit unsigned number is placed in AX, DX must be cleared to zero
- In the 80386 and above, the number is zero-extended by using the MOVZX instruction.

32-Bit Division

- 80386 - Pentium 4 perform 32-bit division on signed or unsigned numbers.
  - 64-bit contents of EDX–EAX are divided by the operand specified by the instruction
    - leaving a 32-bit quotient in EAX
    - and a 32-bit remainder in EDX
- Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions.

64-Bit Division

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Pentium 4 operated in 64-bit mode performs 64-bit division on signed or unsigned numbers.
- The 64-bit division uses the RDX:RAX register pair to hold the dividend.
- The quotient is found in RAX and the remainder is in RDX after the division.

The Remainder

- Could be used to round the quotient or dropped to truncate the quotient.
- If division is unsigned, rounding requires the remainder be compared with half the divisor to decide whether to round up the quotient
- The remainder could also be converted to a fractional remainder.

5-3 BCD and ASCII Arithmetic

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The microprocessor allows arithmetic manipulation of both BCD (**binary-coded decimal**) and ASCII (**American Standard Code for Information Interchange**) data.
- BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic.

BCD Arithmetic

- Two arithmetic techniques operate with BCD data: addition and subtraction.
- DAA (**decimal adjust after addition**) instruction follows BCD addition,
- DAS (**decimal adjust after subtraction**) follows BCD subtraction.
  - both correct the result of addition or subtraction so it is a BCD number

DAA Instruction

- DAA follows the ADD or ADC instruction to adjust the result into a BCD result.
- After adding the BL and DL registers, the result is adjusted with a DAA instruction before being stored in CL.

DAS Instruction

- Functions as does DAA instruction, except it follows a subtraction instead of an addition.

ASCII Arithmetic

- ASCII arithmetic instructions function with coded numbers, value 30H to 39H for 0–9.
- Four instructions in ASCII arithmetic operations:
  - AAA (**ASCII adjust after addition**)
  - AAD (**ASCII adjust before division**)
  - AAM (**ASCII adjust after multiplication**)
  - AAS (**ASCII adjust after subtraction**)
- These instructions use register AX as the source and as the destination.

AAA Instruction

- Addition of two one-digit ASCII-coded numbers will not result in any useful data.

AAD Instruction

- Appears before a division.
- The AAD instruction requires the AX registe contain a two-digit unpacked BCD number (not ASCII) before executing.

AAM Instruction

- Follows multiplication instruction after multiplying two one-digit unpacked BCD numbers.
- AAM converts from binary to unpacked BCD.
- If a binary number between 0000H and 0063H appears in AX, AAM converts it to BCD.

AAS Instruction

- AAS adjusts the AX register after an ASCII subtraction.

5-4 BASIC LOGIC INSTRUCTIONS

- Include AND, OR, Exclusive-OR, and NOT.
  - also TEST, a special form of the AND instruction
  - NEG, similar to the NOT instruction
- Logic operations provide binary bit control in low-level software.
  - allow bits to be set, cleared, or complemented
- Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system.

- All logic instructions affect the flag bits.
- Logic operations always clear the carry and overflow flags
  - other flags change to reflect the result
- When binary data are manipulated in a register or a memory location, the rightmost bit position is always numbered bit 0.
  - position numbers increase from bit 0 to the left, to bit 7 for a byte, and to bit 15 for a word
  - a doubleword (32 bits) uses bit position 31 as its leftmost bit and a quadword (64-bits) position 63

AND

- Performs logical multiplication, illustrated by a truth table.
- AND can replace discrete AND gates if the speed required is not too grea
  - normally reserved for embedded control applications
- In 8086, the AND instruction often executes
  in about a microsecond.
  - with newer versions, the execution speed is greatly increased

Figure 5–3 (a) The truth table for the AND operation and (b) the logic symbol of an AND gate.

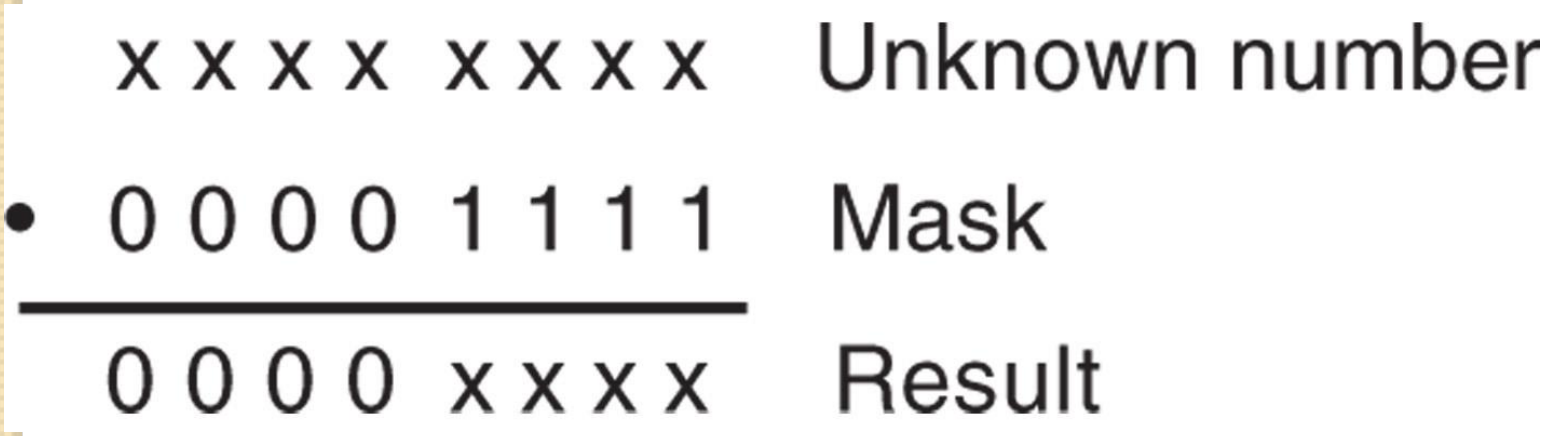
A	B	T
0	0	0
0	1	0
1	0	0
1	1	1

(a)

(b)

- AND clears bits of a binary number.
  - called **masking**
- AND uses any mode except memory-to-memory and segment register addressing.
- An ASCIInumber can be converted to BCD by using AND to mask off the leftmost four binary bit positions.

Figure 5–4 The operation of the AND function showing how bits of a number are cleared to zero.



OR

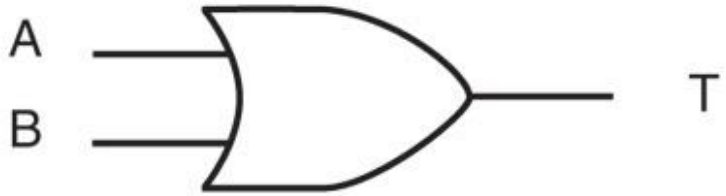
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Performs logical addition
  - often called the *Inclusive-OR* function
- The OR function generates a logic 1 output if any inputs are 1.
  - a 0 appears at output only when all inputs are 0
- Figure 5–6 shows how the OR gate sets (1) any bit of a binary number.
- The OR instruction uses any addressing mode except segment register addressing.

Figure 5–5 (a) The truth table for the OR operation and (b) the logic symbol of an OR gate.

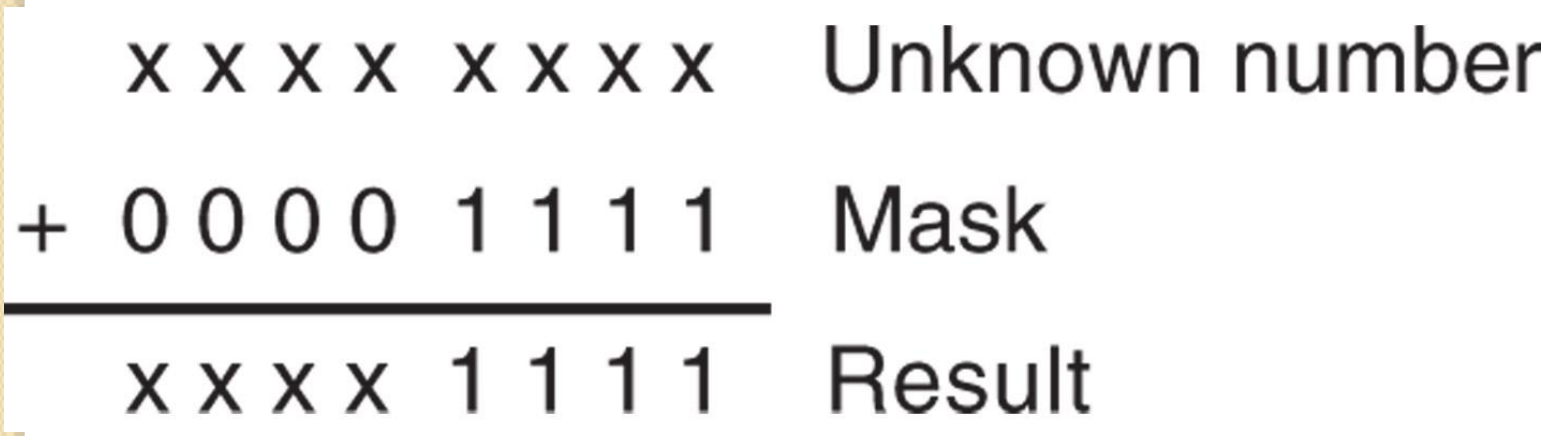
A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

(a)



(b)

Figure 5–6 The operation of the OR function showing how bits of a number are set to one.



