
Assembly Language and Microcomputer Interface

Chapter 3 – Addressing Modes

Ming Cai

cm@zju.edu.cn

College of Computer Science and Technology
Zhejiang University

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.*)

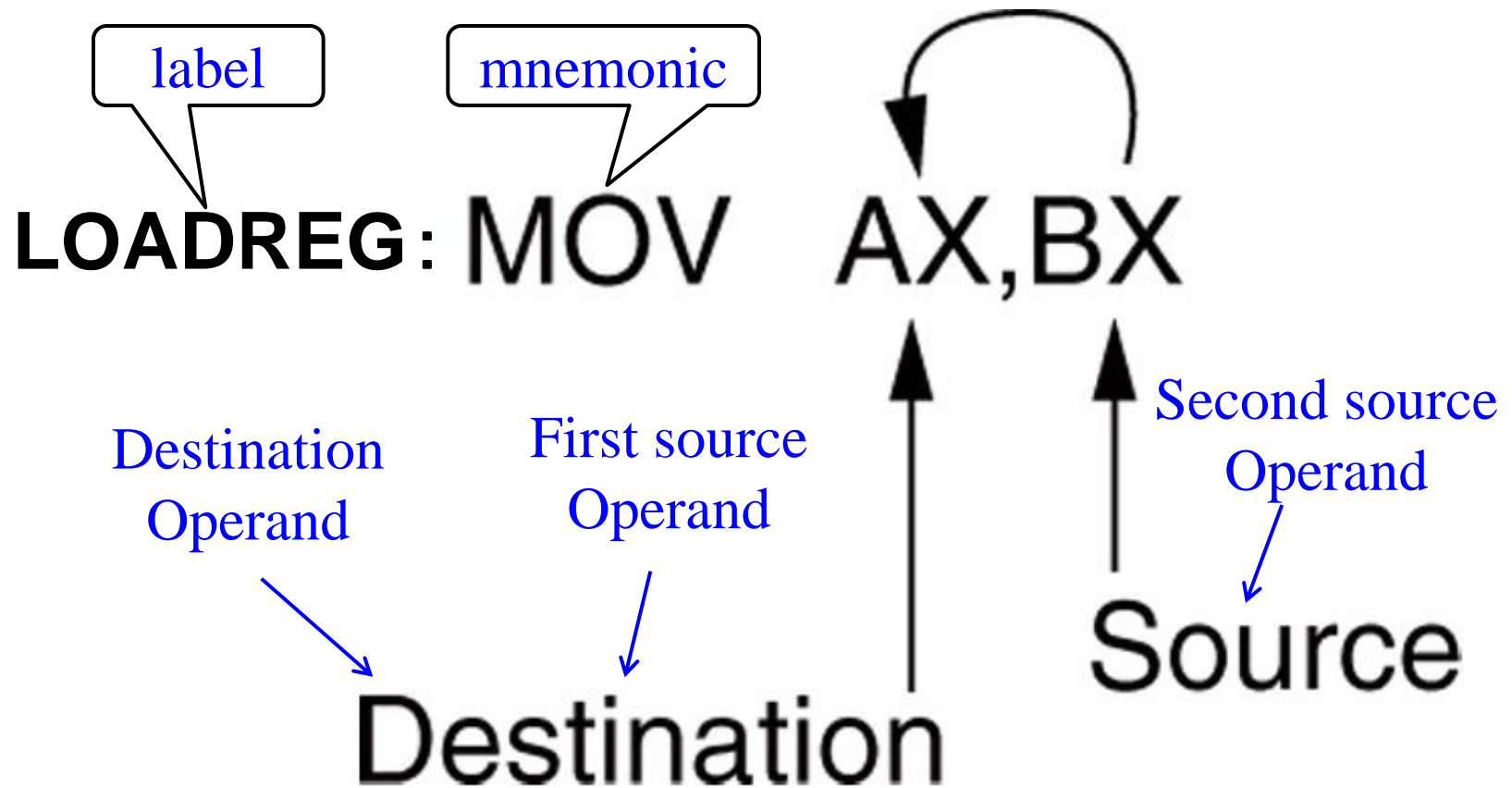
Upon completion of this chapter, you will be able to:

- Select the appropriate addressing mode to accomplish a given task.
- Detail the difference between addressing memory data using real mode and protected mode operation.
- 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.

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.



Note: Some assembly languages (e.g., AT&T syntax) put the source and destination in reverse order.

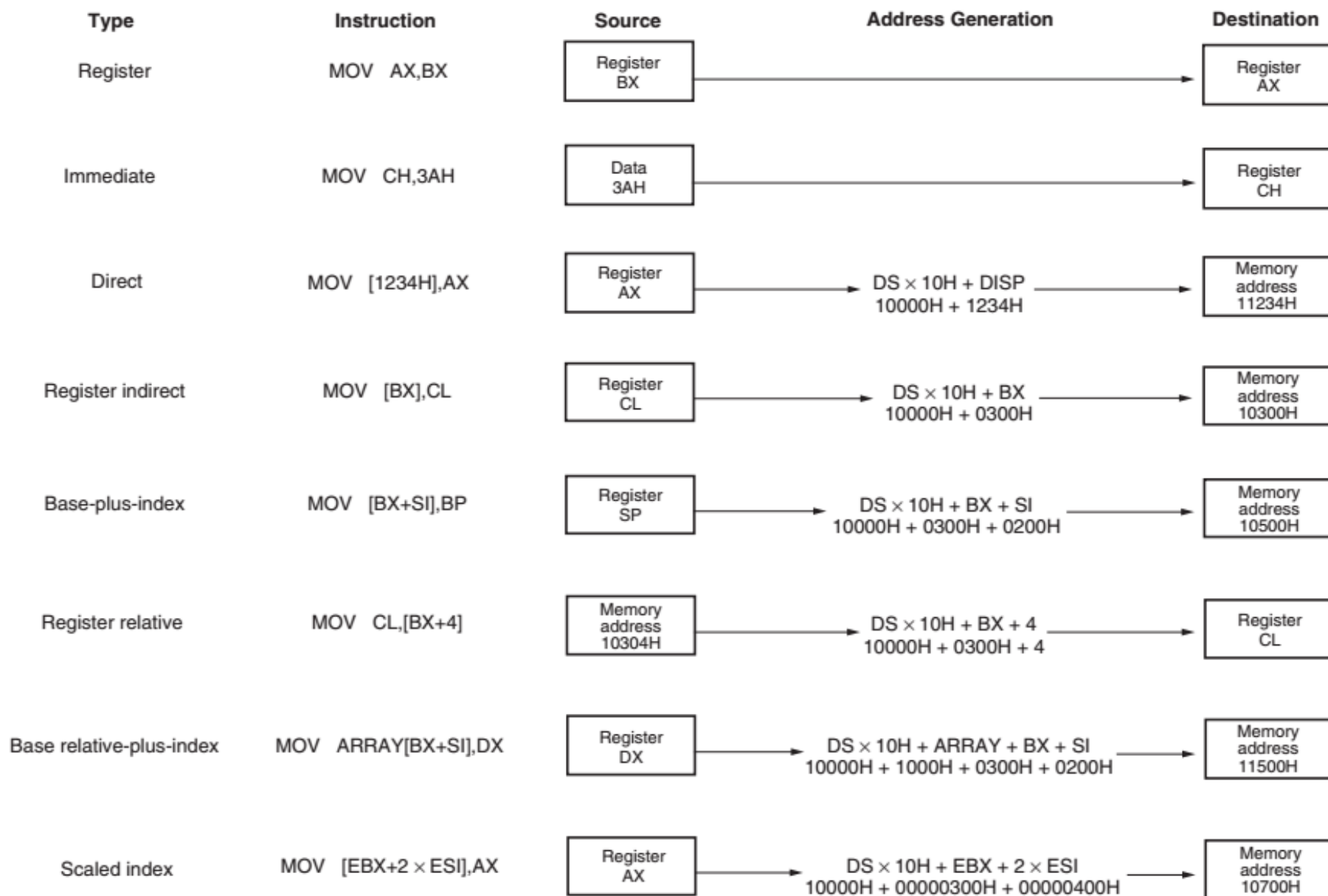
- Each statement in an assembly language program consists of four parts or fields.
- The leftmost field is called the *label*.
 - used to store a symbolic name for the memory location it represents
- All labels must begin with a letter or one of the following special characters: @, \$, -, or ?.
 - a 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 (;)

- There are **three basic types of operands**: immediate, register, and memory.
- An **immediate operand** is a constant value that is encoded as part of the instruction. Only source operands can specify an immediate value.
- **Register operands** are contained in a general purpose or SIMD register.
- A **memory operand** specifies a location in memory.

- 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.



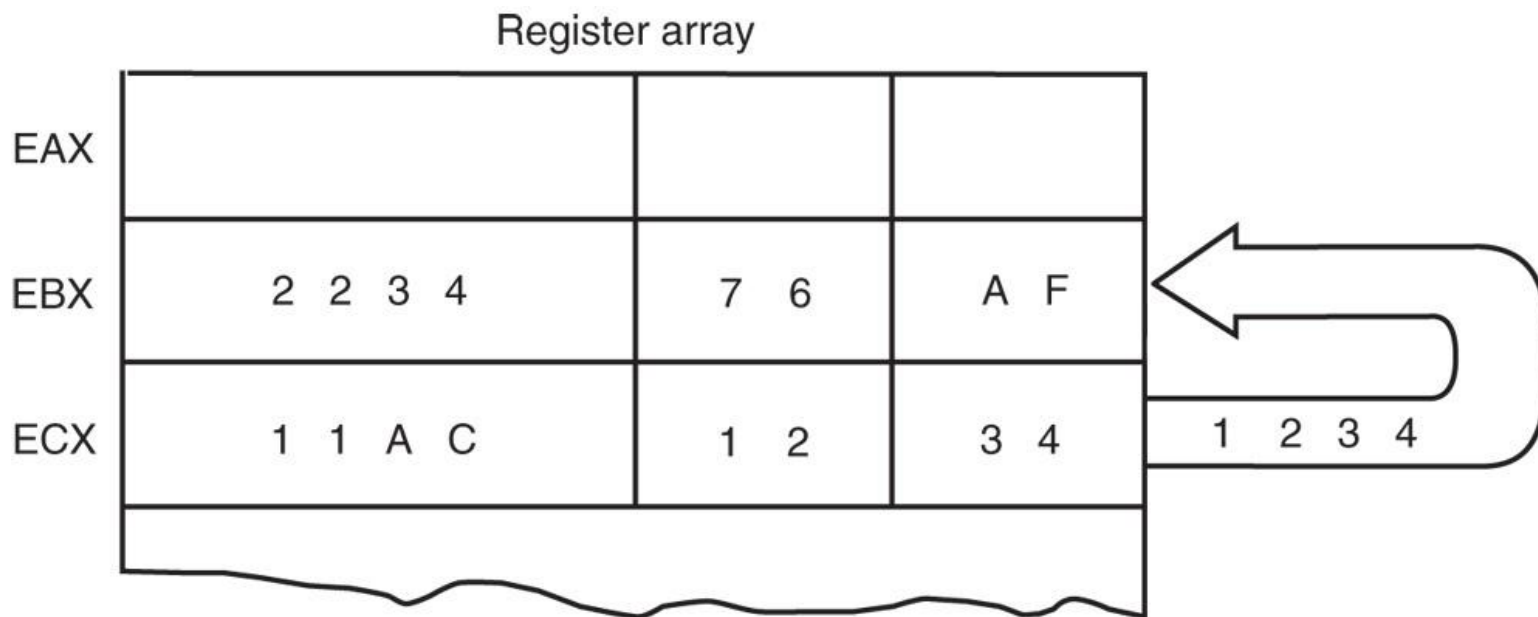
Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

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.

