

CSE213

MICROCONTROLLER PROGRAMMING

Addressing Modes

Introduction

- 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.

Chapter Objectives

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

- Explain the operation of each data-addressing mode.
- Use the data-addressing modes to form assembly language statements.
- Explain the operation of each program memory-addressing mode.
- Use the program memory-addressing modes to form assembly and machine language statements.

Chapter Objectives

(*cont.*)

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

- Select the appropriate addressing mode to accomplish a given task.
- Describe sequence of events that place data onto the stack or remove data from the stack.
- Explain how a data structure is placed in memory and used with software.

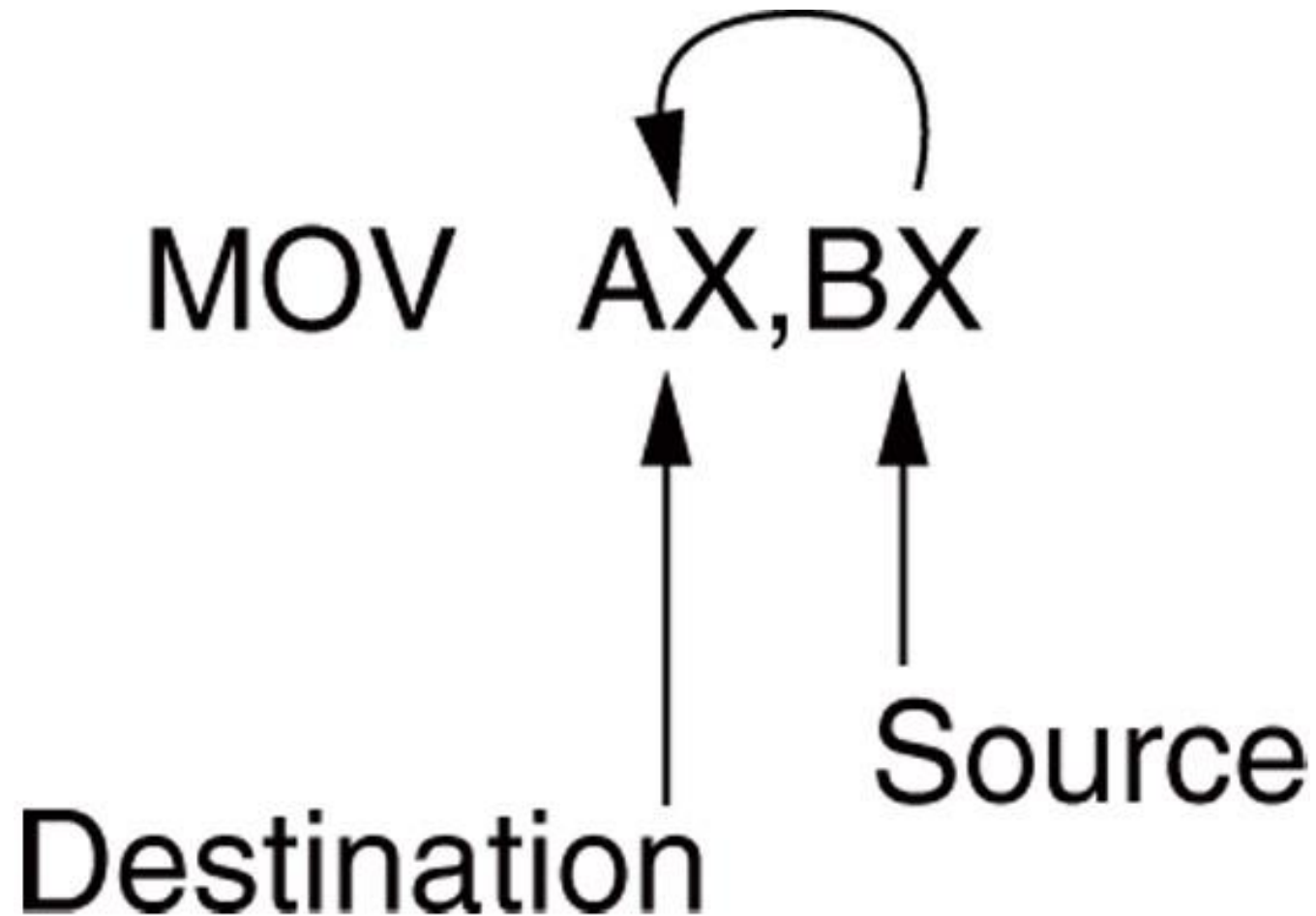
Addressing Mode Types

1. Data Addressing Modes
2. Program Memory Addressing Modes
3. Stack Memory Addressing Modes

3–1 DATA ADDRESSING MODES

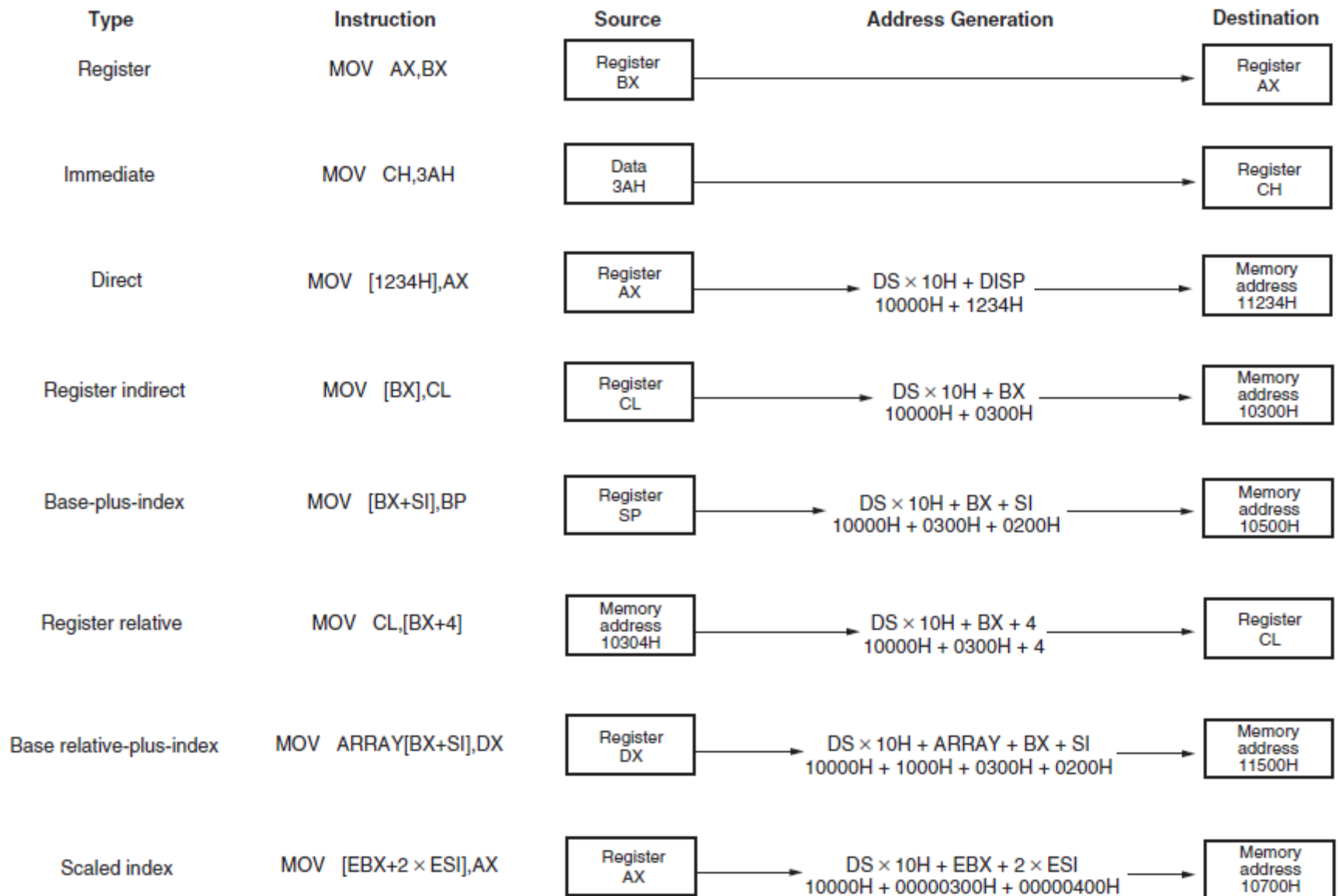
- MOV instruction is a common and flexible instruction.
 - provides a basis for explanation of data-addressing modes
- Figure 3–1 illustrates the MOV instruction and defines the direction of data flow.
- **Source** is to the right and **destination** the left, next to the opcode MOV.
 - an **opcode**,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