Exclusive-OR

- Differs from Inclusive-OR (OR) in that the 1,1 condition of Exclusive-OR produces a 0.
  - a 1,1 condition of the OR function produces a 1
- The Exclusive-OR operation *excludes* this condition; the Inclusive-OR *includes* it.
- If inputs of the Exclusive-OR function are both 0 or both 1, the output is 0; if the inputs are different, the output is 1.
- Exclusive-OR is sometimes called a comparator.

Figure 5–7 (a) The truth table for the Exclusive-OR operation and (b) the logic symbol of an Exclusive-OR gate.

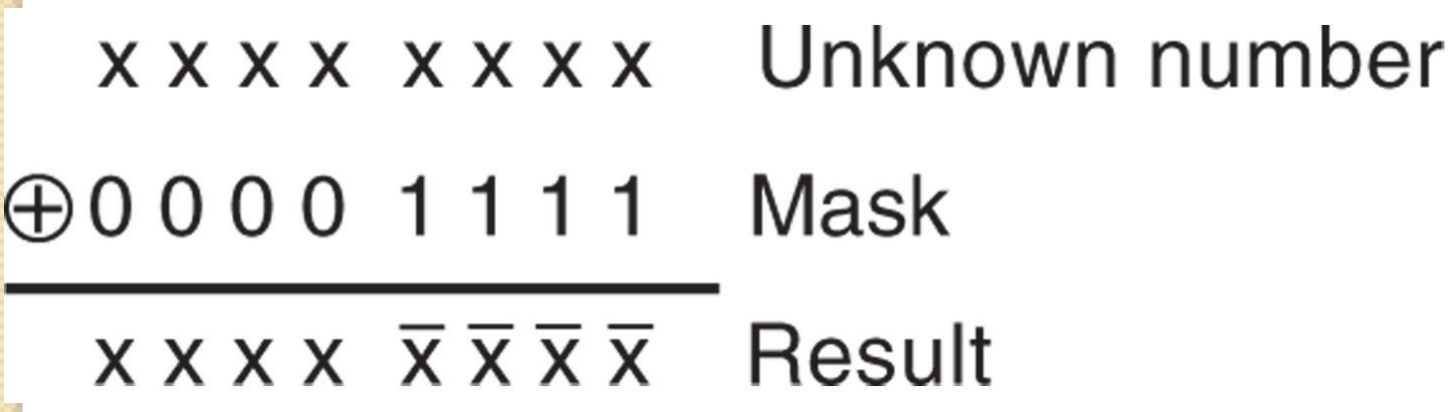
A	B	T
0	0	0
0	1	1
1	0	1
1	1	0

(a)

(b)

- XOR uses any addressing mode except segment register addressing.
- Exclusive-OR is useful if some bits of a register or memory location must be inverted.
- Figure 5–8 shows how just part of an unknown quantity can be inverted by XOR.
  - when a 1 Exclusive-ORs with X, the result is X
  - if a 0 Exclusive-ORs with X, the result is X
- A common use for the Exclusive-OR instruction is to clear a register to zero

Figure 5–8 The operation of the Exclusive-OR function showing how bits of a number are inverted.



Test and Bit Test Instructions

- **TEST** performs the AND operation.
  - only affects the condition of the flag register, which indicates the result of the test
  - functions the same manner as a CMP
- Usually the followed by either the JZ (jump if zero) or JNZ (jump if not zero) instruction.
- The destination operand is normally tested against immediate data.

- 80386 - Pentium 4 contain additional test instructions that test single bit positions.
  - four different bit test instructions available
- All forms test the bit position in the destination operand selected by the source operand.

NOT and NEG

- NOT and NEG can use any addressing mode except segment register addressing.
- The NOT instruction inverts all bits of a byte, word, or doubleword.
- NEG two's complements a number.
  - the arithmetic sign of a signed number changes from positive to negative or negative to positive
- The NOT function is considered logical, NEG function is considered an arithmetic operation.

Shift and Rotate

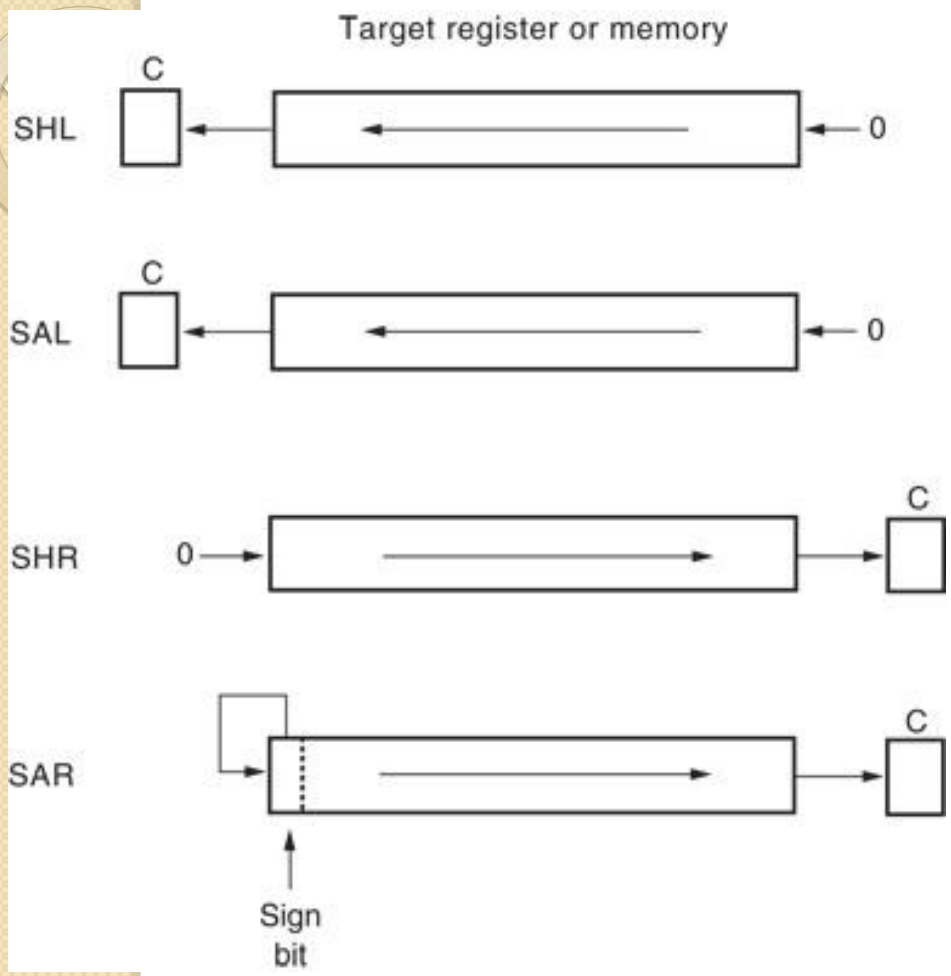
SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Shift and rotate instructions manipulate binary numbers at the binary bit level.
  - as did AND, OR, Exclusive-OR, and NOT
- Common applications in low-level software used to control I/O devices.
- The microprocessor contains a complete complement of shift and rotate instructions that are used to shift or rotate any memory data or register.

Shift

- Position or move numbers to the left or right within a register or memory location.
  - also perform simple arithmetic as multiplication by powers of 2^{+n} (left shift) and division by powers of 2^{-n} (right shift).
- The microprocessor's instruction set contains four different shift instructions:
  - two are logical; two are arithmetic shifts
- All four shift operations appear in Figure 5–9.

Figure 5–9 The shift instructions showing the operation and direction of the shift.



- logical shifts move 0 in the rightmost bit for a logical left shift;
- 0 to the leftmost bit position for a logical right shift
- arithmetic right shift copies the sign-bit through the number
- logical right shift copies a 0 through the number.