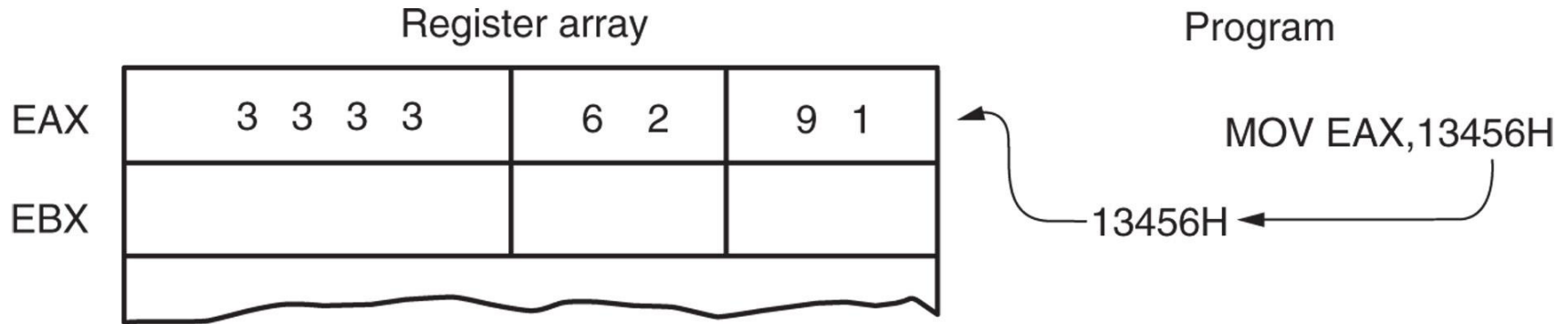


- Figure 3–3 shows the operation of the MOV BX, CX instruction.
- The source register's contents do not change.
 - the destination register's contents do change
- The contents of the destination register or destination memory location change for all instructions except the CMP and TEST instructions.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

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 # 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.
- In MASM, hexadecimal numbers must always start with a decimal digit (0–9). Otherwise they would be mistaken for label names.
- If necessary, add a leading zero to distinguish between symbols and hexadecimal numbers that start with a letter. E.g.,
 - MOV AX, F2H ; load a label named F2H
 - MOV AX, 0F2H ; load a hexadecimal F2H

- **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
- **Binary data** are represented if the binary number is followed by the letter B.
 - in some assemblers, the letter Y
- 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 (‘)

TABLE 3–2 Examples of immediate addressing

Assembly Language	Size	Operation
MOV BL,44	8 bits	Copies 44 decimal (2CH) into BL
MOV AX,44H	16 bits	Copies 0044H into AX
MOV SI,0	16 bits	Copies 0000H into SI
MOV CH,100	8 bits	Copies 100 decimal (64H) into CH
MOV AL,'A'	8 bits	Copies ASCII A into AL
MOV AH,1	8 bits	Not allowed in 64-bit mode, but allowed in 32- or 16-bit modes
MOV AX,'AB'	16 bits	Copies ASCII BA* into AX
MOV CL,11001110B	8 bits	Copies 11001110 binary into CL
MOV EBX,12340000H	32 bits	Copies 12340000H into EBX
MOV ESI,12	32 bits	Copies 12 decimal into ESI
MOV EAX,100B	32 bits	Copies 100 binary into EAX
MOV RCX,100H	64 bits	Copies 100H into RCX

*Note: This is not an error. The ASCII characters are stored in in reverse order as BA, so be care when using word-sized pairs of ASCII characters.

Direct Data Addressing

Effective Address = Base + (Scale x Index) + Disp



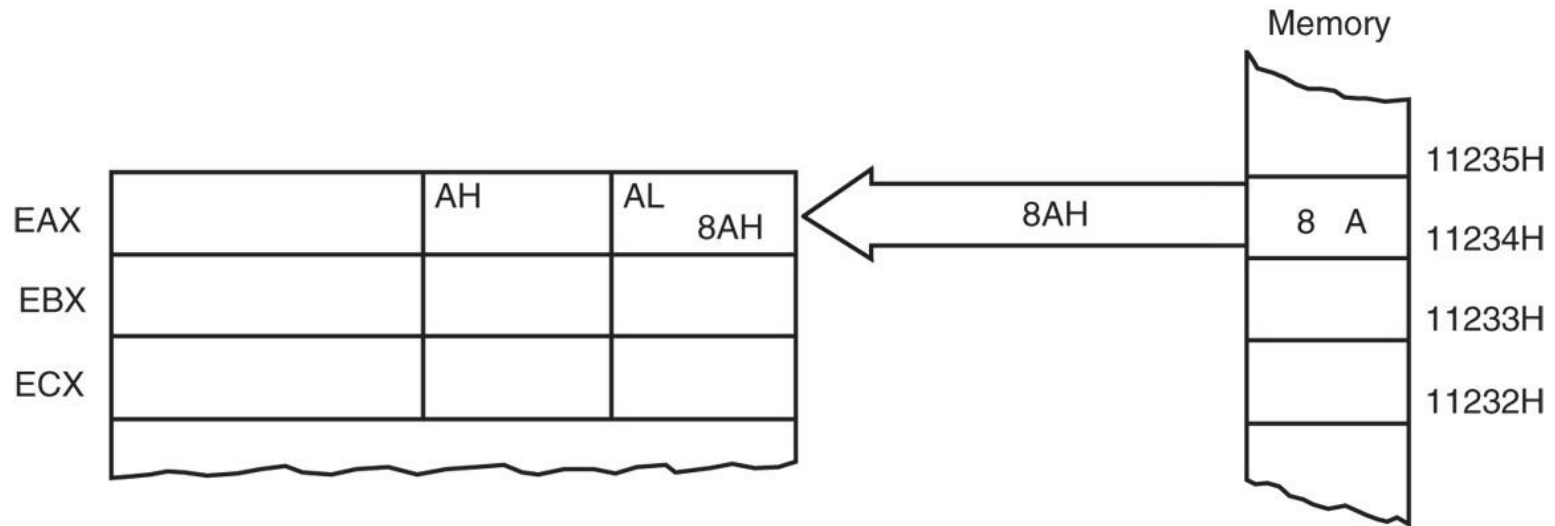
memory location

- 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

TABLE 3–3 Direct addressed instructions using EAX, AX, and AL and RAX in 64-bit mode

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV AL,NUMBER	8 bits	Copies the byte contents of data segment memory location NUMBER into AL
MOV AX,COW	16 bits	Copies the word contents of data segment memory location COW into AX
MOV EAX,WATER*	32 bits	Copies the doubleword contents of data segment location WATER into EAX
MOV NEWS,AL	8 bits	Copies AL into byte memory location NEWS
MOV THERE,AX	16 bits	Copies AX into word memory location THERE
MOV HOME,EAX*	32 bits	Copies EAX into doubleword memory location HOME
MOV ES:[2000H],AL	8 bits	Copies AL into extra segment memory at offset address 2000H
MOV AL,MOUSE	8 bits	Copies the contents of location MOUSE into AL; in 64-bit mode MOUSE can be any address
MOV RAX,WHISKEY	64 bits	Copies 8 bytes from memory location WHISKEY into RAX

Displacement Addressing

- Almost identical to direct addressing, except the instruction is 4 bytes wide instead of 3.

EXAMPLE 3–5

0000 A0 1234 R

MOV AL, DS: [1234H]

0003 BA 0E 1234 R

MOV CL, DS: [1234H]

- 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.

TABLE 3–4 Examples of direct data addressing using a displacement

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV CH,DOG	8 bits	Copies the byte contents of data segment memory location DOG into CH
MOV CH,DS:[1000H]*	8 bits	Copies the byte contents of data segment memory offset address 1000H into CH
MOV ES,DATA6	16 bits	Copies the word contents of data segment memory location DATA6 into ES
MOV DATA7,BP	16 bits	Copies BP into data segment memory location DATA7
MOV NUMBER,SP	16 bits	Copies SP into data segment memory location NUMBER
MOV DATA1,EAX	32 bits	Copies EAX into data segment memory location DATA1
MOV EDI,SUM1	32 bits	Copies the doubleword contents of data segment memory location SUM1 into EDI

Register Indirect Addressing

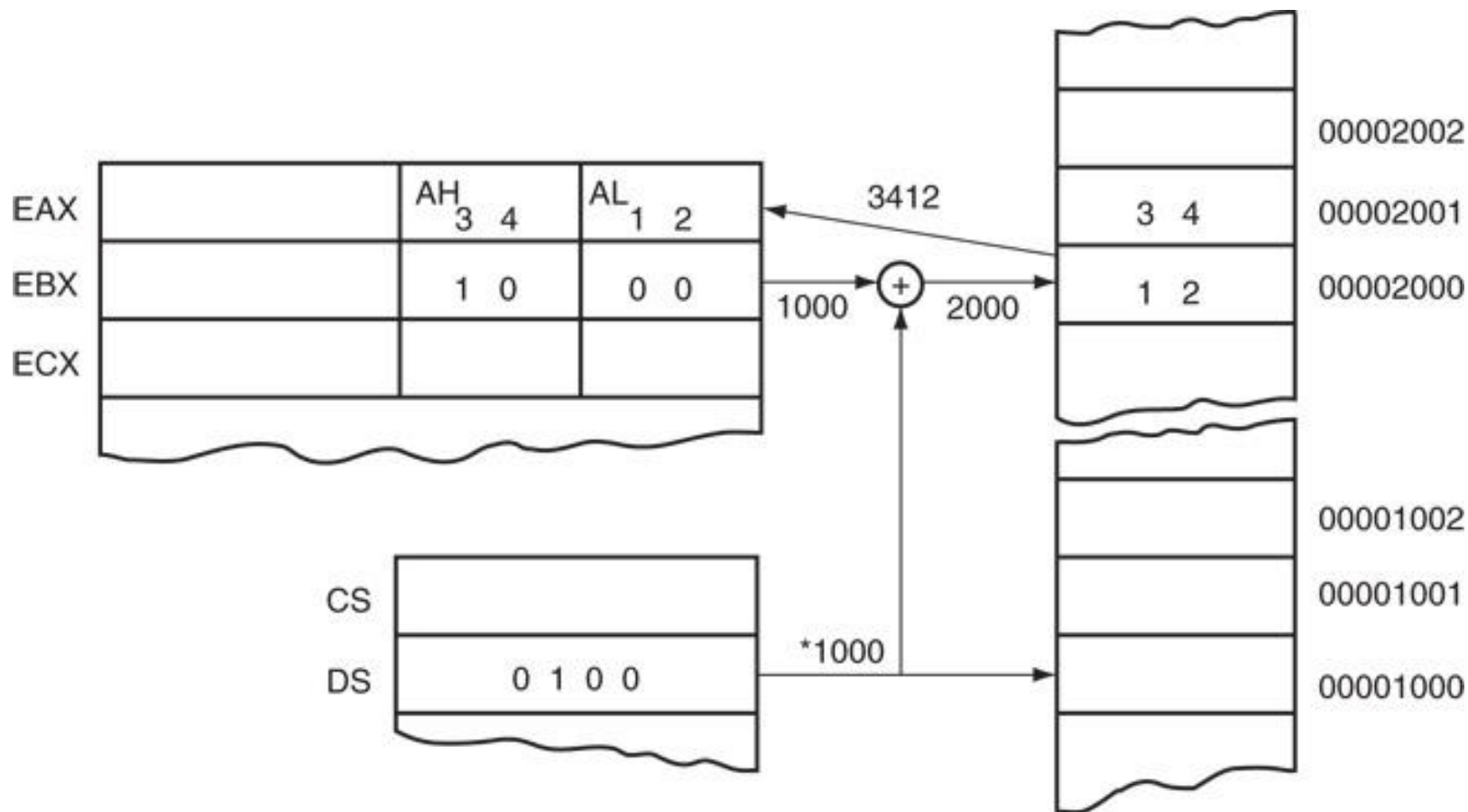
Effective Address = Base + (Scale x Index) + Disp



memory location (GPR)