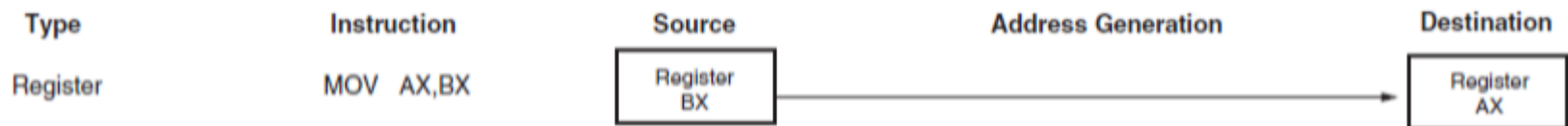
Figure 3–2 8086–Core2 data-addressing modes.



Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

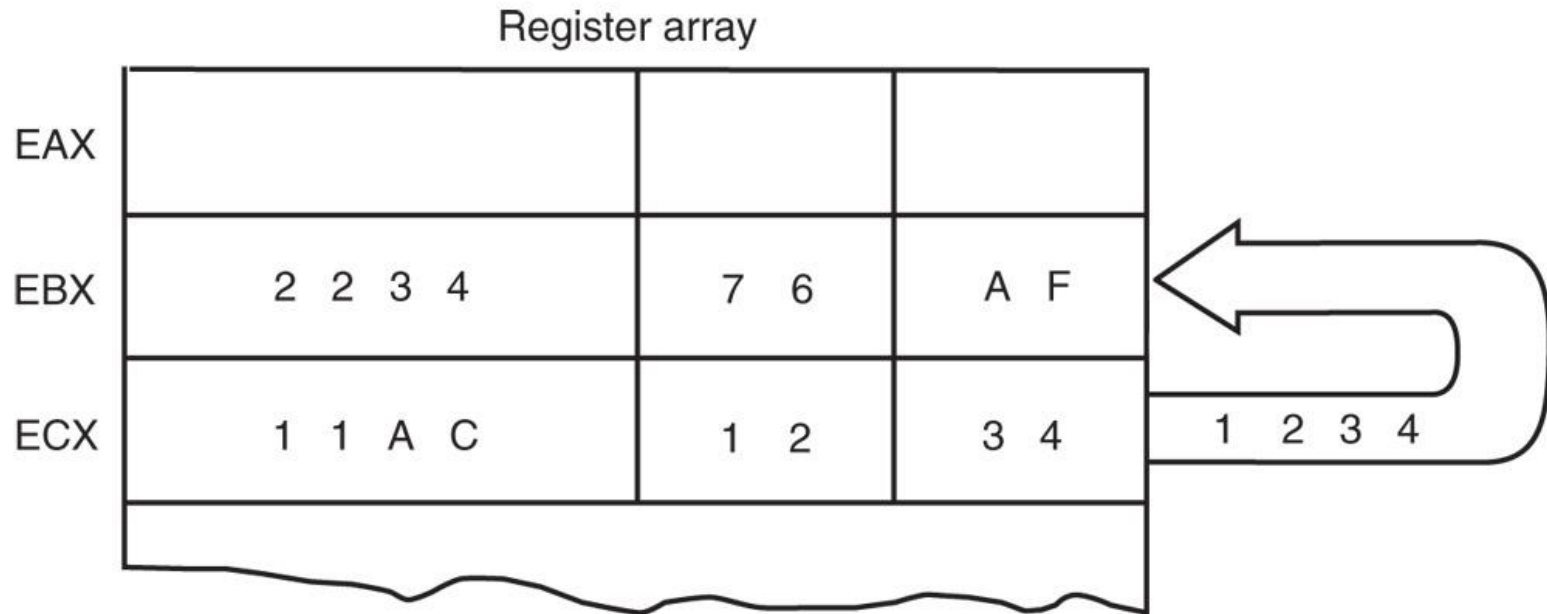
Register Addressing

- 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.



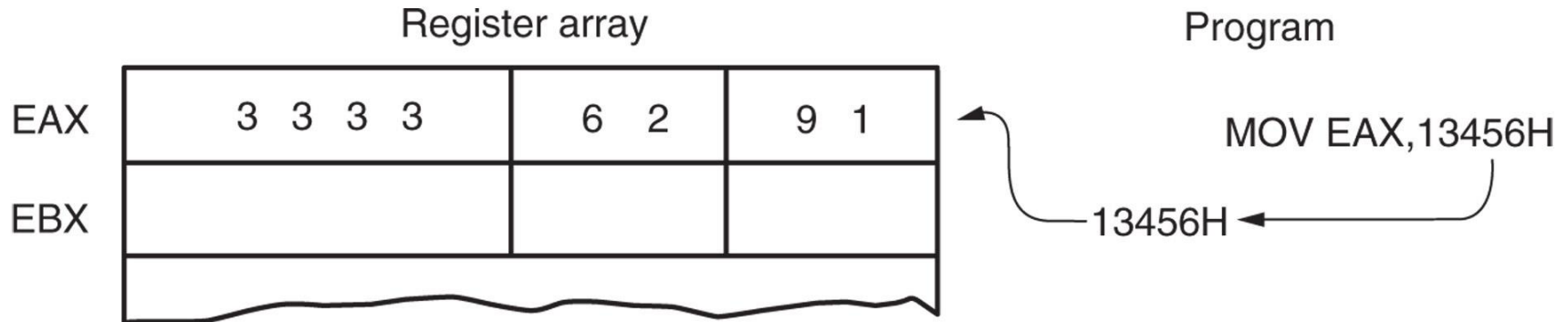
- Figure 3–3 shows the operation of the MOV BX, CX instruction.
- The source register's contents do not change.
 - the destination register's contents do change
- The contents of the destination register or destination memory location change for all instructions except the CMP and TEST instructions.
- The MOV BX, CX instruction does not affect the leftmost 16 bits of register EBX.
 - The MOV BL, CL instruction does not affect the leftmost 8 bits of register BX.