- Logical shifts multiply or divide unsigned data; arithmetic shifts multiply or divide signed data.
  - a shift left always multiplies by 2 for each bit position shifted
  - a shift right always divides by 2 for each position
  - shifting a two places, multiplies or divides by 4

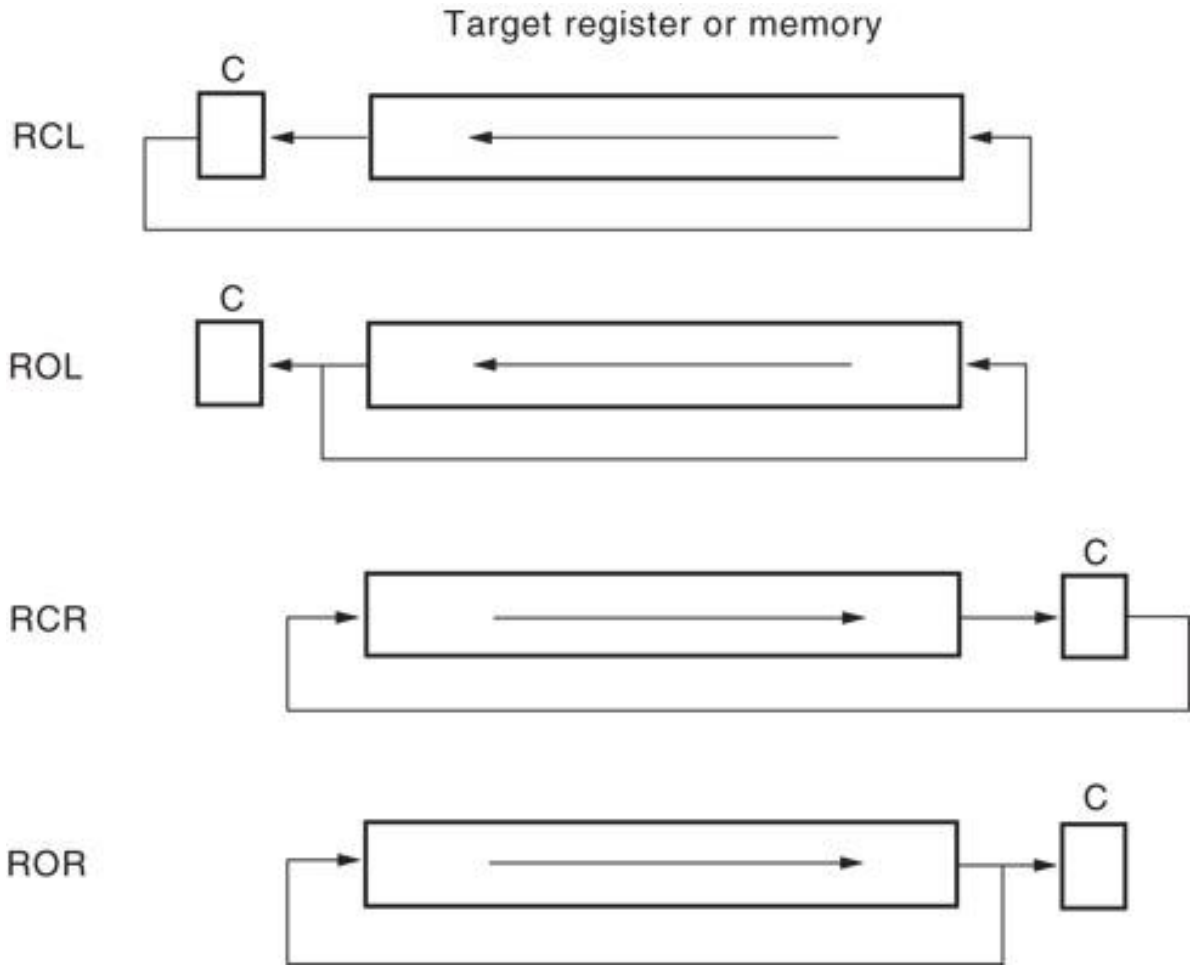
Double-Precision Shifts (80386–Core2 Only)

- 80386 and above contain two double precision shifts: SHLD (shift left) and SHRD (shift right).
- Each instruction contains three operands.
- Both function with two 16-or 32-bit registers,
  - or with one 16- or 32-bit memory location and a register

Rotate

- Positions binary data by rotating information in a register or memory location, either from one end to another or through the carry flag.
  - used to shift/position numbers wider than 16 bits
- With either type of instruction, the programmer can select either a left or a right rotate.
- Addressing modes used with rotate are the same as those used with shifts.
- Rotate instructions appear in Figure 5–10.

Figure 5–10 The rotate instructions showing the direction and operation of each rotate.



- A rotate count can be immediate or located in register CL.
  - if CL is used for a rotate count, it does not change
- Rotate instructions are often used to shift wide numbers to the left or right.

Bit Scan Instructions

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Scan through a number searching for a 1-bit.
  - accomplished by shifting the number
  - available in 80386–Pentium 4
- BSF scans the number from the leftmost bit toward the right; BSR scans the number from the rightmost bit toward the left.
  - if a 1-bit is encountered, the zero flag is set and the bit position number of the 1-bit is placed into the destination operand
  - if no 1-bit is encountered the zero flag is cleared

5-6 STRING COMPARISONS

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- String instructions are powerful because they allow the programmer to manipulate large blocks of data with relative ease.
- Block data manipulation occurs with MOVS, LODS, STOS, INS, and OUTS.
- Additional string instructions allow a section of memory to be tested against a constant or against another section of memory.
  - SCAS (**string scan**); CMPS (**string compare**)

SCAS

- Compares the AL register with a byte block of memory, AX with a word block, or EAX with a doubleword block of memory.
- Opcode used for byte comparison is SCASB; for word comparison SCASW; doubleword comparison is SCASD
- SCAS uses direction flag (D) to select auto-increment or auto-decrement operation for DI.
  - also repeat if prefixed by conditional repeat prefix