- For indirect addressing
 - In the 8086 through the 80286, indirect addressing can only use the BX, BP, SI and DI registers, e.g., `MOV AX,[BX]`.
 - 80386 and above allow any extended register except ESP, e.g., `MOV AX,[EDX]`
 - 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.

TABLE 3–5 Examples of register indirect addressing

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV CX,[BX]	16 bits	Copies the word contents of the data segment memory location addressed by BX into CX
MOV [BP],DL*	8 bits	Copies DL into the stack segment memory location addressed by BP
MOV [DI],BH	8 bits	Copies BH into the data segment memory location addressed by DI
MOV [DI],[BX]	—	Memory-to-memory transfers are not allowed except with string instructions
MOV AL,[EDX]	8 bits	Copies the byte contents of the data segment memory location addressed by EDX into AL
MOV ECX,[EBX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by EBX into ECX
MOV RAX,[RDX]	64 bits	Copies the quadword contents of the memory location address by the linear address located in RDX into RAX (64-bit mode)

- 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.

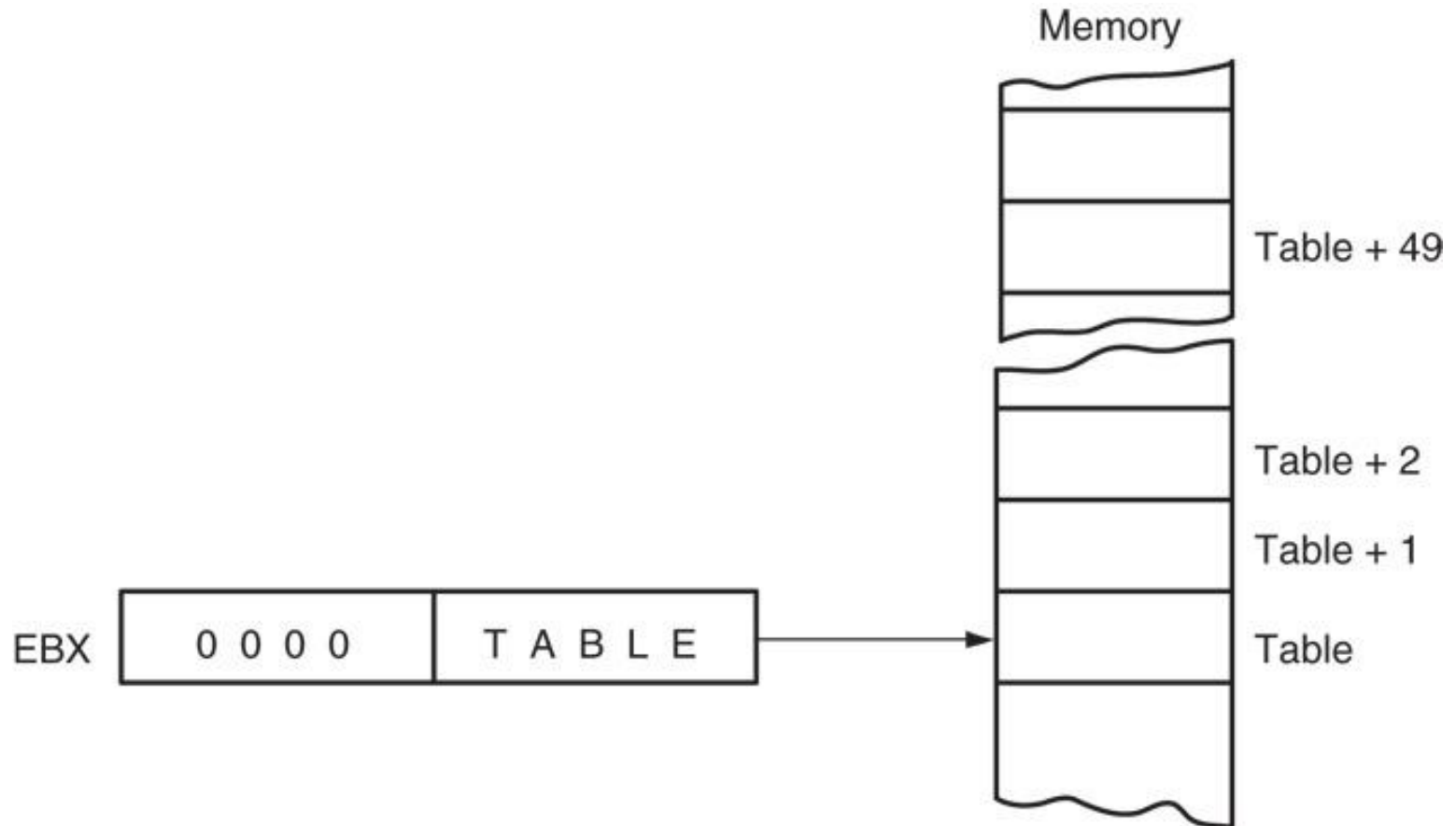
- When using a 32-bit register to address memory in the real mode, contents of the register must never exceed 0000FFFFH.
- In the protected mode, any value can be used in a 32-bit register that is used to indirectly address memory.
 - as long as it does not access a location outside the segment, dictated 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)
- For example, the `MOV AL,[DI]` instruction is clearly a byte-sized move instruction, but the `MOV [DI],10H` instruction is ambiguous.
- The instruction `MOV BYTE PTR [DI],10H` clearly designates the location addressed by DI as a byte-sized memory location.

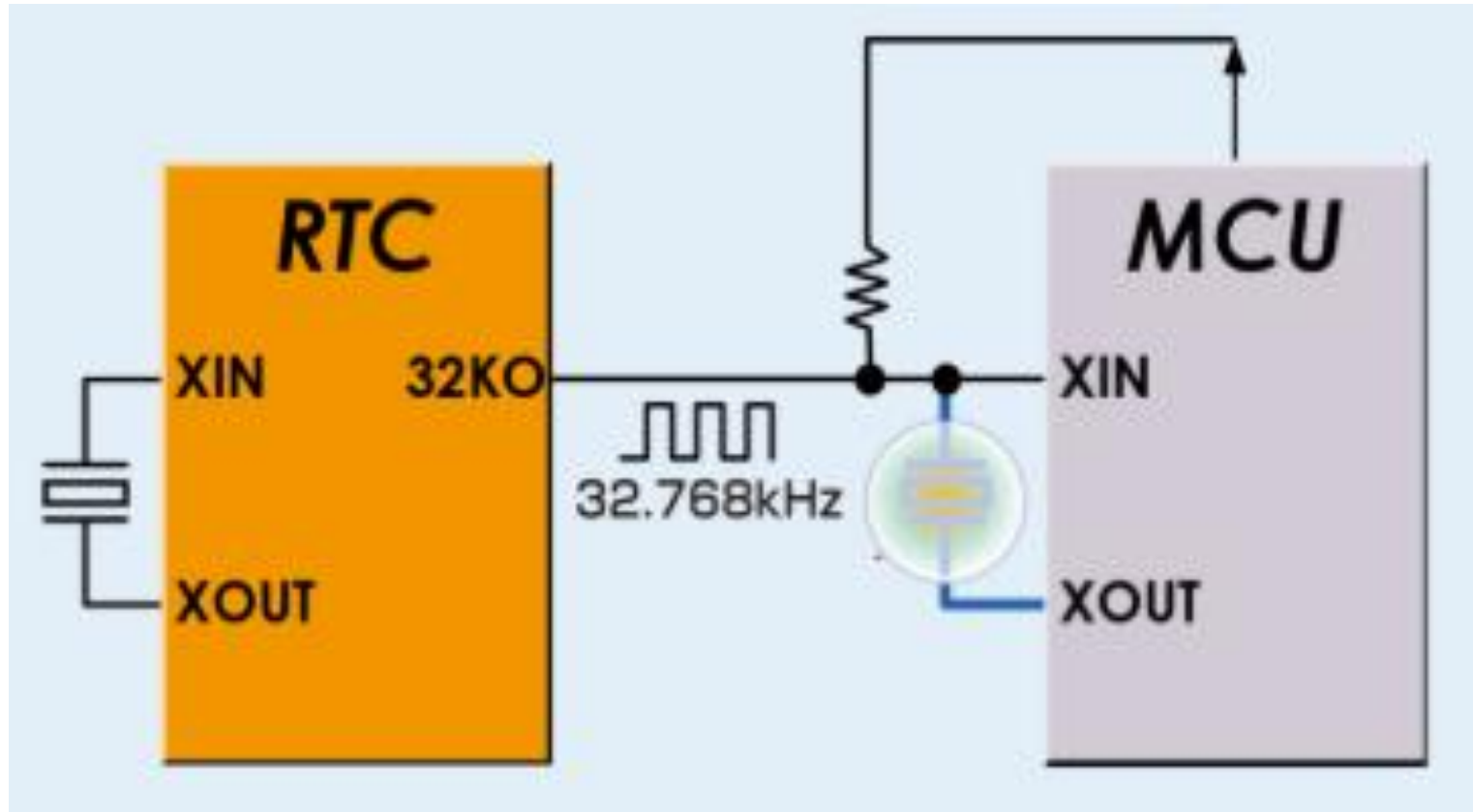
- Likewise, the **MOV DWORD PTR [DI],10H** identifies the memory location as doubleword-sized.
- The BYTE PTR, WORD PTR,DWORD PTR, and QWORD PTR directives are used only with instructions that address a memory location through a pointer or index register with immediate data.
- With SIMD instructions, the octal QWORD 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.