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,13456H 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 # (number or pound sign) 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
 - to represent a decimal 10, 00001010B is used in assembly language

- 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 (;)

Sample Code

```
name "add-sub"

org 100h

mov al, 5          ; bin=00000101b
mov bl, 10         ; hex=0ah or bin=00001010b

; 5 + 10 = 15 (decimal) or hex=0fh or bin=00001111b
add bl, al

; 15 - 1 = 14 (decimal) or hex=0eh or bin=00001110b
sub bl, 1

; print result in binary:
mov cx, 8
print: mov ah, 2    ; print function.
       mov dl, '0'
       test bl, 10000000b ; test first bit.
       jz zero
       mov dl, '1'
zero:  int 21h
       shl bl, 1
loop print

; print binary suffix:
mov dl, 'b'
int 21h

; wait for any key press:
mov ah, 0
int 16h

ret
```

Direct Data Addressing

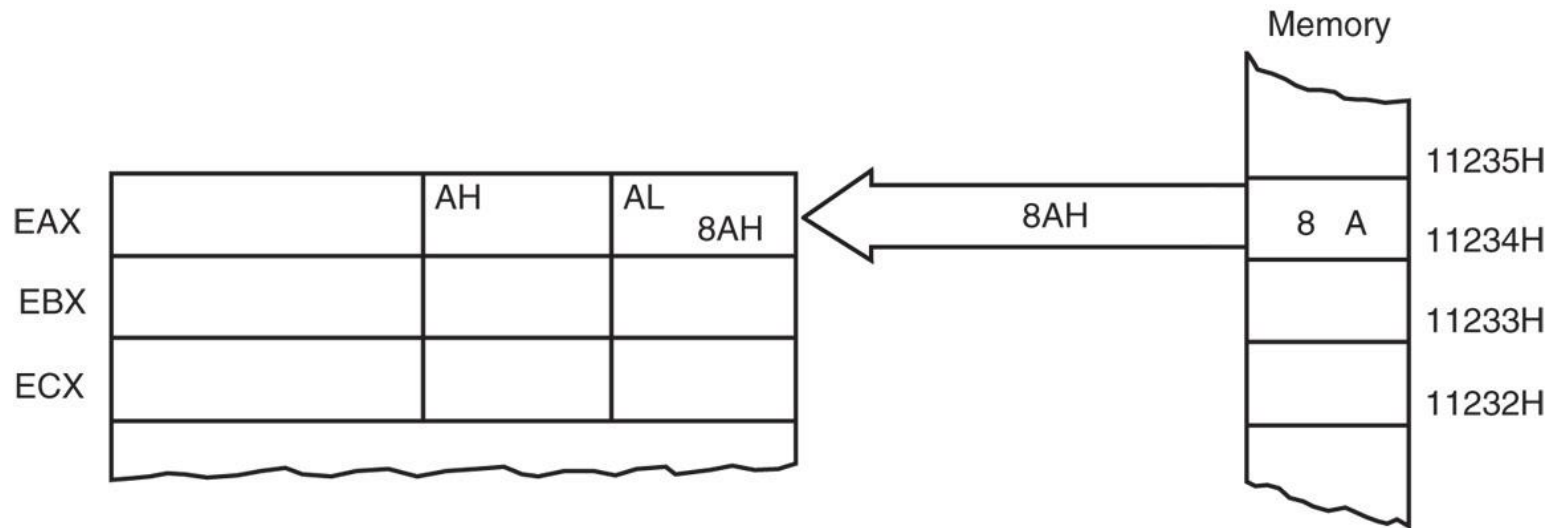
- Applied to many instructions in a typical program.
- Two basic forms of direct data addressing:
 - direct addressing, which applies to a MOV between a memory location and AL, AX, or EAX
 - displacement addressing, which applies to almost any instruction in the instruction set
- Address is formed by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

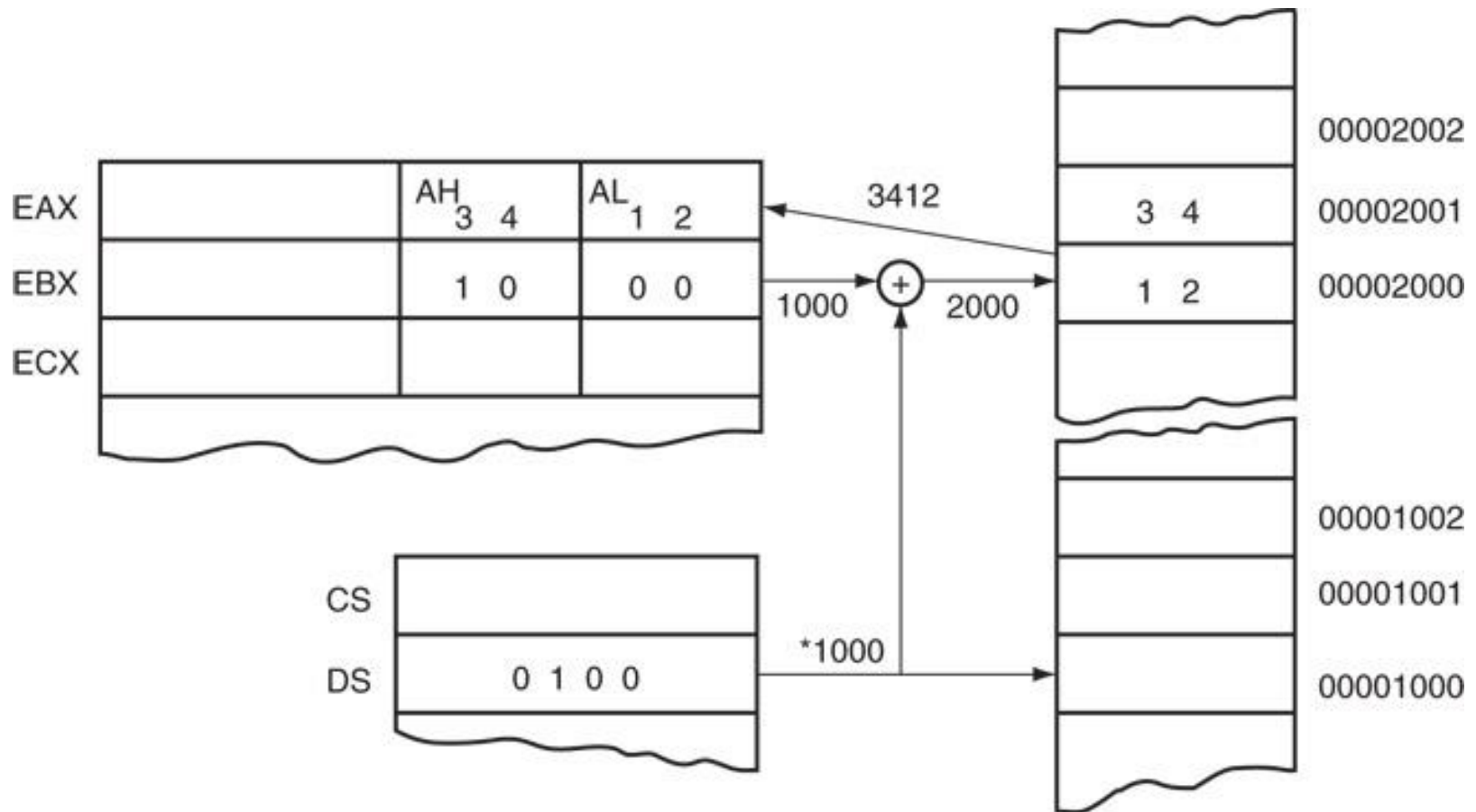
- Almost identical to direct addressing, except the instruction is 4 bytes wide instead of 3.
- In 80386 through Pentium 4, this instruction can be up to 7 bytes wide if a 32-bit register and a 32-bit displacement are specified.
- This type of direct data addressing is much more flexible because most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