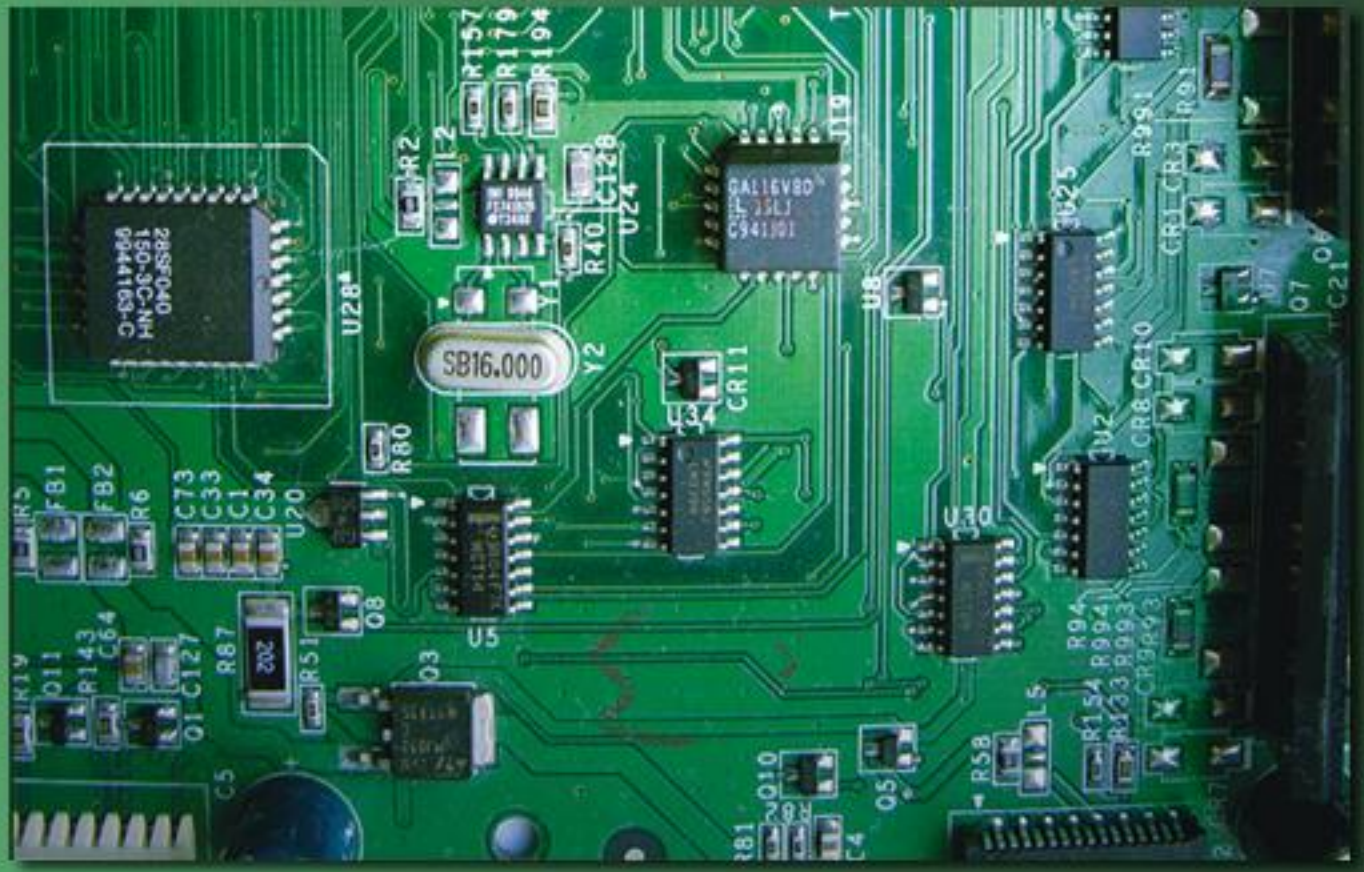
CMPS

- Always compares two sections of memory data as bytes (CMPSB), words (CMPSW), or doublewords (CMPSD).
  - contents of the data segment memory location addressed by SI are compared with contents of extra segment memory addressed by DI
  - CMPS instruction increments/decrements SI & DI
- Normally used with REPE or REPNE prefix.
  - alternates are REPZ (repeat while zero) and REPNZ (repeat while not zero)

The Intel Microprocessors

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

Architecture, Programming, and Interfacing



EIGHTH EDITION

Barry B. Brey

PEARSON

Chapter 6: Program Control Instructions

Introduction

- This chapter explains the program control instructions, including the jumps, calls, returns, interrupts, and machine control instructions.
- This chapter also presents the relational assembly language statements (.IF, .ELSE, .ELSEIF, .ENDIF, .WHILE, .ENDW, .REPEAT, and .UNTIL) that are available in version 6.xx and above of MASM or TASM, with version 5.xx set for MASM compatibility.

6–1 THE JUMP GROUP

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Allows programmer to skip program sections and branch to any part of memory for the next instruction.
- A conditional jump instruction allows decisions based upon numerical tests.
  - results are held in the flag bits, then tested by conditional jump instructions
- LOOP and conditional LOOP are also forms of the jump instruction.

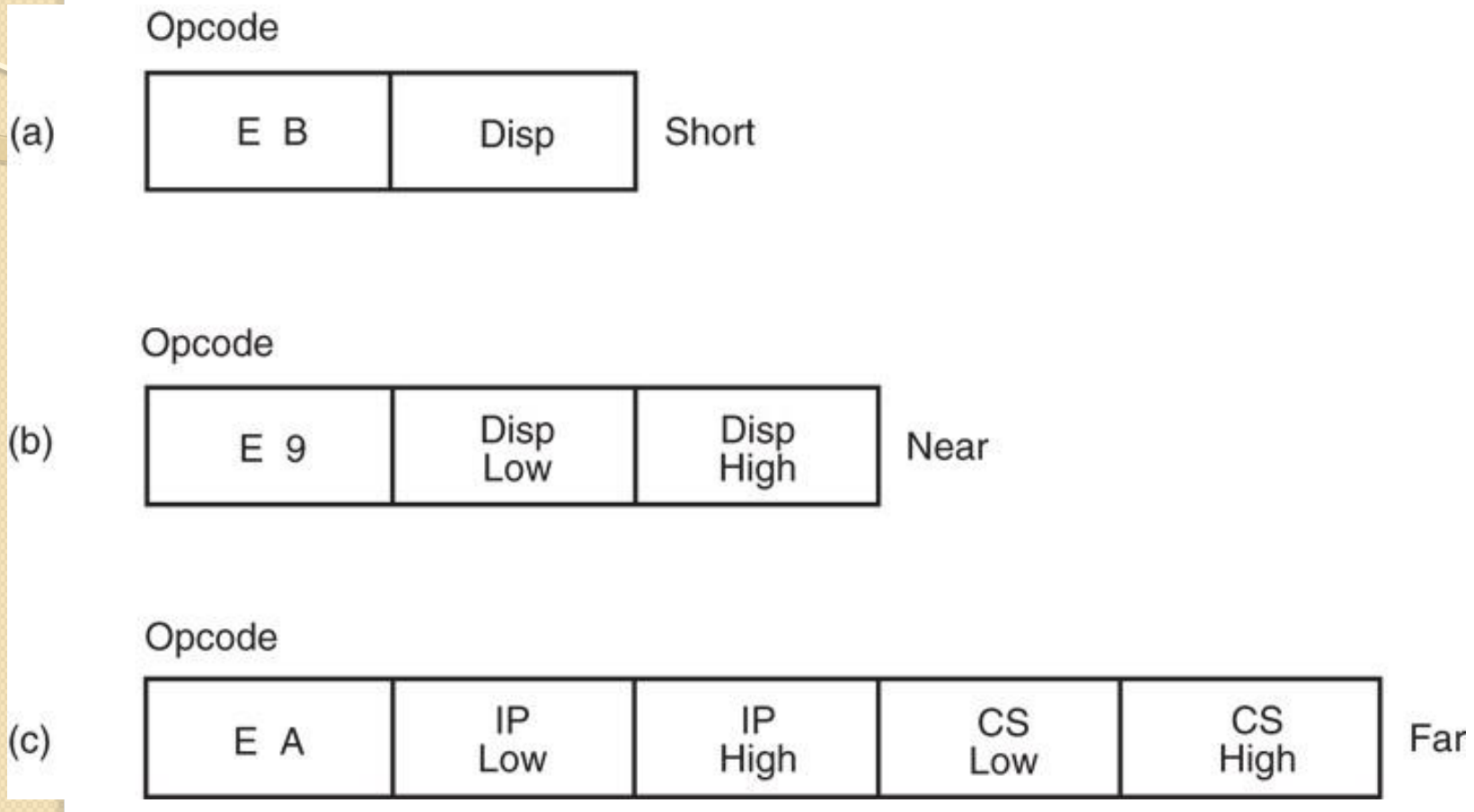
Unconditional Jump (JMP)

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Three types: short jump, near jump, far jump.
- **Short jump** is a 2-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes.
  - from the address following the jump
- 3-byte **near jump** allows a branch or jump within ±32K bytes from the instruction in the current code segment.

- 5-byte **far jump** allows a jump to any memory location within the real memory system.
- The short and near jumps are often called **intrasegment jumps.**
- Far jumps are called **intersegment jumps**.

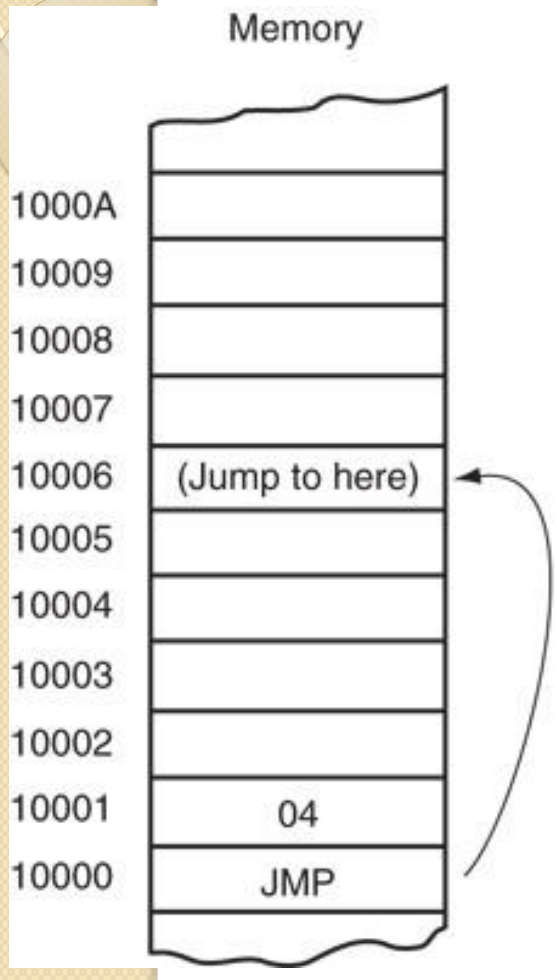
Figure 6–1 The three main forms of the JMP instruction. Note that Disp is either an 8- or 16-bit signed displacement or distance.