EXAMPLE 3–7 create a table that contains 50 samples taken from memory location 0000:046C (real-time clock)



EXAMPLE 3–7 create a table that contains 50 samples taken from memory location 0000:046C (real-time clock)

```
.MODEL SMALL          ;select small model
.DATA                ;start data segment

[ DATAS DW 50 DUP(?) ;setup array of 50 words 0000]

.CODE                ;start code segment
.STARTUP             ;start program
MOV AX,0
MOV ES,AX            ;address segment 0000 with ES
MOV BX,OFFSET DATAS ;address DATAS array with BX
MOV CX,50            ;load counter with 50
AGAIN:
MOV AX,ES:[046CH]    ;get clock value
MOV [BX],AX          ;save clock value in DATAS
INC BX               ;increment BX to next element
INC BX
LOOP AGAIN           ;repeat 50 times

.EXIT               ;exit to DOS
END                 ;end program listing
```

Base-Plus-Index Addressing

Effective Address = Base + (Scale x Index) + Disp



memory location (GPRs)

- 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

Base-Plus-Index Addressing

Effective Address = Base + (Scale x Index) + Disp



memory location (GPRs)

- For base-plus-index addressing
 - In the 8086 through the 80286, this type of addressing uses one base register (BP or BX) and one index register (DI or SI) to indirectly address memory, e.g., MOV DX,[BX + DI].
 - 80386 and above allow the combination of any two 32-bit extended registers except ESP, e.g., MOV DL,[EAX+EBX].

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=1000H and DI=0010H.

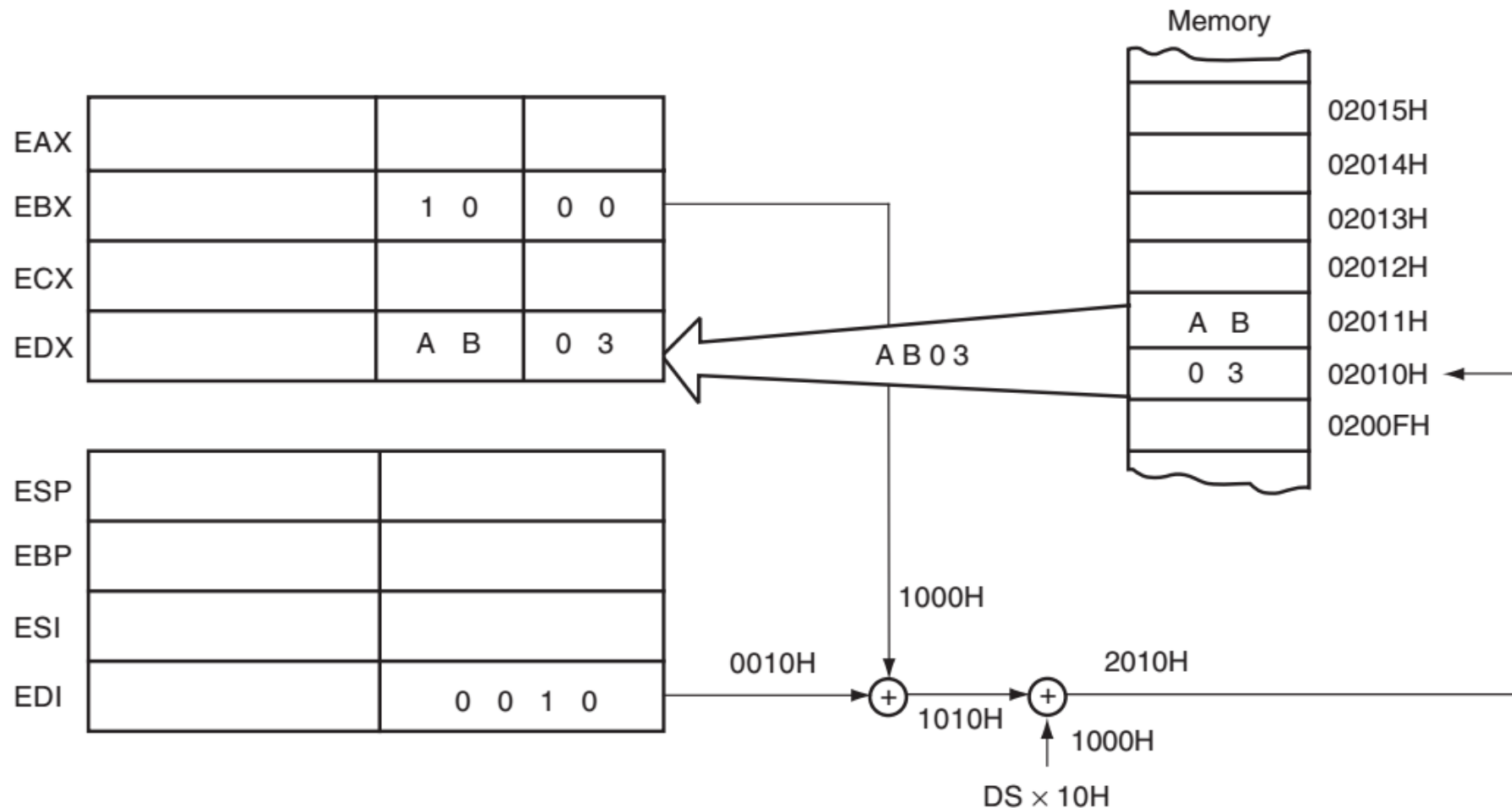


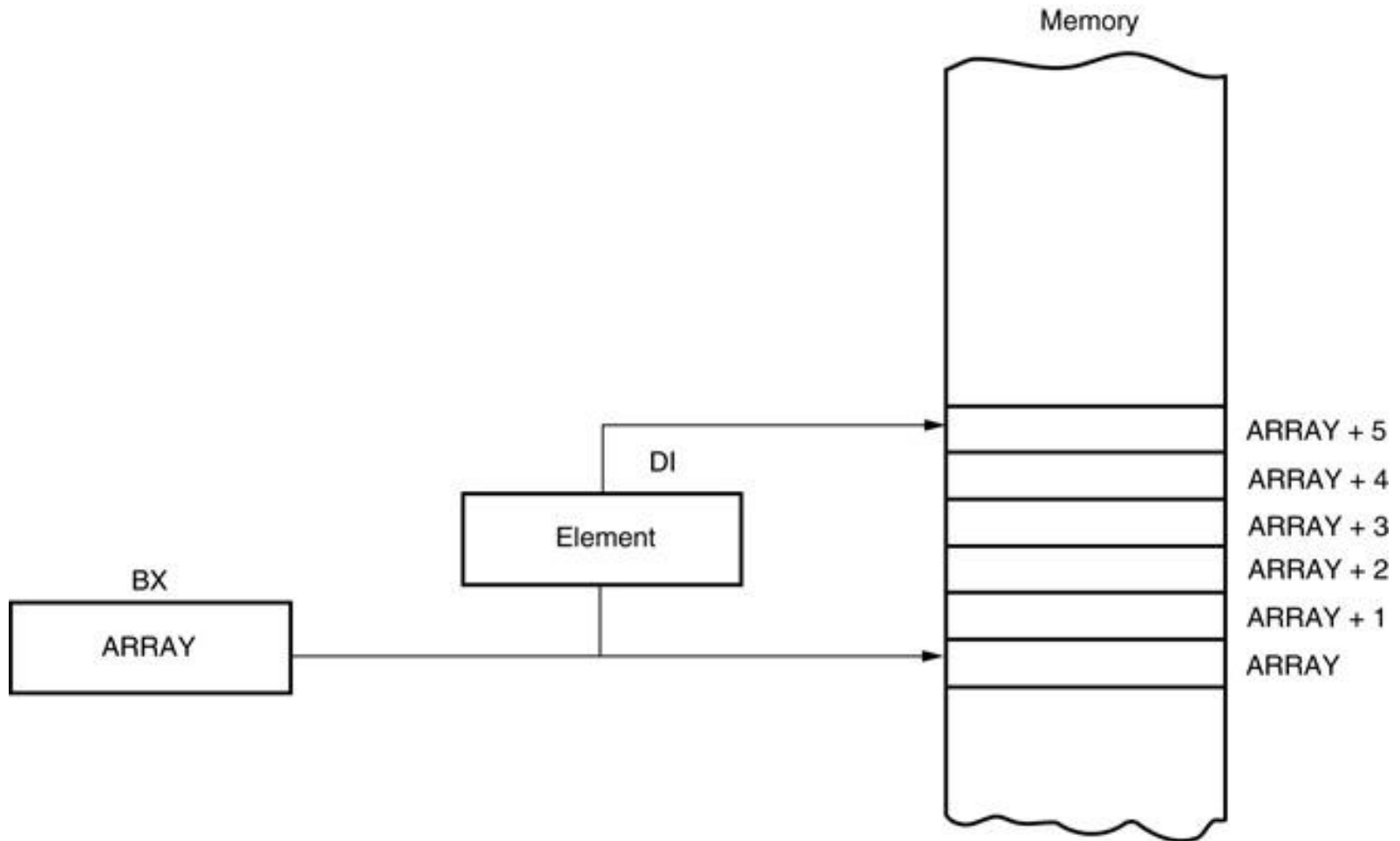
TABLE 3–6 Examples of base-plus-index addressing

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV CX,[BX+DI]	16 bits	Copies the word contents of the data segment memory location addressed by BX plus DI into CX
MOV CH,[BP+SI]	8 bits	Copies the byte contents of the stack segment memory location addressed by BP plus SI into CH
MOV [BX+SI],SP	16 bits	Copies SP into the data segment memory location addressed by BX plus SI
MOV [BP+DI],AH	8 bits	Copies AH into the stack segment memory location addressed by BP plus DI
MOV CL,[EDX+EDI]	8 bits	Copies the byte contents of the data segment memory location addressed by EDX plus EDI into CL
MOV [EAX+EBX],ECX	32 bits	Copies ECX into the data segment memory location addressed by EAX plus EBX
MOV [RSI+RBX],RAX	64 bit	Copies RAX into the linear memory location addressed by RSI plus RBX (64-bit mode)

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.



EXAMPLE 3–8 moves array element 10H into element 20H

```
0000                                .MODEL SMALL                ;select small model
                                .DATA                          ;start data segment
0000 0010 [                        ARRAY DB 16 DUP(?)          ;setup array of 16 bytes
                                00
                                ]
0010 29                            DB 29H                      ;element 10H
0011 001E [                        DB 20 dup(?)
                                00
                                ]
0000                                .CODE                        ;start code segment
                                .STARTUP

0017 B8 0000 R                      MOV BX,OFFSET ARRAY        ;address ARRAY
001A BF 0010                        MOV DI,10H                 ;address element 10H
001D 8A 01                          MOV AL,[BX+DI]             ;get element 10H
001F BF 0020                        MOV DI,20H                 ;address element 20H
0022 88 01                          MOV [BX+DI],AL            ;save in element 20H

                                .EXIT                          ;exit to DOS
                                END                            ;end program
```

Register Relative Addressing

Effective Address = Base + (Scale x Index) + Disp

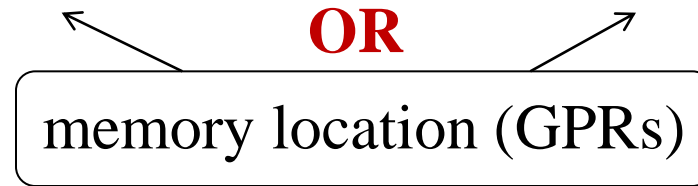
OR

memory location (GPRs)

- For register relative addressing
 - In the 8086 through the 80286, the memory data are addressed by adding the displacement to the contents of a base register (BP or BX) or an index register (DI or SI), e.g., MOV AX,[DI+100H].
 - In the 80386 and above, the displacement can be a 32-bit number and the register can be any 32-bit register except the ESP, e.g., MOV DL,[EAX+10H].