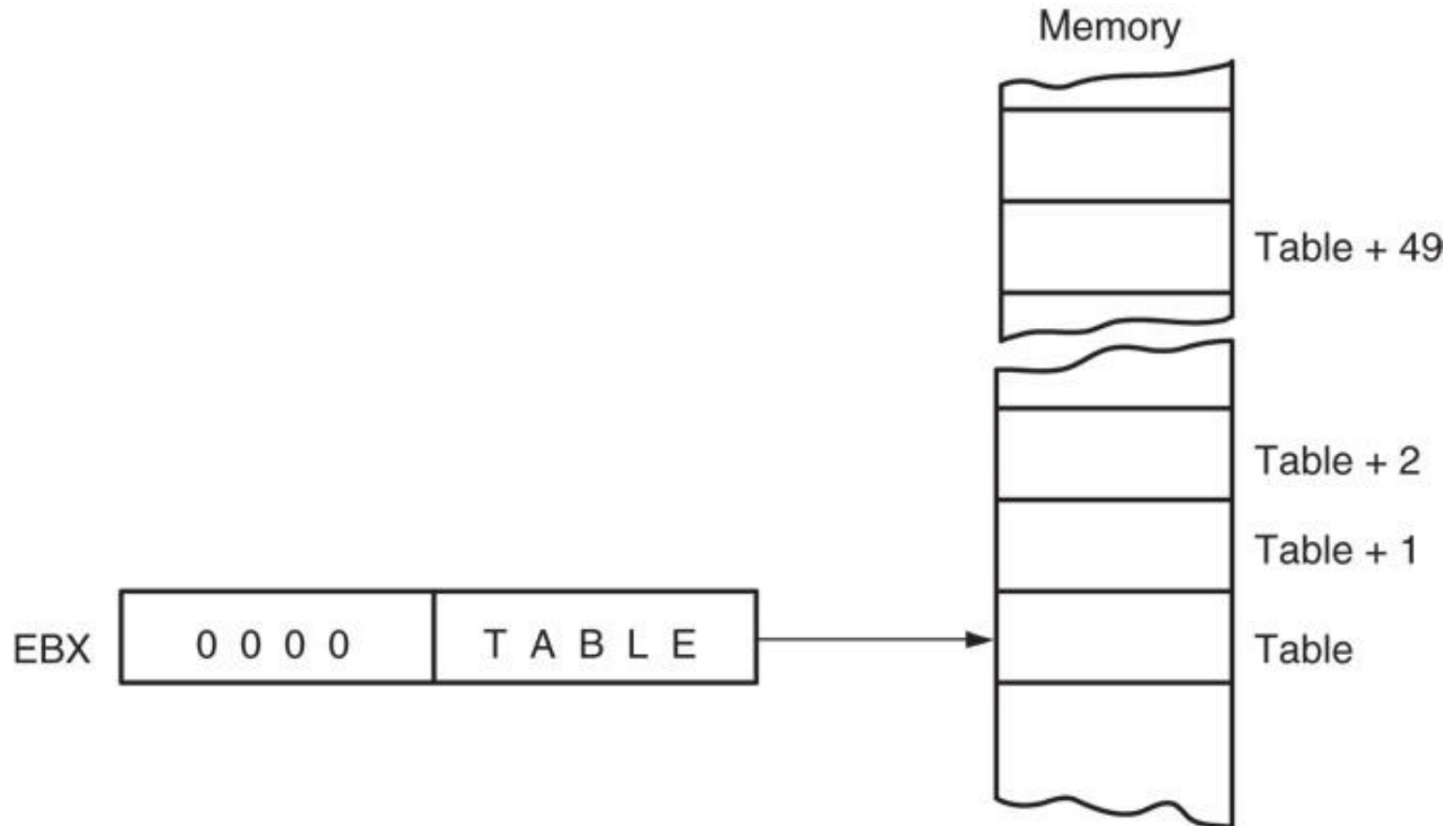
*After DS is appended with a 0.

- 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.

- 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.

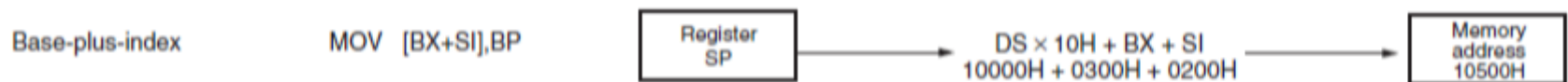
- Indirect addressing often allows a program to refer to tabular data located in memory.
- Figure 3–7 shows the table and the BX register used to sequentially address each location in the table.
- To accomplish this task, load the starting location of the table into the BX register with a MOV immediate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

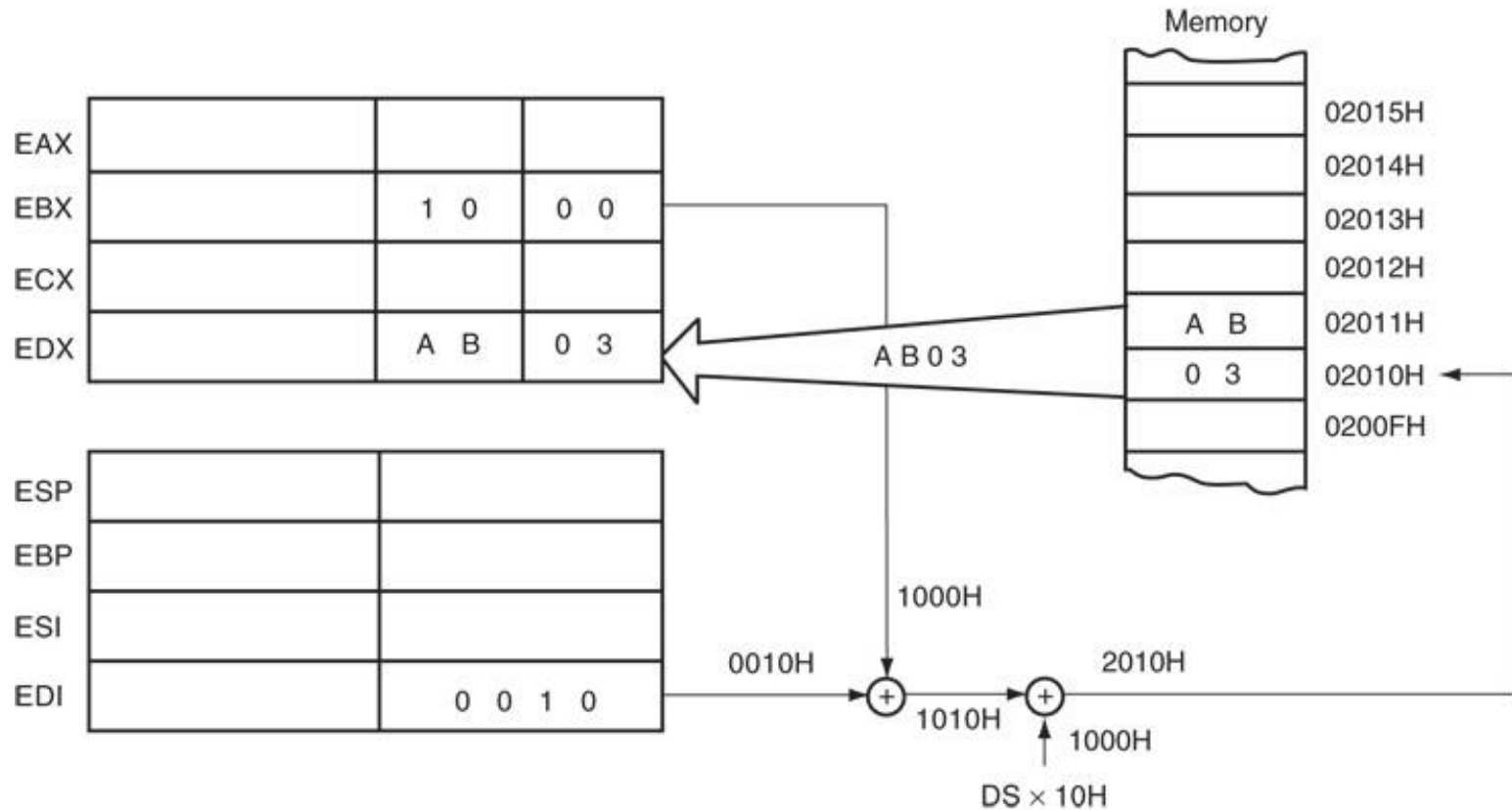
- Similar to indirect addressing because it indirectly addresses memory data.
- The base register often holds the beginning location of a memory array.
 - the index register holds the relative position of an element in the array
 - whenever BP addresses memory data, both the stack segment register and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



Locating Data with Base-Plus-Index Addressing

- 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.