Short Jump

- Called **relative jumps** because they can be moved, with related software, to any location in the current code segment without a change.
  - jump address is not stored with the opcode
  - a **distance**, or displacement, follows the opcode
- The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and –128.
- Short jump instruction appears in Figure 6–2.

Figure 6–2 A short jump to four memory locations beyond the address of the next instruction.



CS = 1000H
IP = 0002H
New IP = IP + 4
New IP = 0006H

- when the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment
- The instruction branches to this new address for the next instruction in the program

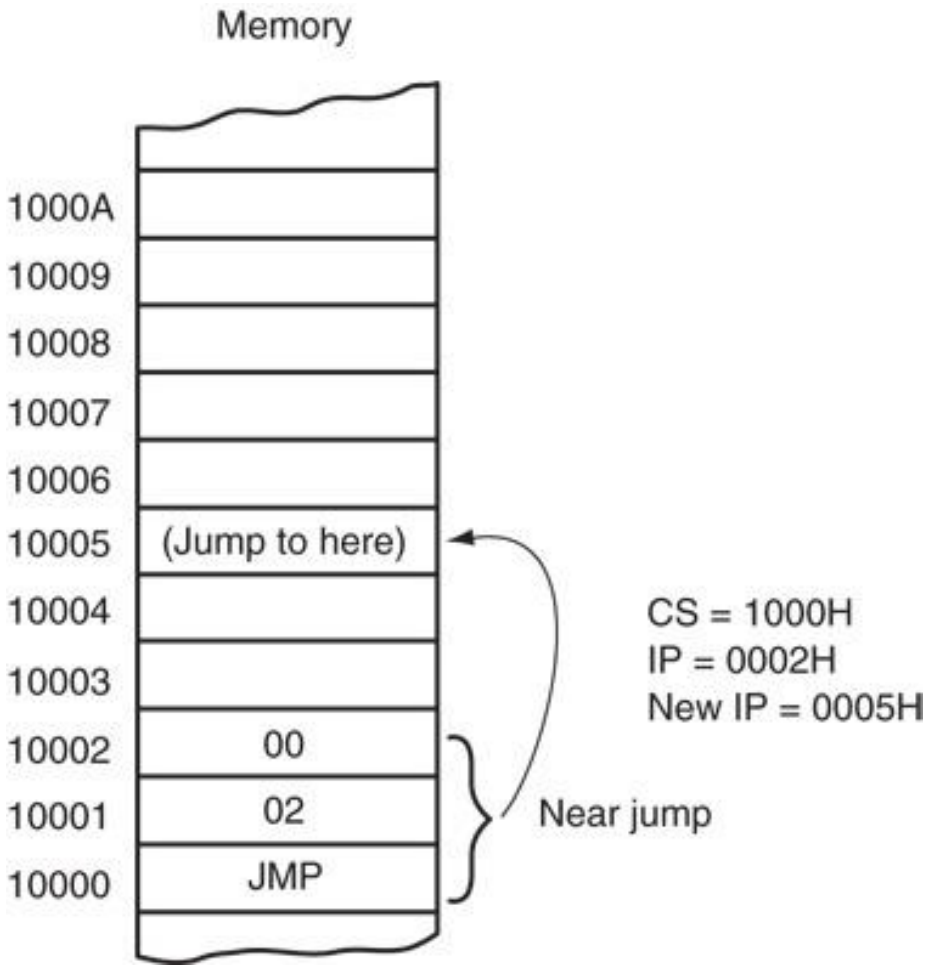
- When a jump references an address, a label normally identifies the address.
- The JMP NEXT instruction is an example.
  - it jumps to label NEXT for the next instruction
  - very rare to use an actual hexadecimal address with any jump instruction
- The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it
  - if a colon does not follow, you cannot jump to it
- The only time a colon is used is when the label is used with a jump or call instruction.

Near Jump

- A near jump passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction.
  - distance is ±2G in 80386 and above when operated in protected mode
- Near jump is a 3-byte instruction with opcode followed by a signed 16-bit displacement.
  - 80386 - Pentium 4 displacement is 32 bits and the near jump is 5 bytes long

- Signed displacement adds to the instruction pointer (IP) to generate the jump address.
  - because signed displacement is ±32K, a near jump can jump to any memory location within the current real mode code segment
- The protected mode code segment in the 80386 and above can be 4G bytes long.
  - 32-bit displacement allows a near jump to any location within ±2G bytes
- Figure 6–3 illustrates the operation of the real mode near jump instruction.

Figure 6–3 A near jump that adds the displacement (0002H) to the contents of IP.



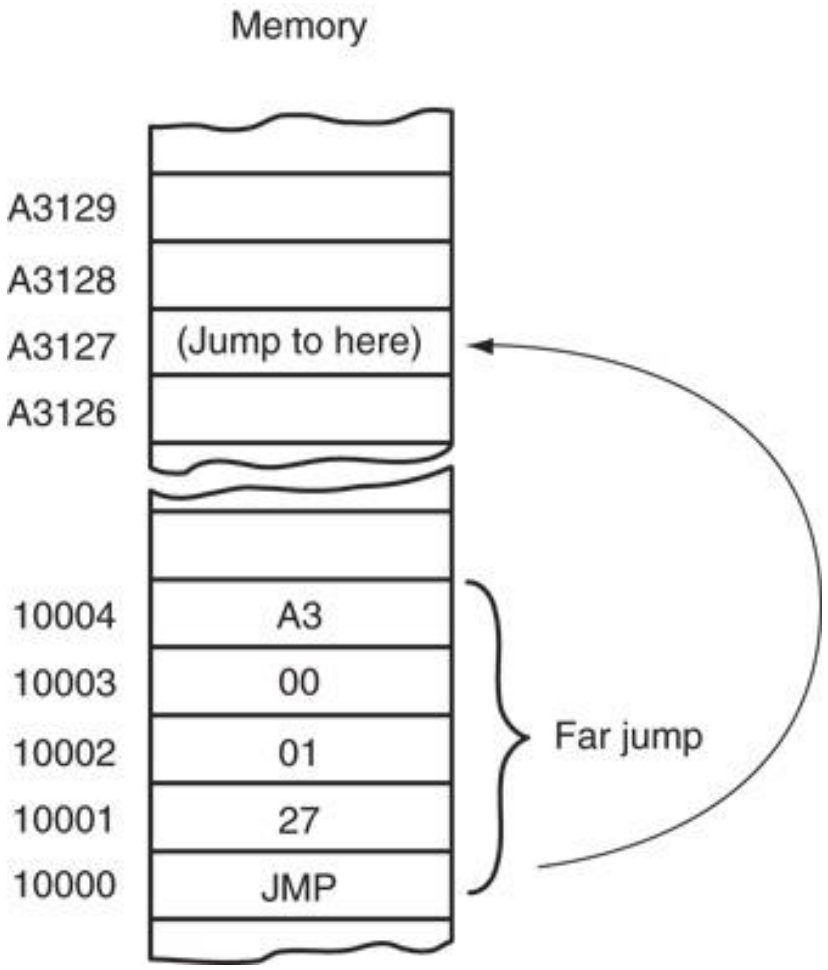
- The near jump is also relocatable because it is also a relative jump.
- This feature, along with the relocatable data segments, Intel microprocessors ideal for use in a general-purpose computer system.
- Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

Far Jump

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Obtains a new segment and offset address to accomplish the jump:
  - bytes 2 and 3 of this 5-byte instruction contain the new offset address
  - bytes 4 and 5 contain the new segment address
  - in protected mode, the segment address accesses a descriptor with the base address of the far jump segment
  - offset address, either 16 or 32 bits, contains the offset address within the new code segment

Figure 6–4 A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.



- The far jump instruction sometimes appears with the FAR PTR directive.
  - another way to obtain a far jump is to define a label as a **far label**
  - a label is far only if it is external to the current code segment or procedure
- The JMP UP instruction references a far label.
  - label UP is defined as a far label by the EXTRN UP:FAR directive
- **External labels** appear in programs that contain more than one program file.

- Another way of defining a label as global is to use a *double colon* (LABEL::)
  - required inside procedure blocks defined as near if the label is accessed from outside the procedure block
- When the program files are joined, the linker inserts the address for the UP label into the JMP UP instruction.
- Also inserts segment address in JMP START instruction.

Jumps with Register Operands

- Jump can also use a 16- or 32-bit register as an operand.
  - automatically sets up as an **indirect jump**
  - address of the jump is in the register specified by the jump instruction
- Unlike displacement associated with the near jump, register contents are transferred directly into the instruction pointer.
- An indirect jump does not add to the instruction pointer.