Register Relative Addressing

Effective Address = Base + (Scale x Index) + Disp



- 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**.
- 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=100H and DS=0200H .

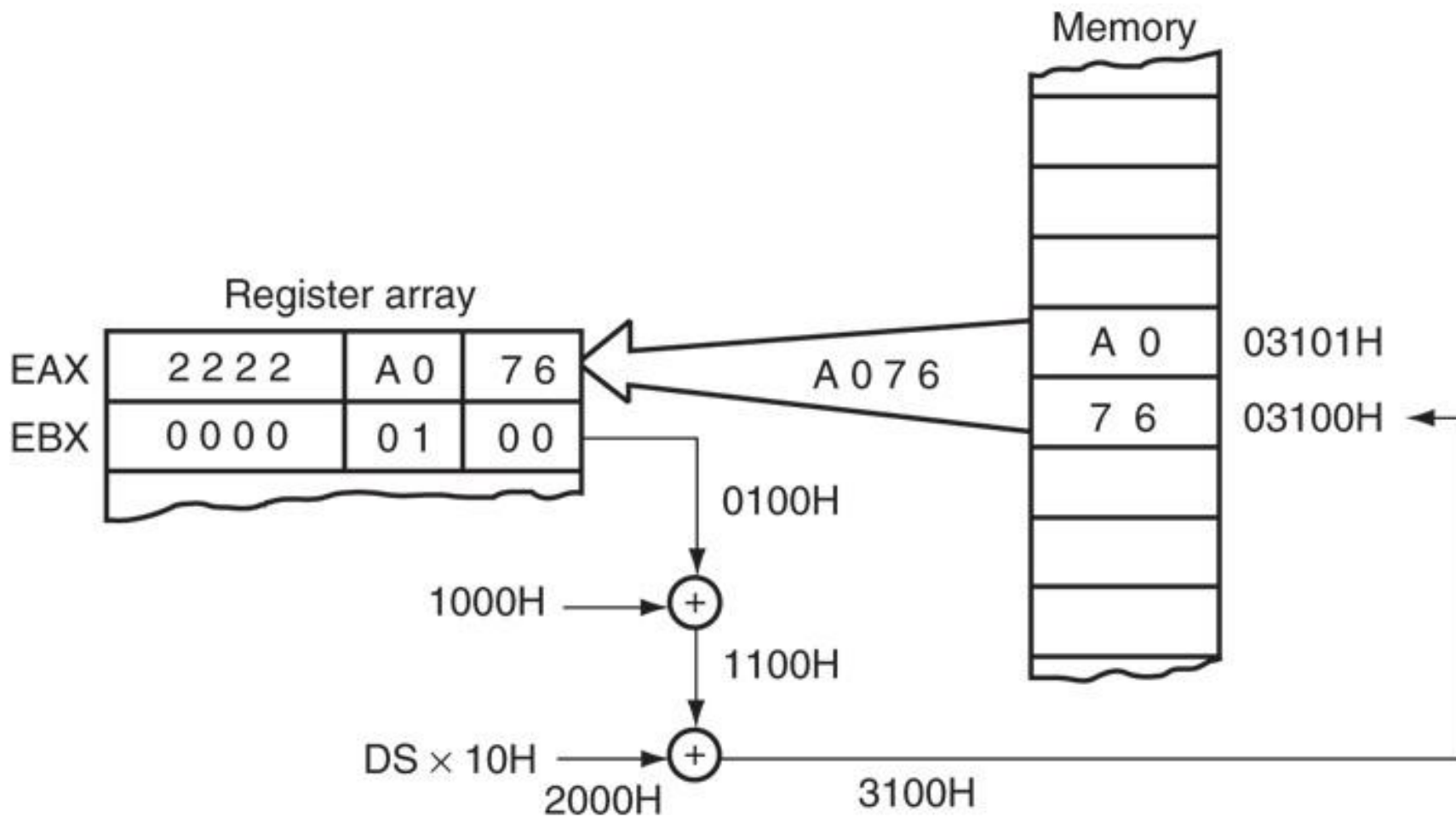


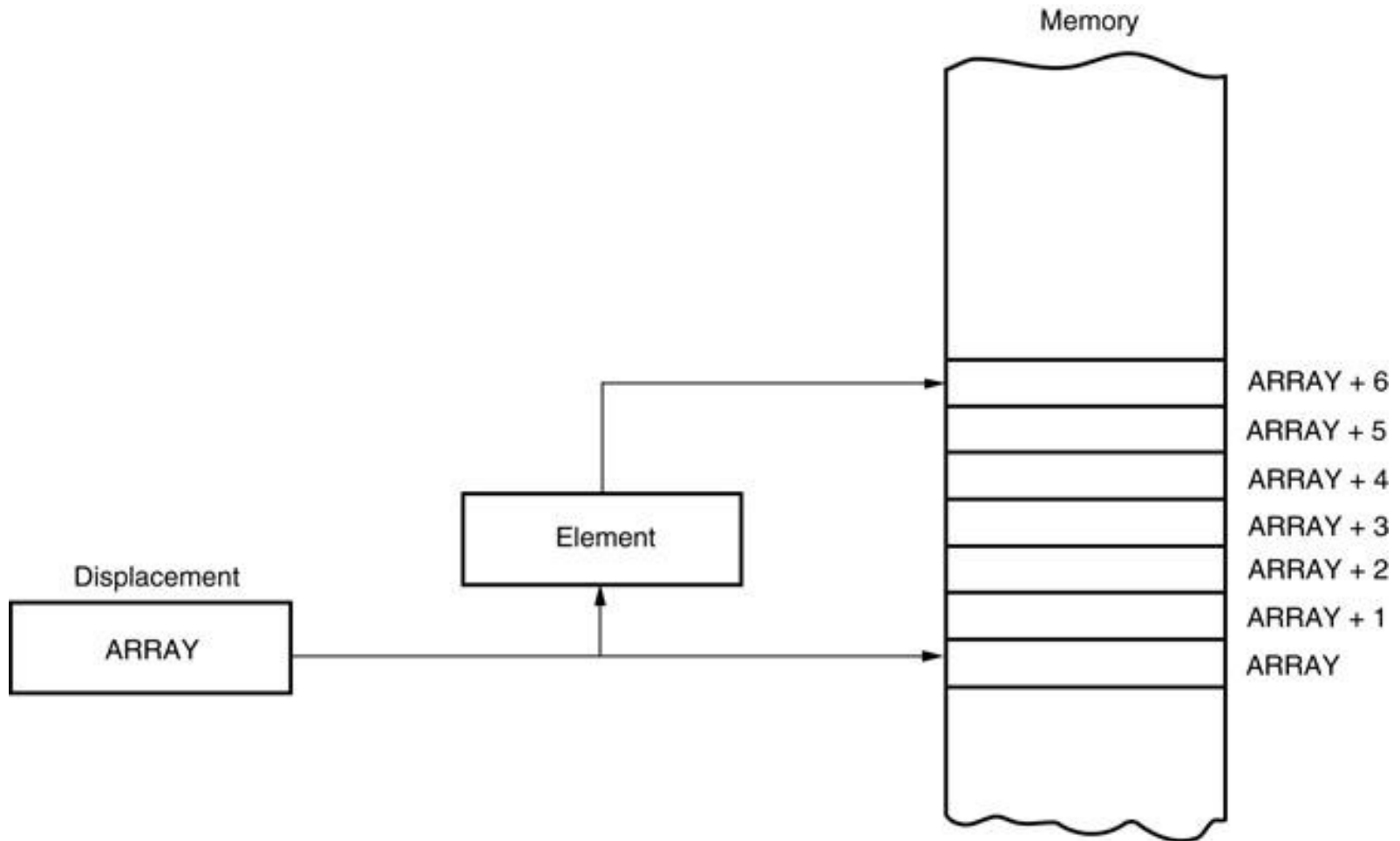
TABLE 3–7 Examples of register relative addressing

<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV AX,[DI+100H]	16 bits	Copies the word contents of the data segment memory location addressed by DI plus 100H into AX
MOV ARRAY[SI],BL	8 bits	Copies BL into the data segment memory location addressed by ARRAY plus SI
MOV LIST[SI+2],CL	8 bits	Copies CL into the data segment memory location addressed by the sum of LIST, SI, and 2
MOV DI,SET_IT[BX]	16 bits	Copies the word contents of the data segment memory location addressed by SET_IT plus BX into DI
MOV DI,[EAX+10H]	16 bits	Copies the word contents of the data segment location addressed by EAX plus 10H into DI
MOV ARRAY[EBX],EAX	32 bits	Copies EAX into the data segment memory location addressed by ARRAY plus EBX
MOV ARRAY[RBX],AL	8 bits	Copies AL into the memory location ARRAY plus RBX (64-bit mode)
MOV ARRAY[RCX],EAX	32 bits	Copies EAX into memory location ARRAY plus RCX (64-bit mode)

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.