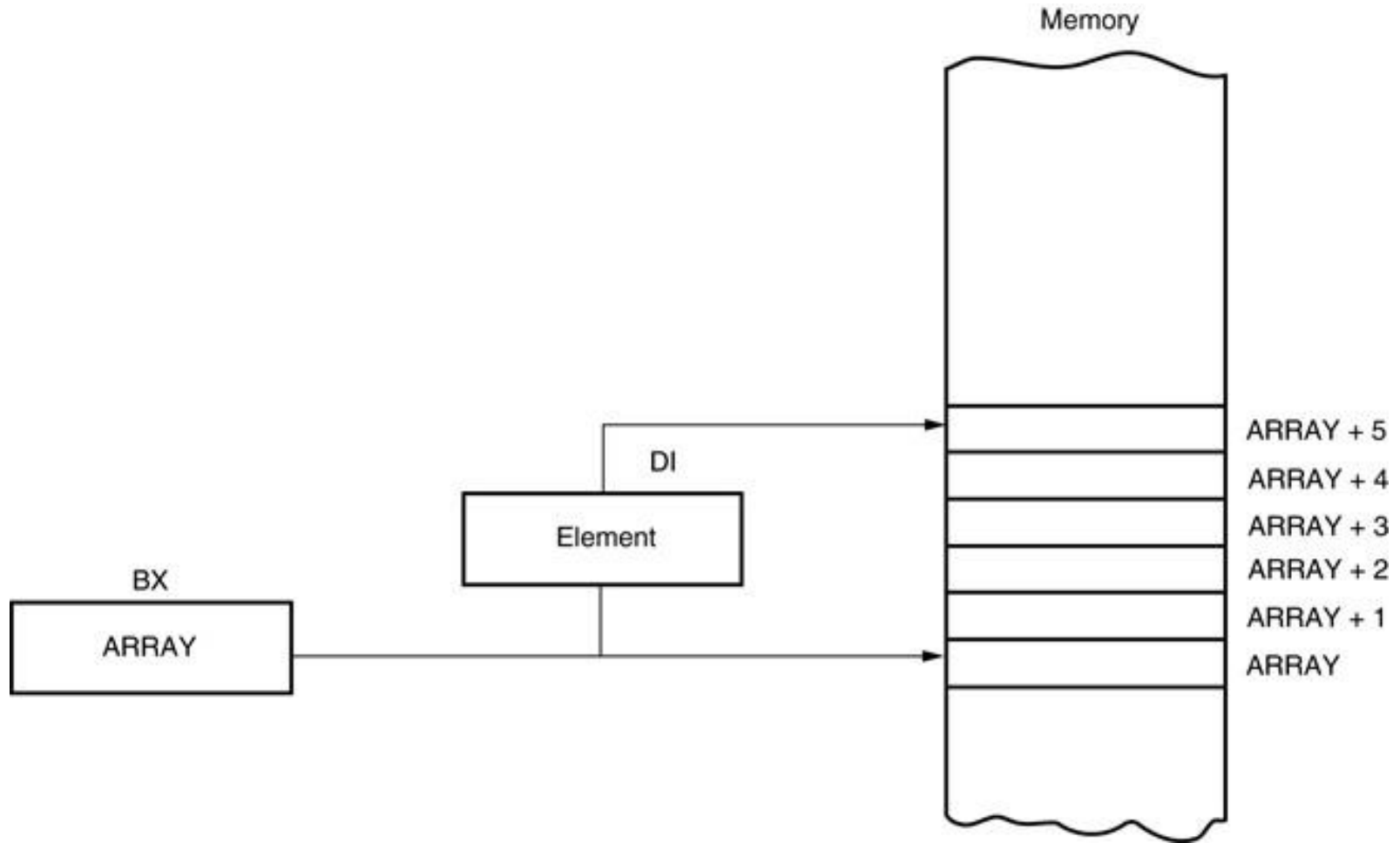
Figure 3–8 An example showing how the base-plus-index addressing mode functions for the MOV DX,[BX + DI] instruction. Notice that memory address 02010H is accessed because DS=0100H, BX=100H and DI=0010H.



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.

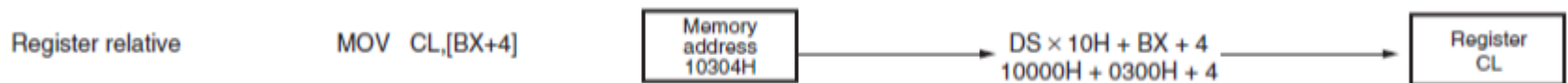
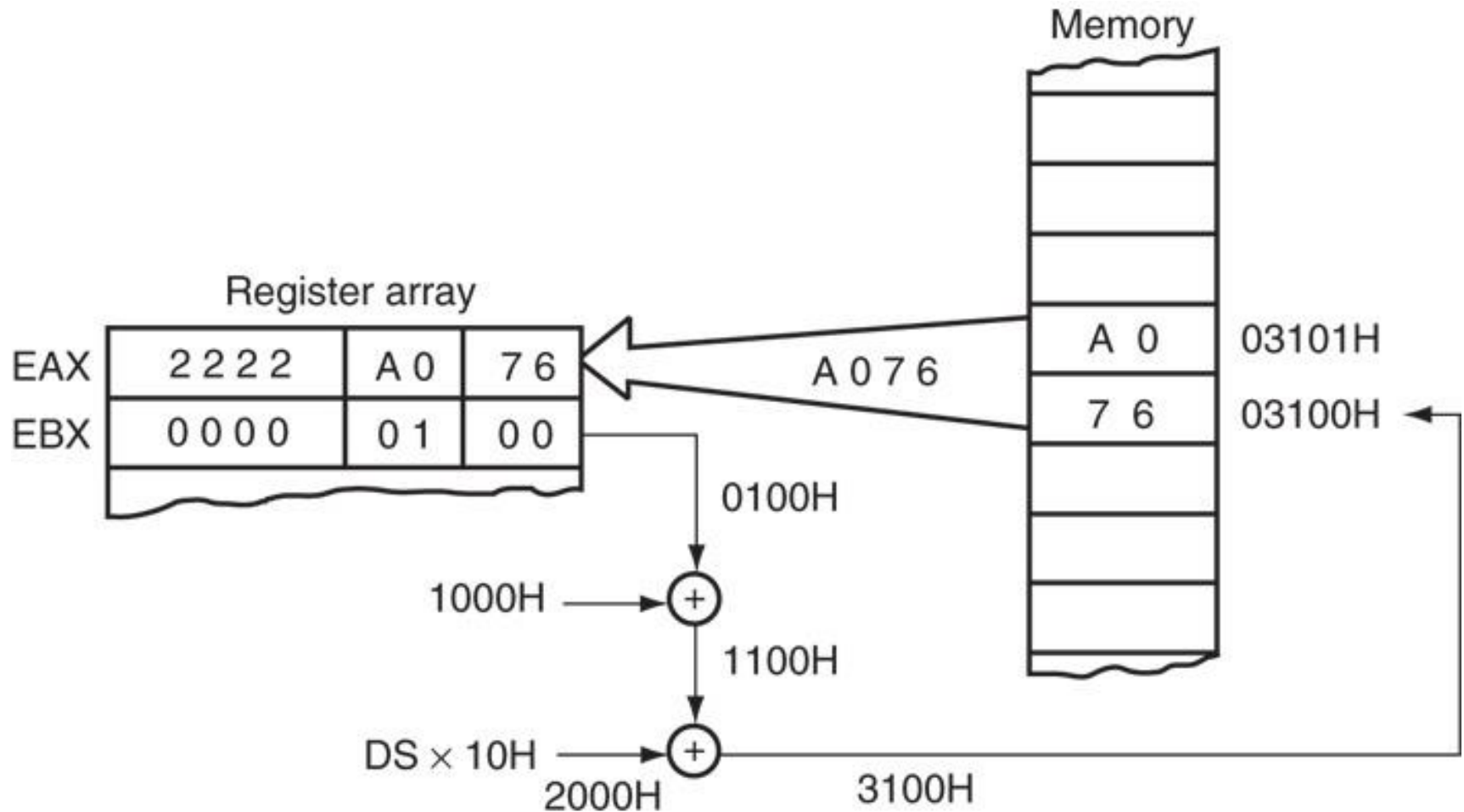