- JMP AX, for example, copies the contents of the AX register into the IP.
  - allows a jump to any location within the current code segment
- In 80386 and above, JMP EAX also jumps to any location within the current code segment;
  - in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed

Indirect Jumps Using an Index

- Jump instruction may also use the [ ] form of addressing to directly access the jump table.
- The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps.
  - also known as a *double-indirect jump* if the register jump is called an *indirect jump*
- The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction.

- Mechanism used to access the jump table is identical with a normal memory reference.
  - JMP TABLE [SI] instruction points to a jump address stored at the code segment offset location addressed by SI
- Both the register and indirect indexed jump instructions usually address a 16-bit offset.
  - both types of jumps are near jumps
- If JMP FAR PTR [SI] or JMP TABLE [SI], with TABLE data defined with the DD directive:
  - microprocessor assumes the jump table contains doubleword, 32-bit addresses (IP and CS)

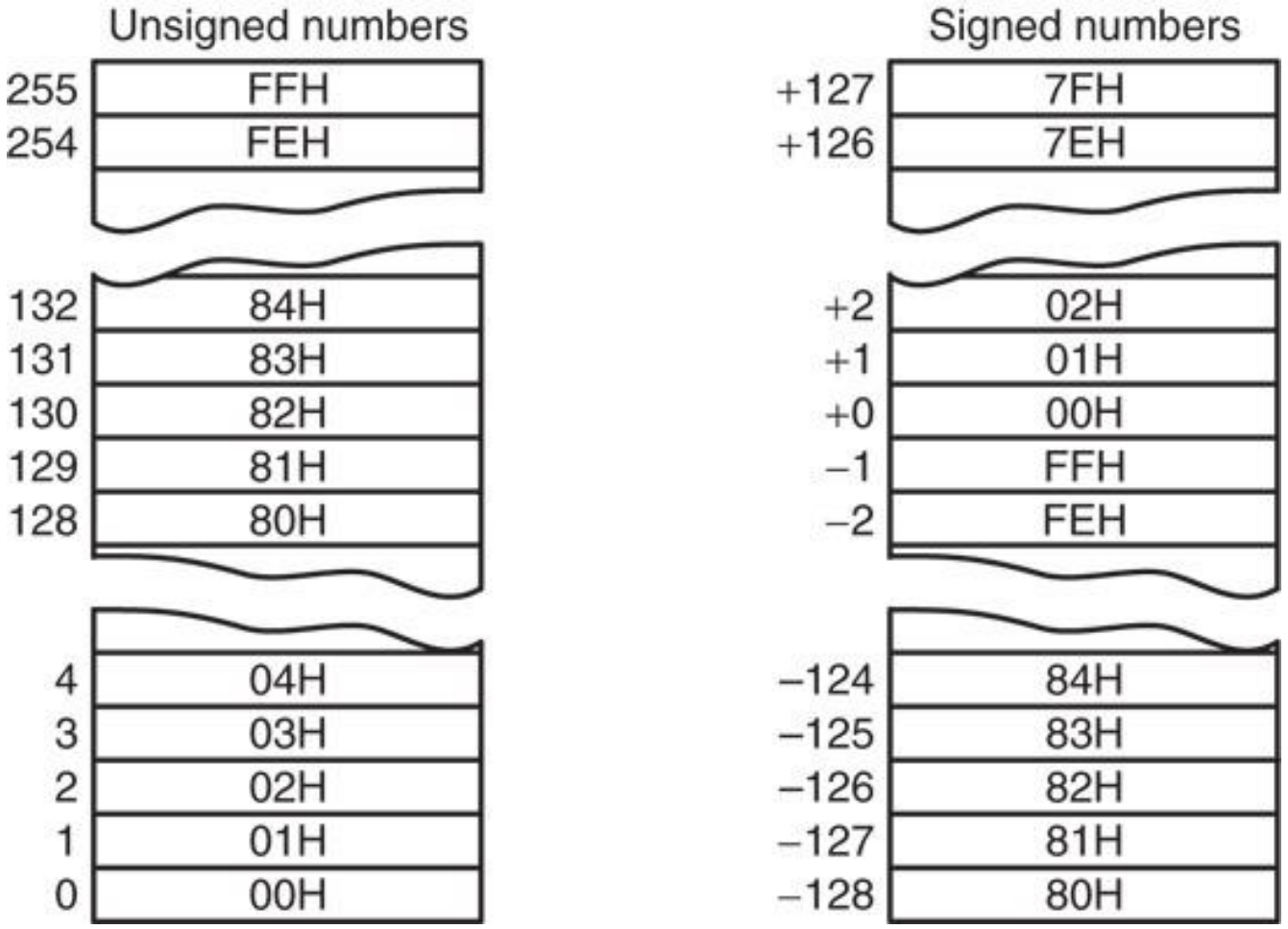
Conditional Jumps and Conditional Sets

- Always short jumps in 8086 - 80286.
  - limits range to within +127 and –128 bytes from the location following the conditional jump
- In 80386 and above, conditional jumps are either short or near jumps (±32K).
  - in 64-bit mode of the Pentium 4, the near jump distance is ±2G for the conditional jumps
- Allows a conditional jump to any location within the current code segment.

- Conditional jump instructions test flag bits:
  - sign (S), zero (Z), carry (C)
  - parity (P), overflow (0)
- If the condition under test is true, a branch to the label associated with the jump instruction occurs.
  - if false, next sequential step in program executes
  - for example, a JC will jump if the carry bit is set
- Most conditional jump instructions are straightforward as they often test one flag bit.
  - although some test more than one

- Because both signed and unsigned numbers are used in programming.
- Because the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons.
- 16- and 32-bit numbers follow the same order as 8-bit numbers, except that they are larger.
- Figure 6–5 shows the order of both signed and unsigned 8-bit numbers.

Figure 6–5 Signed and unsigned numbers follow different orders.



- When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.
  - terms *greater than* and *less than* refer to signed numbers
- When unsigned numbers are compared, use the JA, JB, JAB, JBE, JE, and JNE instructions.
  - terms *above* and *below* refer to unsigned numbers
- Remaining conditional jumps test individual flag bits, such as overflow and parity.

- Remaining conditional jumps test individual flag bits, such as overflow and parity.
  - notice that JE has an alternative opcode JZ
- All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test.

The Conditional Set Instructions

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- 80386 - Core2 processors also contain conditional set instructions.
  - conditions tested by conditional jumps put to work with the conditional set instructions
  - conditional set instructions set a byte to either 01H or clear a byte to 00H, depending on the outcome of the condition under test
- Useful where a condition must be tested at a point much later in the program.

LOOP

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- A combination of a decrement CX and the JNZ conditional jump.
- In 8086 - 80286 LOOP decrements CX.
  - if CX != 0, it jumps to the address indicated by the label
  - If CX becomes 0, the next sequential instruction executes
- In 80386 and above, LOOP decrements either CX or ECX, depending upon instruction mode.

- In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
  - default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions 80386 - Core2
- In 64-bit mode, the loop counter is in RCX.
  - and is 64 bits wide
- There is no direct move from segment register to segment register instruction.

Conditional LOOPs

- LOOP instruction also has conditional forms:
  LOOPE and LOOPNE
- LOOPE (**loop while equal**) instruction jumps if CX != 0 while an equal condition exists.
  - will exit loop if the condition is not equal or the CX register decrements to 0
- LOOPNE (**loop while not equal**) jumps if CX != 0 while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0

- In 80386 - Core2 processors, conditional LOOP can use CX or ECX as the counter.
  - LOOPEW/LOOPED or LOOPNEW/LOOPNED override the instruction mode if needed
- Under 64-bit operation, the loop counter uses RCX and is 64 bits in width
- Alternates exist for LOOPE and LOOPNE.
  - LOOPE same as LOOPZ
  - LOOPNE instruction is the same as LOOPNZ
- In most programs, only the LOOPE and LOOPNE apply.

6–2 CONTROLLING THE FLOW OF THE PROGRAM

- Easier to use assembly language statements .IF, .ELSE, .ELSEIF, and .ENDIF to control the flow of the program than to use the correct conditional jump statement.
  - these statements always indicate a special assembly language command to MASM
- Control flow assembly language statements beginning with a period available to MASM version 6.xx, and not to earlier versions.

- Other statements developed include .REPEAT–.UNTIL and .WHILE–.ENDW.
  - **the dot commands** do not function using the Visual C++ inline assembler
- Never use uppercase for assembly language commands with the inline assembler.
  - some of them are reserved by C++ and will cause problems

WHILE Loops

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Used with a condition to begin the loop.
  - the .ENDW statement ends the loop
- The .BREAK and .CONTINUE statements are available for use with the while loop.
  - .BREAK is often followed by .IF to select the break condition as in .BREAK .IF AL == 0DH
  - .CONTINUE can be used to allow a DO–.WHILE loop to continue if a certain condition is met
- The .BREAK and .CONTINUE commands function the same manner in C++.