EXAMPLE 3–9 transfer the contents of array element 10H into array element 20H

```

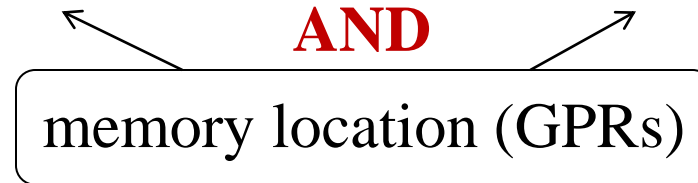
        .MODEL  SMALL                ;select small model
        .DATA                        ;start data segment
ARRAY   DB    16  dup(?)             ;setup ARRAY

        DB    29                      ;element 10H
        DB    30  dup(?)

        .CODE                        ;start code segment
        .STARTUP                     ;start program
MOV     DI,10H                       ;address element 10H
MOV     AL,ARRAY[DI]                 ;get ARRAY element 10H
MOV     DI,20H                       ;address element 20H
MOV     ARRAY[DI],AL                 ;save it in element 20H
        .EXIT                         ;exit to DOS
        END                          ;end of program
```

Base Relative-Plus-Index Addressing

Effective Address = Base + (Scale x Index) + Disp

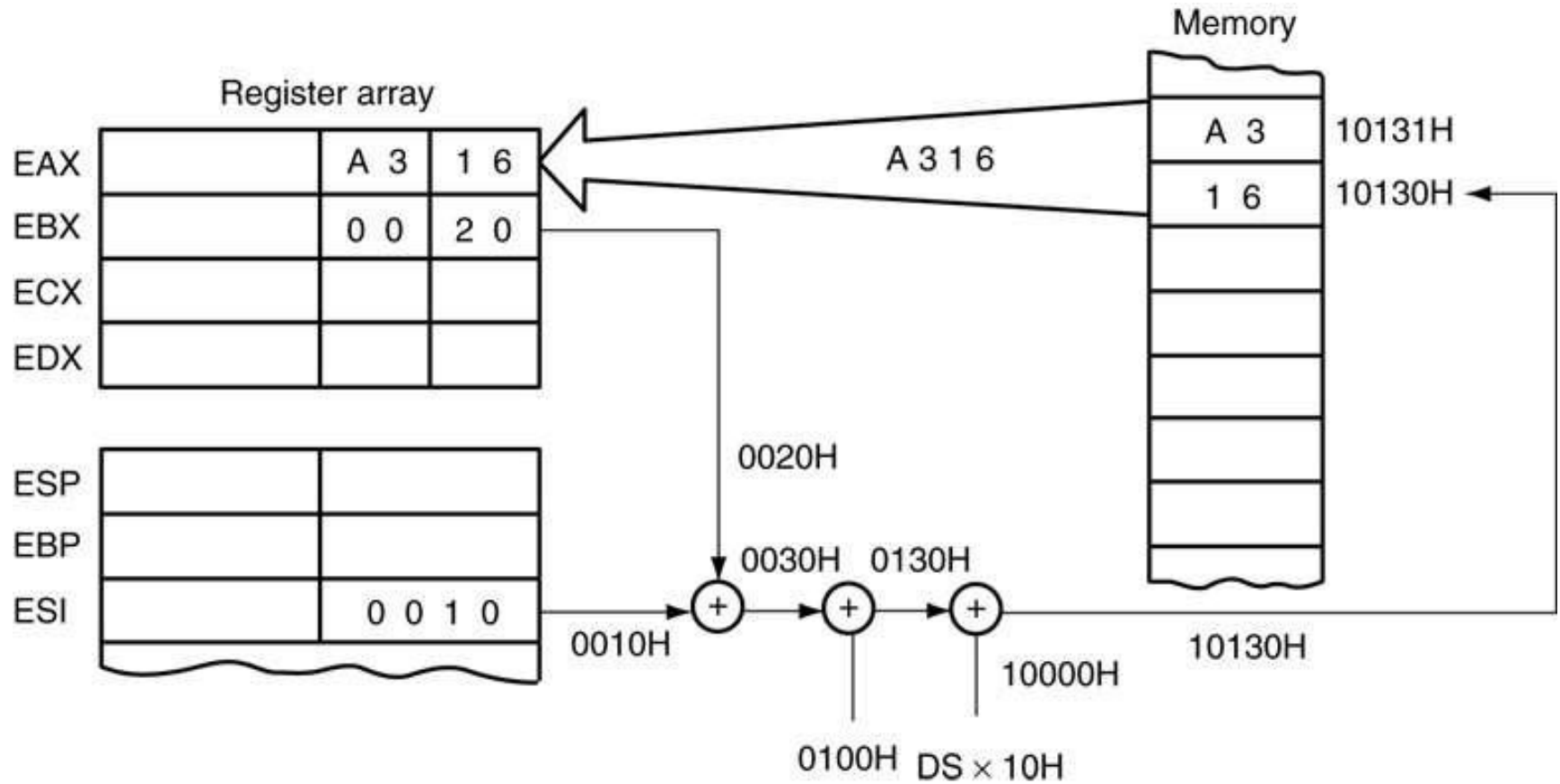


- Similar to base-plus-index addressing.
 - adds a displacement
 - uses a base register and an index register to form the memory address
- This type of addressing mode often addresses a two-dimensional array of memory data.

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.

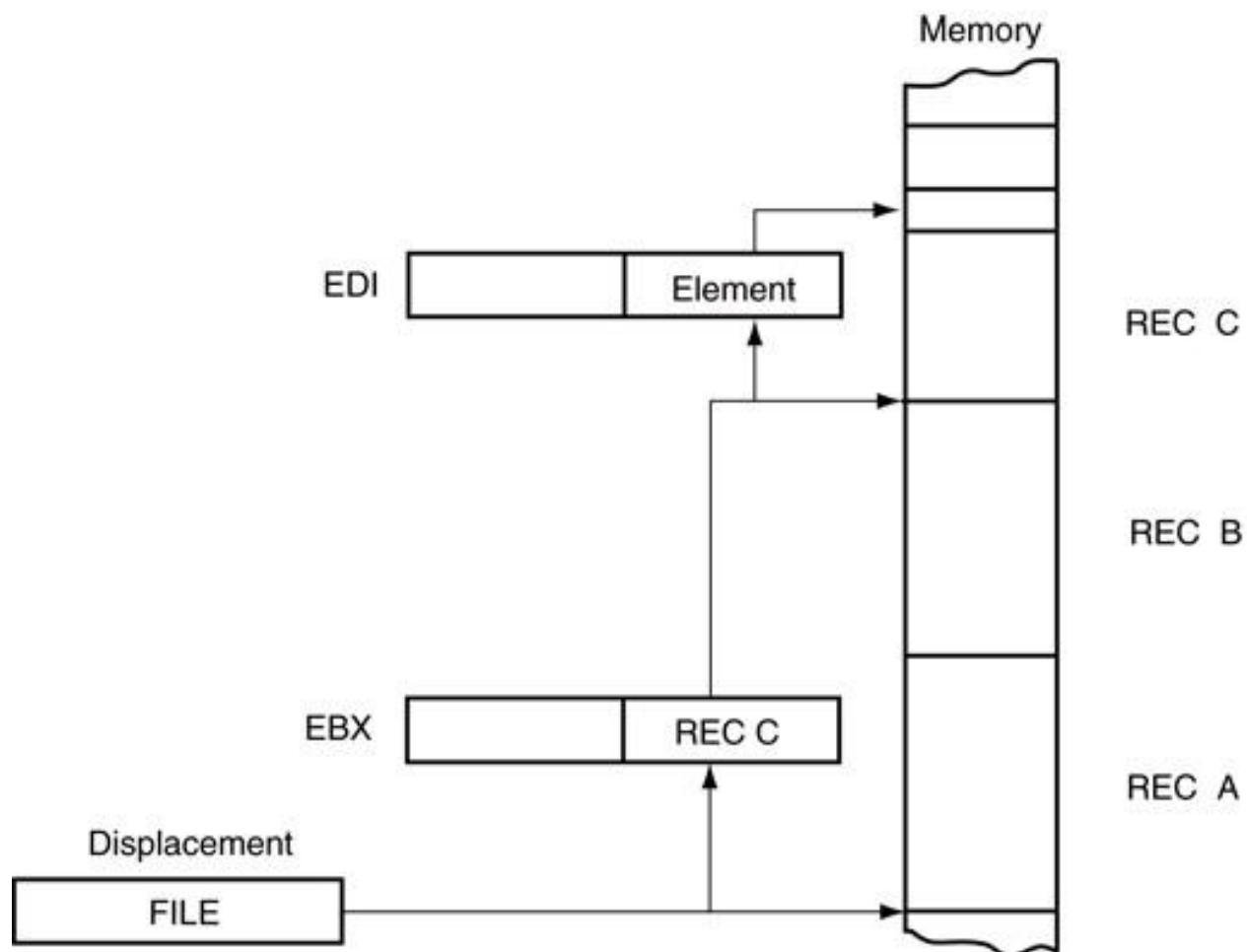
Figure 3–12 An example of base relative-plus-index addressing using a MOV AX,[BX+SI+100H] instruction. Note: DS=1000H



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).



Scaled-Index Addressing

Effective Address = Base + (Scale x Index) + Disp



- 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 1x is implied and need not be included, e.g., MOV AL,[EBX + ECX].
- Displacement is optional.

TABLE 3–9 Examples of scaled-index addressing

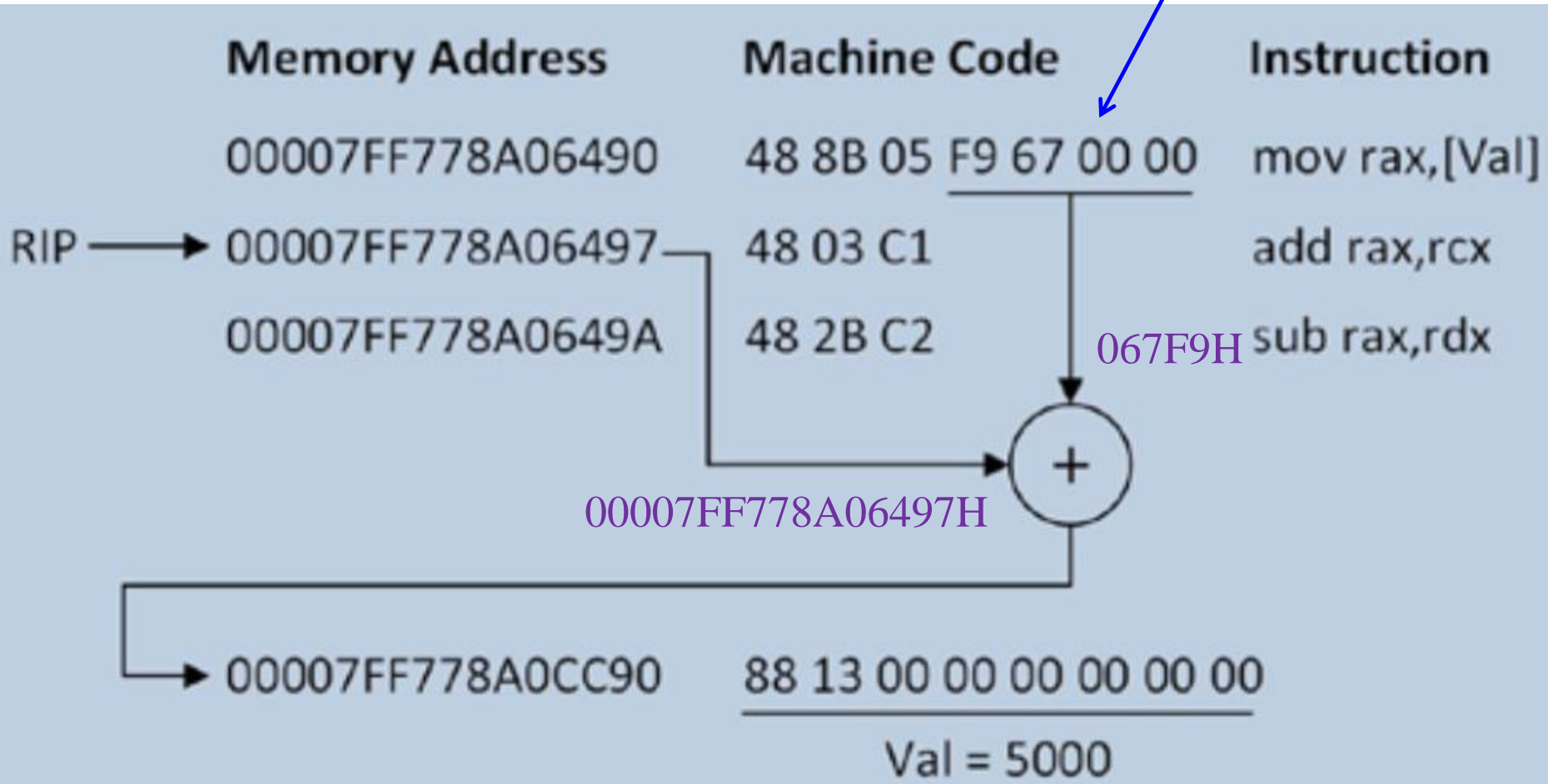
<i>Assembly Language</i>	<i>Size</i>	<i>Operation</i>
MOV EAX,[EBX+4*ECX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by the sum of 4 times ECX plus EBX into EAX
MOV [EAX+2*EDI+100H],CX	16 bits	Copies CX into the data segment memory location addressed by the sum of EAX, 100H, and 2 times EDI
MOV AL,[EBP+2*EDI+2]	8 bits	Copies the byte contents of the stack segment memory location addressed by the sum of EBP, 2, and 2 times EDI into AL
MOV EAX,ARRAY[4*ECX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by the sum of ARRAY and 4 times ECX into EAX

RIP Relative Addressing

- 64-bit mode supports data addressing relative to the 64-bit instruction pointer (RIP) to address a linear location in the flat memory model.
- The legacy x86 architecture supports IP-relative addressing only in control transfer instructions.
- RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP.