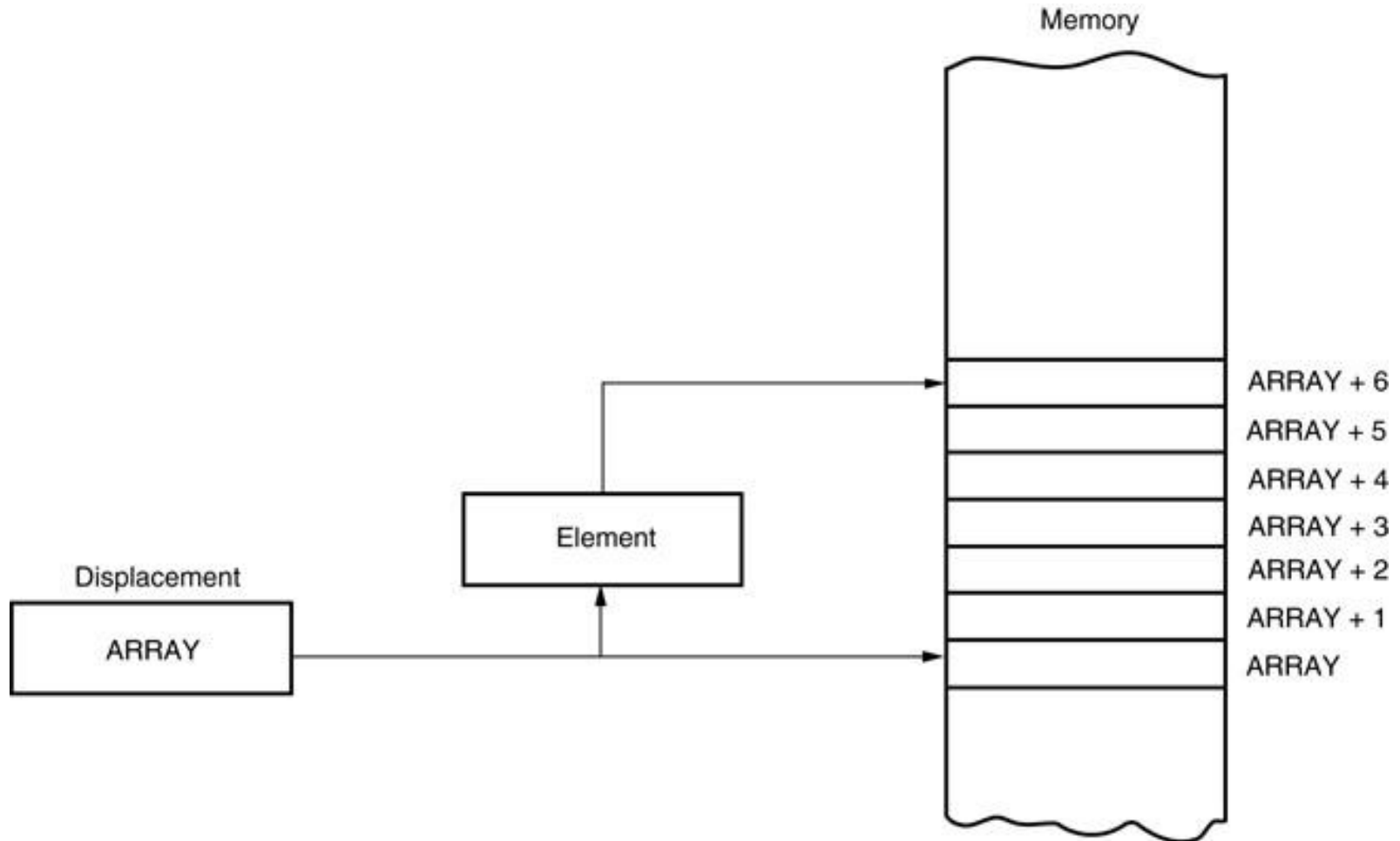
Figure 3–10 The operation of the MOV AX, [BX+1000H] instruction, when BX=0100H and DS=0200H .



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

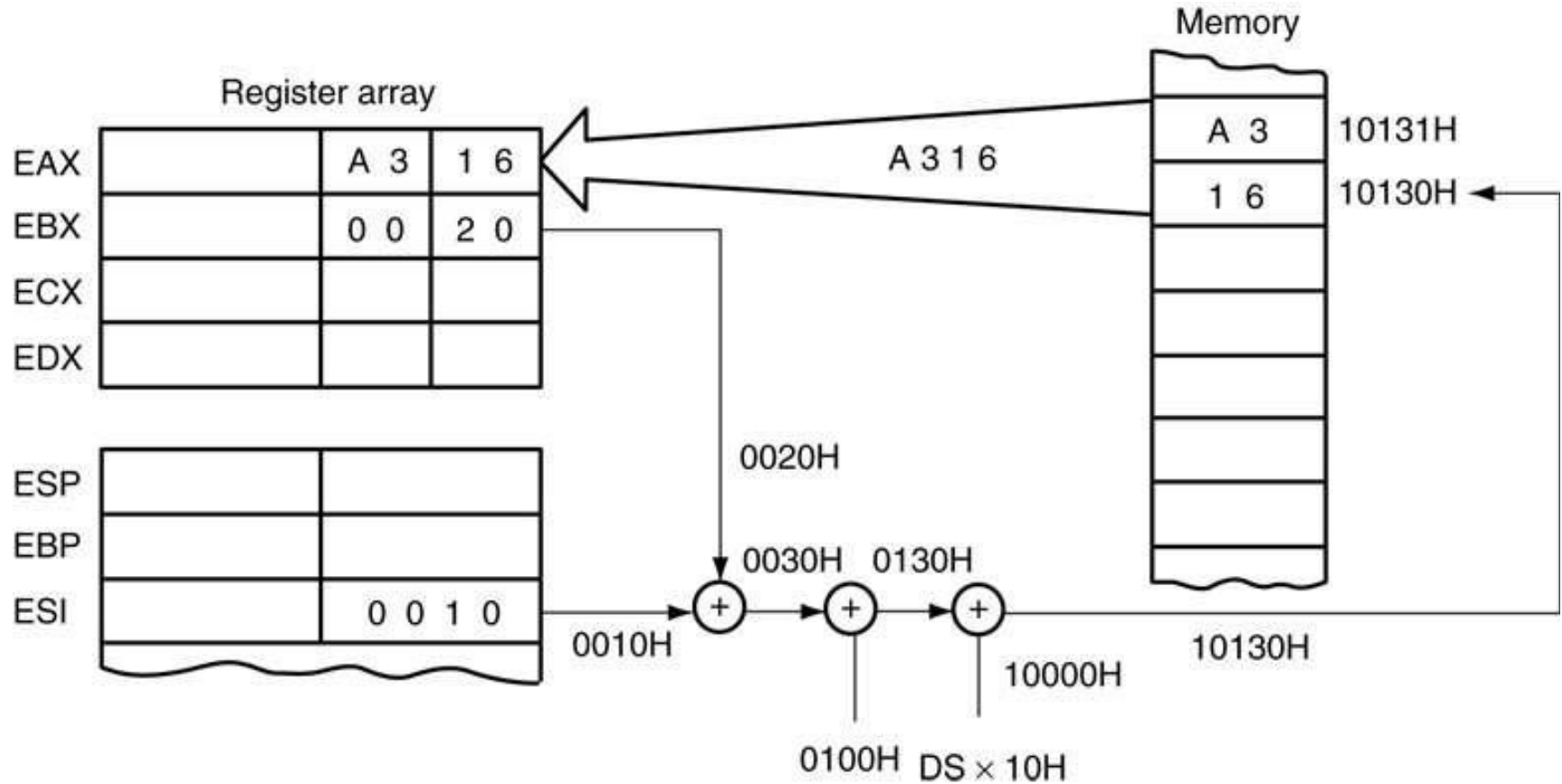
- Similar to base-plus-index addressing.
 - adds a displacement
 - uses a base register and an index register to form the memory address
- This type of addressing mode often addresses a two-dimensional array of memory data.