REPEAT-UNTIL Loops

- A series of instructions is repeated until some condition occurs.
- The .REPEAT statement defines the start of the loop.
  - end is defined with the .UNTIL statement, which contains a condition
- An .UNTILCXZ instruction uses the LOOP instruction to check CX for a repeat loop.
  - .UNTILCXZ uses the CX register as a counter to repeat a loop a fixed number of times

6–3 PROCEDURES

- A procedure is a group of instructions that usually performs one task.
  - subroutine, method, or **function** is an important part of any system's architecture
- A procedure is a reusable section of the software stored in memory once, used as often as necessary.
  - saves memory space and makes it easier to develop software

- Disadvantage of procedure is time it takes the computer to link to, and return from it.
  - CALL links to the procedure; the RET (**return**) instruction returns from the procedure
- CALL pushes the address of the instruction following the CALL (**return address**) on the stack.
  - the stack stores the return address when a procedure is called during a program
- RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

- A procedure begins with the PROC directive and ends with the ENDP directive.
  - each directive appears with the procedure name
- PROC is followed by the type of procedure:
  - NEAR or FAR
- In MASM version 6.x, the NEAR or FAR type can be followed by the USES statement.
  - USES allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure

- Procedures that are to be used by all software (global) should be written as far procedures.
- Procedures that are used by a given task (local) are normally defined as near procedures.
- Most procedures are near procedures.

CALL

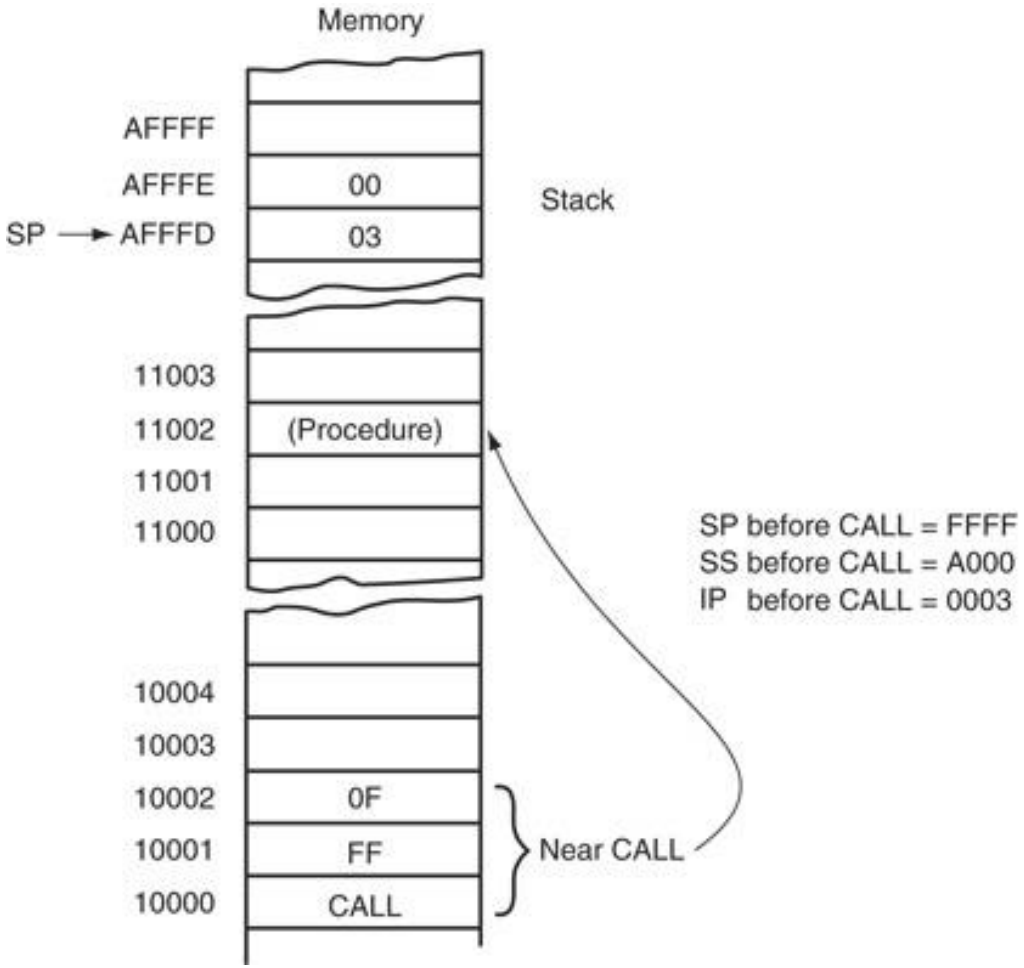
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

Near CALL

- 3 bytes long.
  - the first byte contains the opcode; the second and third bytes contain the displacement
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack.
  - offset address of the next instruction appears in the instruction pointer (IP or EIP)
- It then adds displacement from bytes 2 & 3 to the IP to transfer control to the procedure.

- Why save the IP or EIP on the stack?
  - the instruction pointer always points to the next instruction in the program
- For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
  - program control passes to the instruction following the CALL after a procedure ends
- Figure 6–6 shows the return address (IP) stored on the stack and the call to the procedure.

Figure 6–6 The effect of a near CALL on the stack and the instruction pointer.

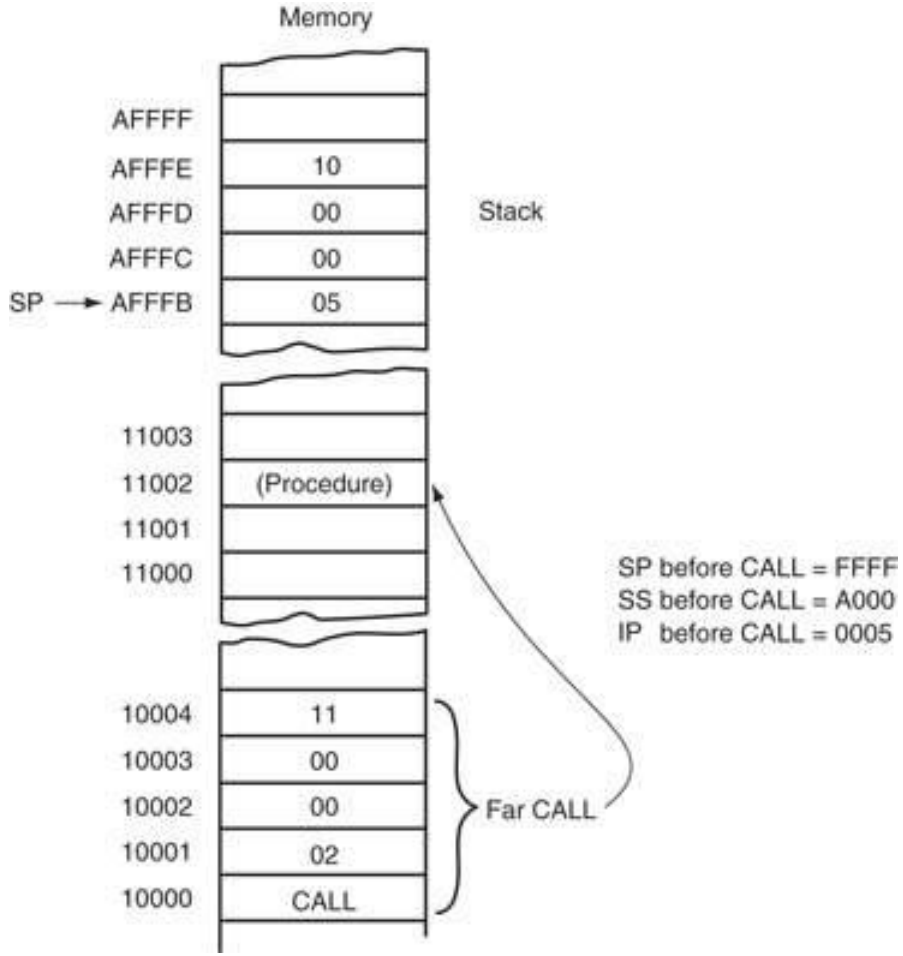


Far CALL

- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
  - bytes 2 and 3 contain new contents of the IP
  - bytes 4 and 5 contain the new contents for CS
- Far CALL places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5.
- This allows far CALL to call a procedure located anywhere in the memory and return from that procedure.