Example RIP-relative effective address calculation

32-bit displacement



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** (or **STRUCT**) directive and the end with the **ENDS** statement.
- Syntax

```
name STRUC [alignment] [, NONUNIQUE]
field-declarations
name ENDS
```

EXAMPLE 3–12 define five fields of information

structure name

```
INFO      STRUC

NAMES     DB    32 dup(?)    ;reserve 32 bytes for a name
STREET    DB    32 dup(?)    ;reserve 32 bytes for the street address
CITY      DB    16 dup(?)    ;reserve 16 bytes for the city
STATE     DB     2 dup(?)    ;reserve 2 bytes for the state
ZIP       DB     5 dup(?)    ;reserve 5 bytes for the zipcode

INFO      ENDS

NAME1     INFO <'Bob Smith', '123 Main Street', 'Wanda', 'OH', '44444'>
NAME2     INFO <'Steve Doe', '222 Moose Lane', 'Miller', 'PA', '18100'>
NAME3     INFO <'Jim Dover', '303 Main Street', 'Orender', 'CA', '90000'>
```

field-declarations

Use structure 3 times

EXAMPLE 3–13 field initialization for a structure

```
;clear NAMES in array NAME1  
;
```

```
    MOV    CX,32
```

```
    MOV    AL,0
```

```
    MOV    DI,OFFSET NAME1.NAMES
```

```
    REP    STOSB    → MOV ES:[DI],AL
```

```
;
```

```
;clear STREET in array NAME2
```

```
;
```

```
    MOV    CX,32
```

```
    MOV    AL,0
```

```
    MOV    DI,OFFSET NAME2.STREET
```

```
    REP    STOSB
```

```
;
```

```
;clear ZIP in NAME3
```

```
;
```

```
    MOV    CX,5
```

```
    MOV    AL,0
```

```
    MOV    DI,OFFSET NAME3.ZIP
```

```
    REP    STOSB
```

refer to the NAMES field
in structure NAME1

refer to the STREET field
in structure NAME2

refer to the ZIP field in
structure NAME3

Canonical Addressing/Canonical Address Form

- In 64-bit mode, an address is considered to be in **canonical form** if address bits 63 through to the most-significant implemented bit by the microarchitecture are set to either all ones or all zeros.
- If a linear-memory reference that is not in canonical form will generate a general-protection exception (#GP).

Canonical Addressing/Canonical Address Form

- By checking canonical-address form, the architecture prevents software from exploiting unused high bits of pointers for other purposes.
- Software complying with canonical-address form on a specific processor implementation can run unchanged on long-mode implementations supporting larger virtual-address spaces.

Summary—Instruction Operands

Table three types of operands: immediate, register, and memory

Type	Example	Equivalent C/C++ Statement
Immediate	<code>mov eax,42</code>	<code>eax = 42</code>
	<code>imul ebx,11h</code>	<code>ebx *= 0x11</code>
	<code>xor dl,55h</code>	<code>dl ^= 0x55</code>
	<code>add esi,8</code>	<code>esi += 8</code>
Register	<code>mov eax,ebx</code>	<code>eax = ebx</code>
	<code>inc ecx</code>	<code>ecx += 1</code>
	<code>add ebx,esi</code>	<code>ebx += esi</code>
	<code>mul ebx</code>	<code>edx:eax = eax * ebx</code>
Memory	<code>mov eax,[ebx]</code>	<code>eax = *ebx</code>
	<code>add eax,[val1]</code>	<code>eax += *val1</code>
	<code>or ecx,[ebx+esi]</code>	<code>ecx = *(ebx + esi)</code>
	<code>sub word ptr [edi],12</code>	<code>*(short*)edi -= 12</code>

Summary—Memory Addressing Modes

Table Memory Operand Addressing Forms

$$\text{EffectiveAddress} = \text{BaseReg} + \text{IndexReg} * \text{ScaleFactor} + \text{Disp}$$

Addressing Form	Example
RIP + Disp	<code>mov rax,[Val]</code>
BaseReg	<code>mov rax,[rbx]</code>
BaseReg + Disp	<code>mov rax,[rbx+16]</code>
IndexReg * SF + Disp	<code>mov rax,[r15*8+48]</code>
BaseReg + IndexReg	<code>mov rax,[rbx+r15]</code>
BaseReg + IndexReg + Disp	<code>mov rax,[rbx+r15+32]</code>
BaseReg + IndexReg * SF	<code>mov rax,[rbx+r15*8]</code>
BaseReg + IndexReg * SF + Disp	<code>mov rax,[rbx+r15*8+64]</code>

3–2 PROGRAM MEMORY- ADDRESSING MODES

- Used with the JMP (jump) and CALL instructions.
- The destination (target) operand specifies the address of the instruction being jumped to.
- This operand can be an immediate value (direct), a general-purpose register (indirect), or a memory location (indirect).
- Consist of three distinct forms:
 - direct, relative, and indirect

PROGRAM MEMORY-ADDRESSING MODES

- Two different types of jump offset:
 - **absolute offset**: that is an offset from the base of the code segment.
 - **relative offset**: a signed displacement relative to the current value of the instruction pointer in the EIP register.
- An absolute offset is specified indirectly in a general-purpose register or a memory location.
- A relative offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a immediate value. This value is added to the value in the EIP register.

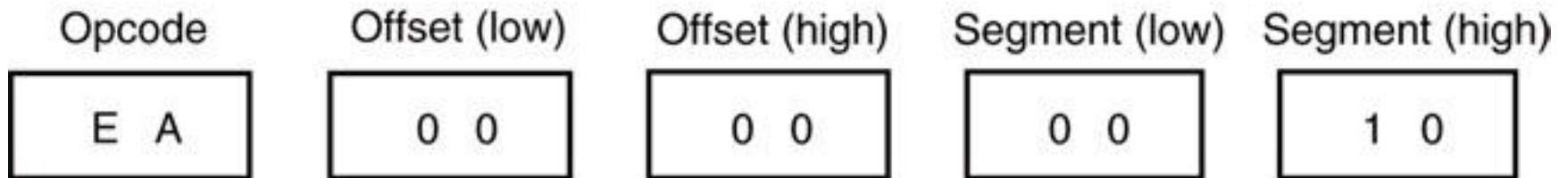
PROGRAM MEMORY-ADDRESSING MODES

- Four different types of jumps:
 - **Short jump**—A near jump where the jump range is limited to -128 to $+127$ from the current EIP value.
 - **Near jump**—A jump to an instruction within the current code segment (the segment pointed to by the CS register), referred to as an intrasegment jump.
 - **Far jump**—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, referred to as an intersegment jump.
 - **Task switch**—A jump to an instruction located in a different task (in protected mode only).

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 This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction.



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

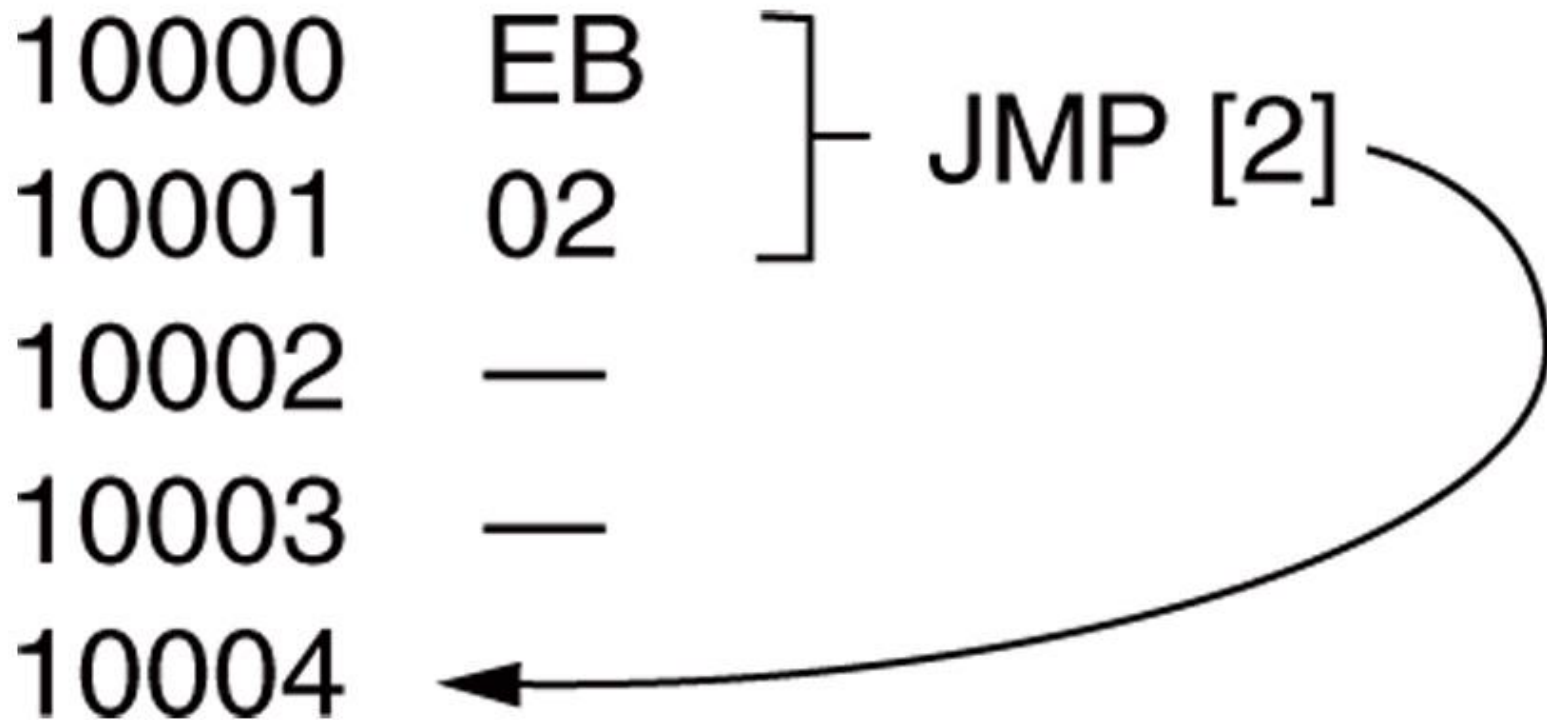
- This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction.
 - an **intersegment jump** is a jump to any memory location within the entire memory system
- Often called a *far jump* because it can jump to any memory location for the next instruction.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

- 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.
- A 1-byte displacement is used in short jumps, and a 2-byte displacement is used with near jumps and calls. Both types are considered to be intrasegment jumps.