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.

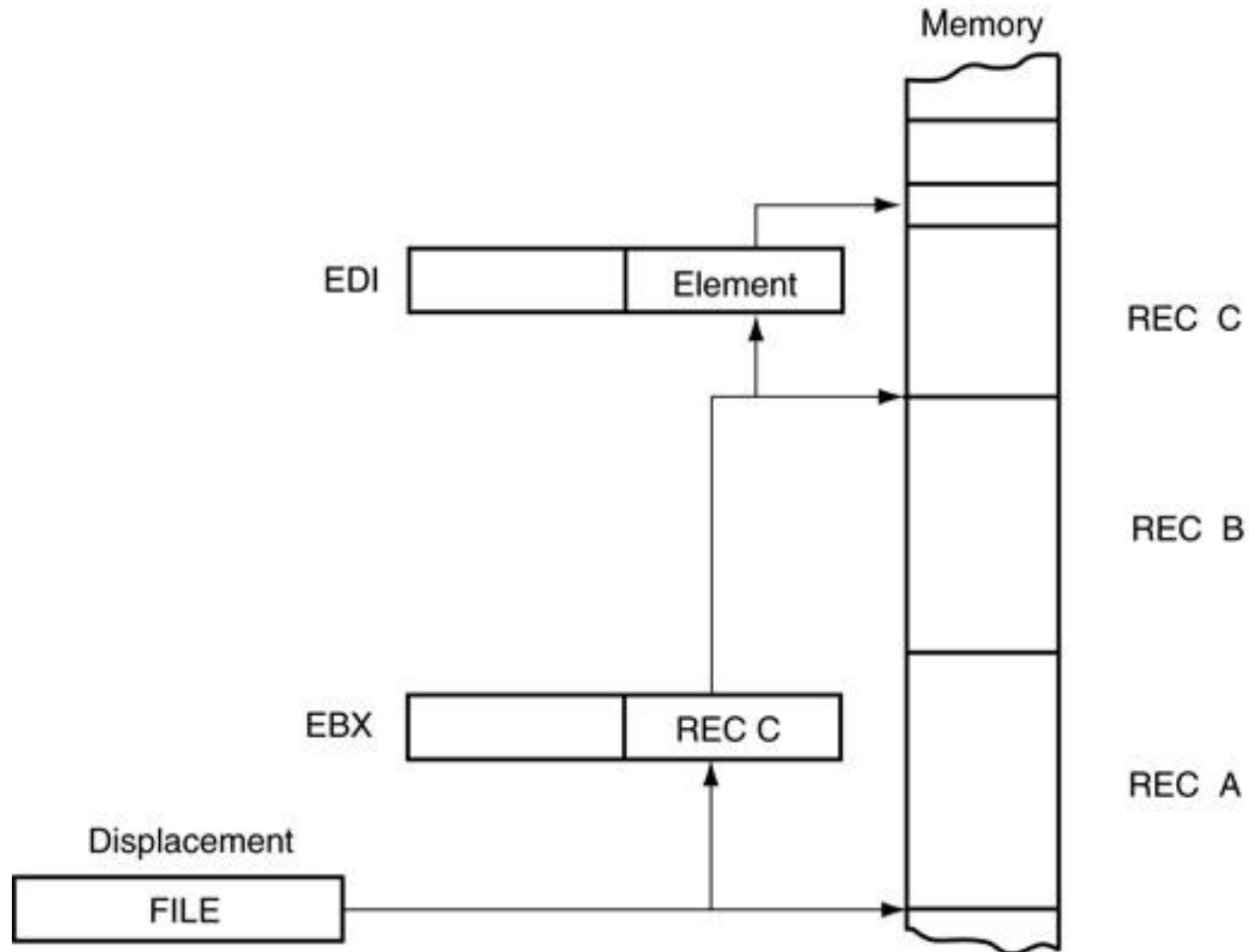
Figure 3–12 An example of base relative-plus-index addressing using a MOV AX,[BX+SI+100H] instruction. Note: DS=1000H



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).



Data Structures

- 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 and the end with the `ENDS` statement.

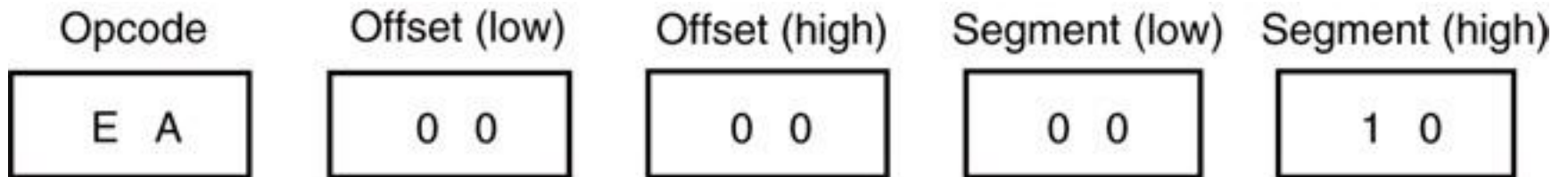
3–2 PROGRAM MEMORY-ADDRESSING MODES

- 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 [1000H] instruction.