- Figure 6–7 shows how far CALL calls a far procedure.
  - contents of IP and CS are pushed onto the stack
- The program branches to the procedure.
  - A variant of far call exists as CALLF, but should be avoided in favor of defining the type of call instruction with the PROC statement
- In 64-bit mode a far call is to any memory location and information placed onto the stack is an 8-byte number.
  - the far return instruction retrieves an 8-byte return address from the stack and places it into RIP

Figure 6–7 The effect of a far CALL instruction.



CALLs with Register Operands

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- An example CALL BX, which pushes the contents of IP onto the stack.
  - then jumps to the offset address, located in register BX, in the current code segment
- Always uses a 16-bit offset address, stored in any 16-bit register except segment registers.

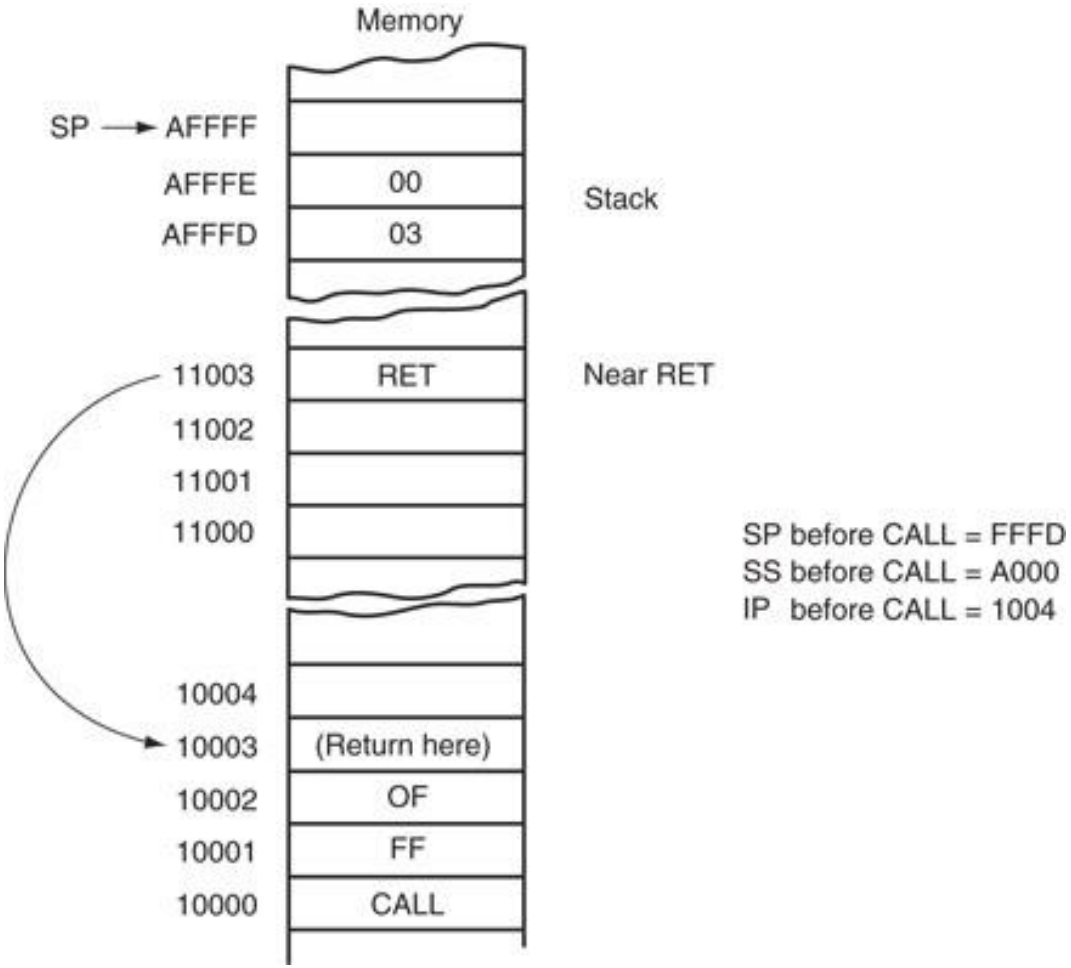
CALLs with Indirect Memory Addresses

- Particularly useful when different subroutines need to be chosen in a program.
  - selection process is often keyed with a number that addresses a CALL address in a lookup table
- Essentially the same as the indirect jump that used a lookup table for a jump address.

RET

- Removes a 16-bit number (**near return**) from the stack placing it in IP, or removes a 32-bit number (**far return**) and places it in IP & CS.
  - near and far return instructions in procedure's PROC directive
  - automatically selects the proper return instruction
- Figure 6–8 shows how the CALL instruction links to a procedure and how RET returns in the 8086–Core2 operating in the real mode.

Figure 6–8 The effect of a near return instruction on the stack and instruction pointer.



- Another form of return adds a number to the contents of the stack pointer (SP) after the return address is removed from the stack.
- A return that uses an immediate operand is ideal for use in a system that uses the C/C++ or PASCAL calling conventions.
  - these conventions push parameters on the stack before calling a procedure
- If the parameters are discarded upon return, the return instruction contains the number of bytes pushed to the stack as parameters.

- Parameters are addressed on the stack by using the BP register, which by default addresses the stack segment.
- Parameter stacking is common in procedures written for C++ or PASCAL by using the C++ or PASCAL calling conventions.
- Variants of the return instruction:
  - RETN and RETF
- Variants should also be avoided in favor of using the PROC statement to define the type of call and return.

6–5 MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

- These instructions provide control of the carry bit, sample the BUSY/TEST pin, and perform various other functions.

Controlling the Carry Flag Bit

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- The carry flag (C) propagates the carry or borrow in multiple-word/doubleword addition and subtraction.
  - can indicate errors in assembly language procedures
- Three instructions control the contents of the carry flag:
  - STC (set carry), CLC (clear carry), and CMC (complement carry)

WAIT

- Monitors the hardware *BUSY* pin on 80286 and 80386, and the *TEST* pin on 8086/8088.
- If the WAIT instruction executes while the *BUSY* pin = 1, nothing happens and the next instruction executes.
  - pin inputs a busy condition when at a logic 0 level
  - if *BUSY* pin = 0 the microprocessor waits for the pin to return to a logic 1

HLT

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Stops the execution of software.
- There are three ways to exit a halt:
  - by interrupt; a hardware reset, or DMA operation
- Often synchronizes external hardware interrupts with the software system.
- DOS and Windows both use interrupts extensively.
  - so HLT will not halt the computer when operated under these operating systems

NOP

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- In early years, before software development tools were available, a NOP, which performs absolutely no operation, was often used to pad software with space for future machine language instructions.
- When the microprocessor encounters a NOP, it takes a short time to execute.

- If you are developing machine language programs, which are extremely rare, it is recommended that you place 10 or so NOPS in your program at 50-byte intervals.
  - in case you need to add instructions at some future point
- A NOP may also find application in time delays to waste time.
- A NOP used for timing is not very accurate because of the cache and pipelines in modern microprocessors.

LOCK Prefix

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Appends an instruction and causes the pin to become a logic 0.
- *LOCK* pin often disables external bus masters or other system components
  - causes pin to activate for duration of instruction
- If more than one sequential instruction islocked, *LOCK* pin remains logic 0 for duration of the sequence of instructions.
- The LOCK:MOV AL,[SI] instruction is an example of a locked instruction.

ESC

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Passes instructions to the floating-point coprocessor from the microprocessor.
- When an ESC executes, the microprocessor provides the memory address, if required, but otherwise performs a NOP.
- Six bits of the ESC instruction provide the opcode to the coprocessor and begin executing a coprocessor instruction.
- ESC is considered obsolete as an opcode.

BOUND

- A comparison instruction that may cause an interrupt (vector type number 5).
- Compares the contents of any 16-bit or 32-bit register against the contents of two words or doublewords of memory
  - an upper and a lower boundary
- If register value compared with memory is not within the boundary, a type 5 interrupt ensues.
- If it is within the boundary, the next instruction in the program executes.

ENTER and LEAVE

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

- Used with stack frames, mechanisms used to pass parameters to a procedure through the stack memory.
- Stack frame also holds local memory variables for the procedure.
- Stack frames provide dynamic areas of memory for procedures in multiuser environments.

- ENTER creates a stack frame by pushing BP onto the stack and then loading BP with the uppermost address of the stack frame.
  - allows stack frame variables to be accessed through the BP register
- ENTER contains two operands:
  - first operand specifies the number of bytes to reserve for variables on the stack frame
  - the second specifies the level of the procedure
- The ENTER and LEAVE instructions were used to call C++ functions in Windows 3.1.