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*

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 shows a **jump table** that is stored, beginning at memory location TABLE. The exact address chosen from the TABLE is determined by an index stored with the jump instruction.

This is a jump table that stores addresses of various programs.

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

```
;Using indirect addressing for a jump
;
MOV  BX, 4           ;address LOC2
JMP  TABLE[BX]      ;jump to LOC2
```

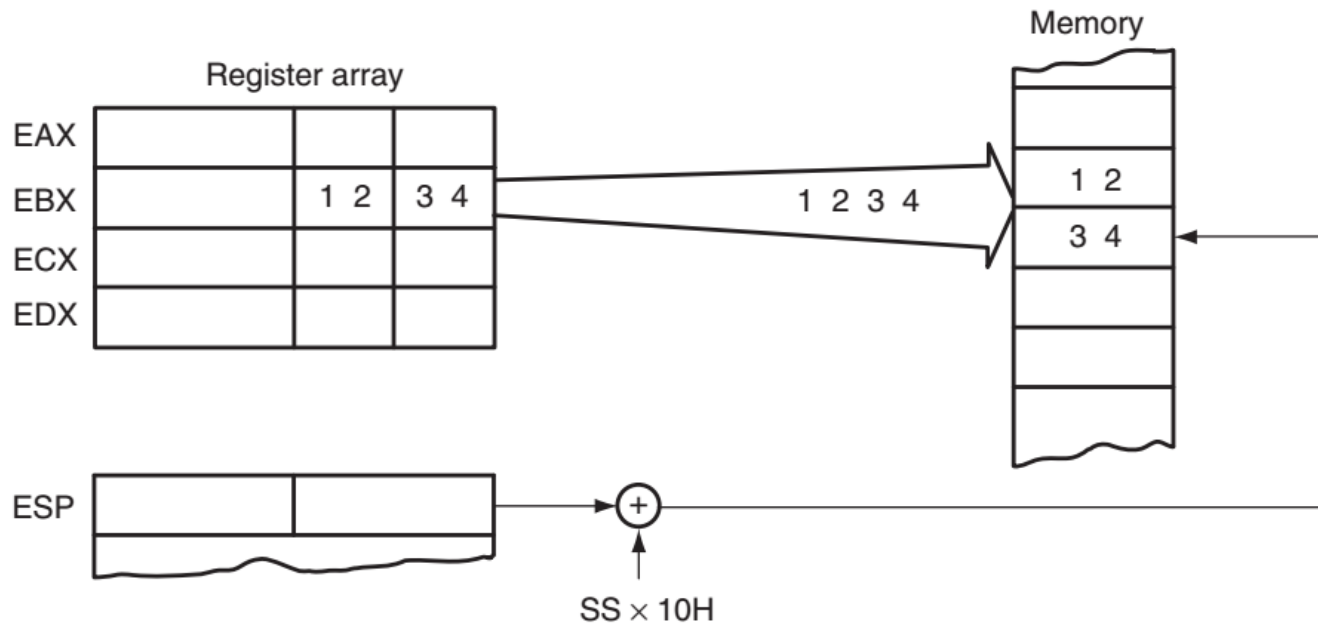
The jump table is referenced by the program

3–3 STACK MEMORY-ADDRESSING MODES

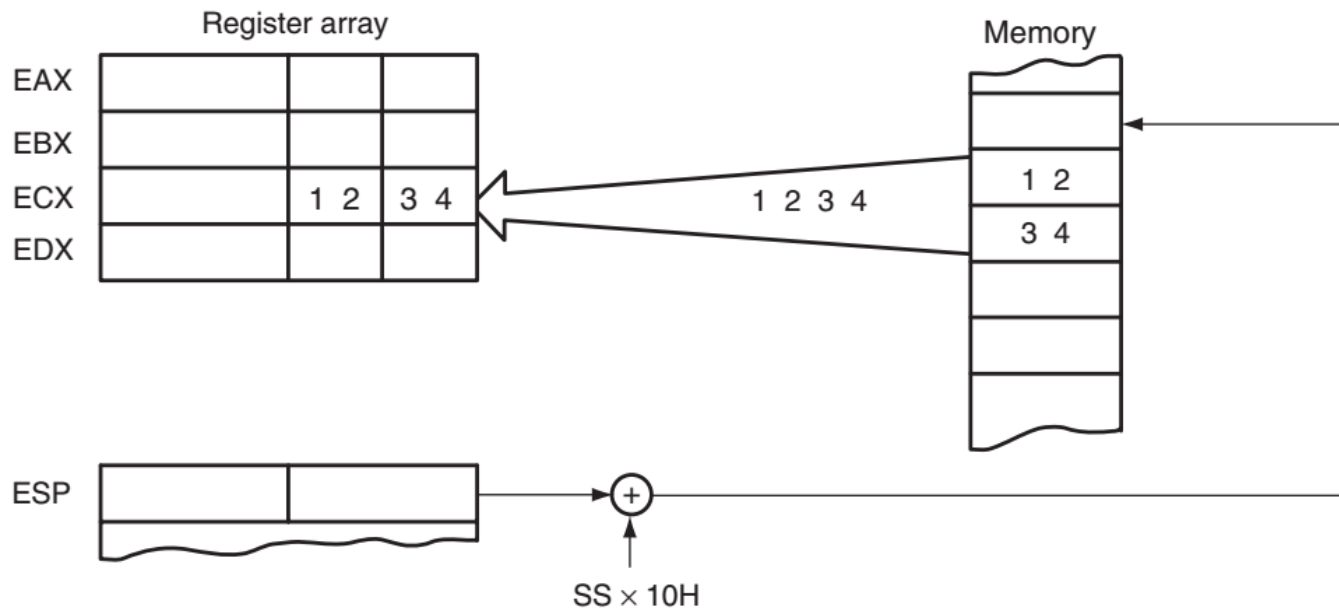
- The stack plays an important role in all microprocessors.
 - holds data temporarily and stores return addresses used by procedures
- Stack memory is LIFO (**last-in, first-out**) memory
 - describes the way data are stored and removed from the stack

- 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.
- 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



PUSH BX places the contents of BX onto the stack



POP CX removes data from the stack and places them into CX

- Note that PUSH and POP store or retrieve **words of data—never bytes**—in 8086 - 80286.
- 80386 and above allow **words, doublewords** or **quadwords** 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.

SUMMARY

- The MOV instruction copies the contents of the source operand into the destination operand.
- The second source never changes for any instruction.
- Register addressing specifies any 8-bit register (AH, AL, BH, BL, CH, CL, DH, or DL) or any 16-bit register (AX, BX, CX, DX, SP, BP, SI, or DI).

SUMMARY

(cont.)

- The segment registers (CS, DS, ES, or SS) are also addressable for moving data between a segment register and a 16-bit register/memory location or for PUSH and POP.
- In the 80386 through the Core2 microprocessors, the extended registers also are used for register addressing; they consist of EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.

SUMMARY

(*cont.*)

- In the 64-bit mode, the registers are RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15.
- The MOV immediate instruction transfers the byte or word that immediately follows the opcode into a register or a memory location.
- Immediate addressing manipulates constant data in a program.

SUMMARY

(*cont.*)

- In the 80386 and above, doubleword immediate data may also be loaded into a 32-bit register or memory location.
- The `.MODEL` statement is used with assembly language to identify the start of a file and the type of memory model used with the file.
- If the size is `TINY`, the program exists in the code segment, and assembled as a command (`.COM`) program.

SUMMARY

(*cont.*)

- If the SMALL model is used, the program uses a code and data segment and assembles as an executable (.EXE) program.
- Direct addressing occurs in two forms in the microprocessor: direct addressing and displacement addressing.

SUMMARY

(*cont.*)

- Both forms of addressing are identical except that direct addressing is used to transfer data between EAX, AX, or AL and memory; displacement addressing is used with any register-memory transfer.
- Direct addressing requires 3 bytes of memory, whereas displacement addressing requires 4 bytes.

SUMMARY

(*cont.*)

- Register indirect addressing allows data to be addressed at the memory location pointed to by either a base (BP and BX) or index register (DI and SI).
- In the 80386 and above, extended registers EAX, EBX, ECX, EDX, EBP, EDI, and ESI are used to address memory data.

SUMMARY

(*cont.*)

- Base-plus-index addressing often addresses data in an array.
- The memory address for this mode is formed by adding a base register, index register, and the contents of a segment register times 10H.
- In the 80386 and above, the base and index registers may be any 32-bit register except EIP and ESP.

SUMMARY

(*cont.*)

- Register relative addressing uses a base or index register, plus a displacement to access memory data.
- Base relative-plus-index addressing is useful for addressing a two-dimensional memory array.
- The address is formed by adding a base register, an index register, displacement, and the contents of a segment register times 10H.

SUMMARY

(*cont.*)

- Scaled-index addressing is unique to the 80386 through the Core2.
- The second of two registers (index) is scaled by a factor of 2 to access words, doublewords, or quadwords in memory arrays.
- The `MOV AX,[EBX + 2*ECX]` and the `MOV [4 * ECX],EDX` are examples of scaled-index instructions.

SUMMARY

(*cont.*)

- Data structures are templates for storing arrays of data and are addressed by array name and field.
- Direct program memory addressing is allowed with the JMP and CALL instructions to any location in the memory system.
- With this addressing mode, the off-set address and segment address are stored with the instruction.

SUMMARY

(*cont.*)

- Relative program addressing allows a JMP or CALL instruction to branch for-ward or backward in the current code segment by bytes.
- In the 80386 and above, the 32-bit displacement allows a branch to any location in the current code segment by using a displacement value of bytes.
- The 32-bit displacement can be used only in protected mode.

SUMMARY

(*cont.*)

- Indirect program addressing allows the JMP or CALL instructions to address another portion of the program or subroutine indirectly through a register or memory location.
- The PUSH and POP instructions transfer a word between the stack and a register or memory location.
- A PUSH immediate instruction is available to place immediate data on the stack.

SUMMARY

- The PUSHA and POPA instructions transfer AX, CX, DX, BX, BP, SP, SI, and DI between the stack and these registers.
- In 80386 and above, the extended register and extended flags can also be transferred between registers and the stack.
- A PUSHFD stores the EFLAGS, whereas a PUSHF stores the FLAGS. POPA and PUSHA are not available in 64-bit mode.

Assignment

3-10, 23, 28-32, 35, 36, 45, 48, 49, 53