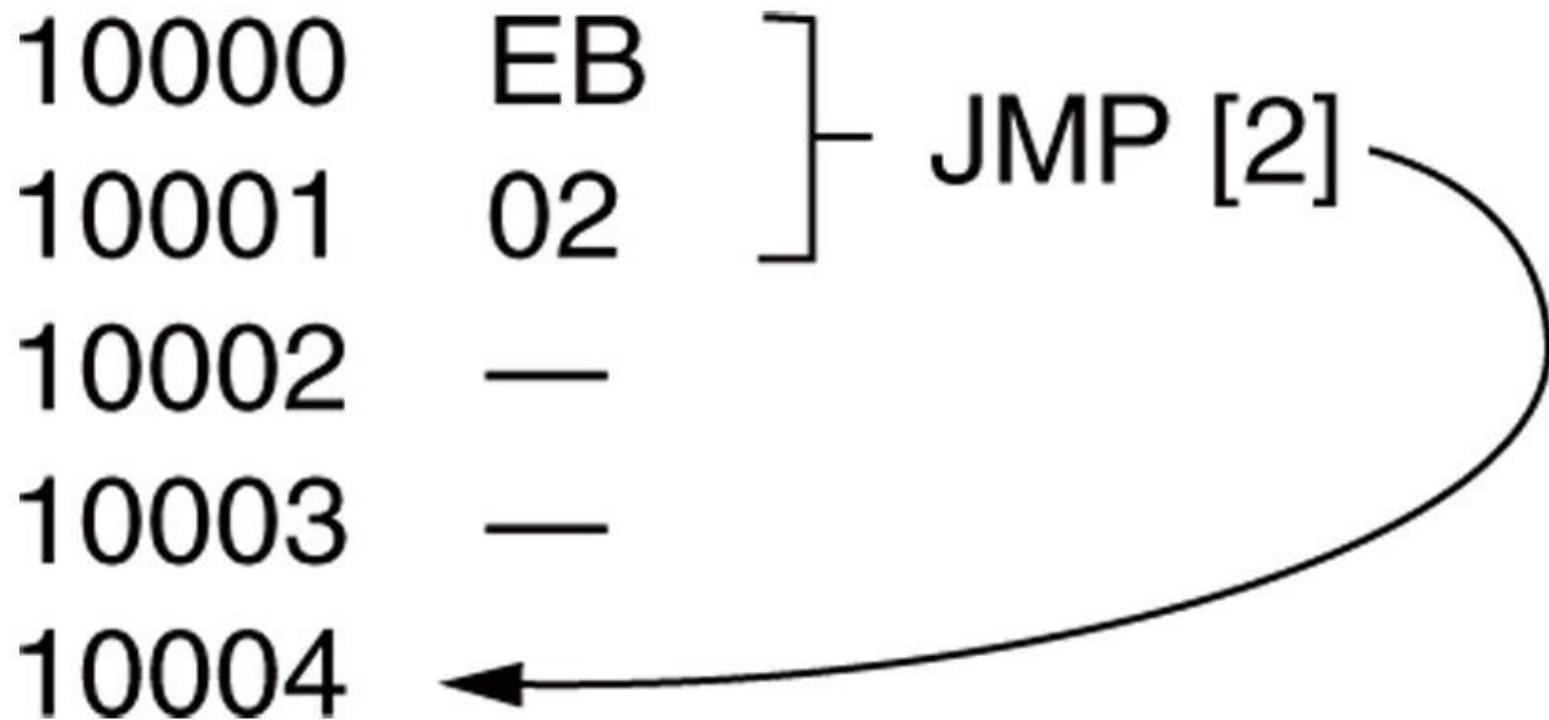
- This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction.
 - an **intersegment jump** is a jump to any memory location within the entire memory system
- Often called a *far jump* because it can jump to any memory location for the next instruction.
 - 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.

Table 3–10 Examples of indirect program memory addressing.

<i>Assembly Language</i>	<i>Operation</i>
JMP AX	Jumps to the current code segment location addressed by the contents of AX
JMP CX	Jumps to the current code segment location addressed by the contents of CX
JMP NEAR PTR[BX]	Jumps to the current code segment location addressed by the contents of the data segment location addressed by BX
JMP NEAR PTR[DI+2]	Jumps to the current code segment location addressed by the contents of the data segment memory location addressed by DI plus 2
JMP TABLE[BX]	Jumps to the current code segment location addressed by the contents of the data segment memory location address by TABLE plus BX
JMP ECX	Jumps to the current code segment location addressed by the contents of ECX
JMP RDI	Jumps to the linear address contained in the RDI register (64-bit mode)

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
```

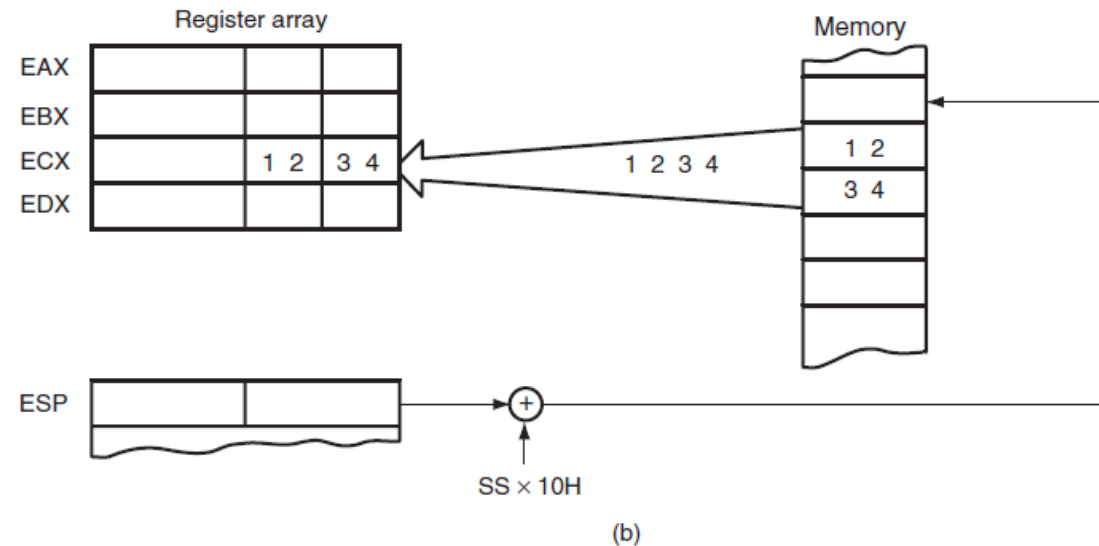
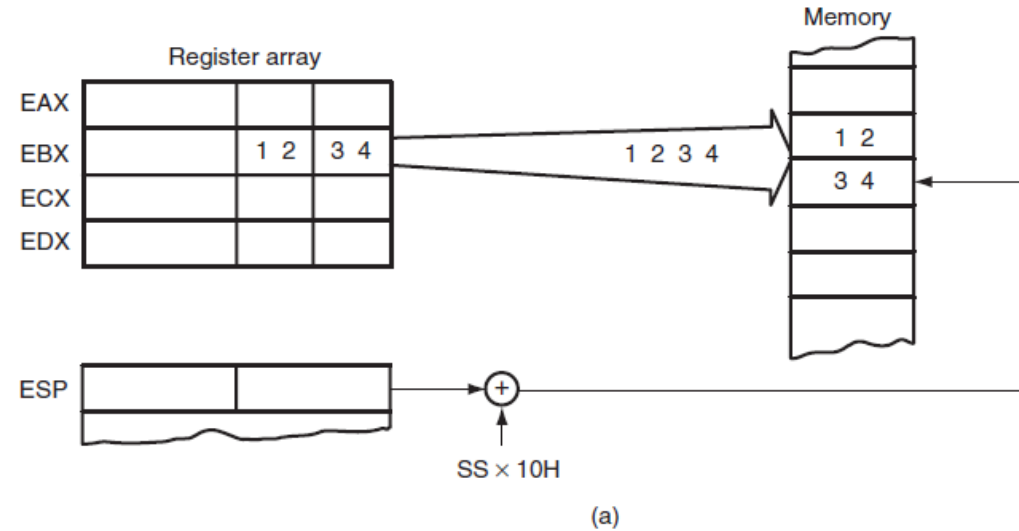

3–3 STACK MEMORY-ADDRESSING MODES

- 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.
- When data are popped from the stack, the low-order 8 bits are removed from the location addressed by SP.
- Then, high-order 8 bits are removed;
- And, 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

Copyright ©2009 by Pearson Education, Inc.
Upper Saddle River, New Jersey 07458 • All rights reserved.