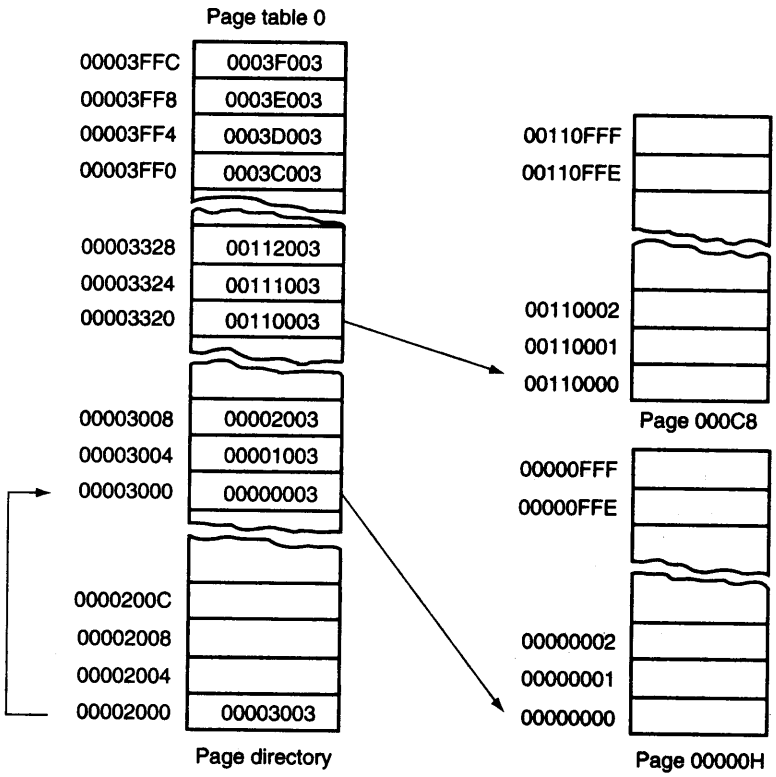


FIGURE 2–14 The page directory, page table 0, and two memory pages. Note how the address of page 000C8000–000C9000 has been moved to 00110000–00110FFF.

the Pentium and Pentium Pro microprocessors, pages can be either 4K bytes in length or 4M bytes in length. Although no software currently supports the 4M-byte pages, as the Pentium 4 and more advanced versions pervade the personal computer, operating systems of the future will undoubtedly begin to support 4M-byte memory pages.

2–5 SUMMARY

1. The programming model of the 8086 through 80286 contains 8- and 16-bit registers. The programming model of the 80386 and above contains 8-, 16-, and 32-bit extended registers as well as two additional 16-bit segment registers: FS and GS.
2. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL. The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI. The segment registers are CS, DS, ES, SS, FS, and GS. The 32-bit extended registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. In addition, the microprocessor contains an instruction pointer (IP/EIP) and flag register (FLAGS or EFLAGS).
3. All real mode memory addresses are a combination of a segment address plus an offset address. The starting location of a segment is defined by the 16-bit number in the segment register that is appended with a hexadecimal zero at its rightmost end. The offset address is a 16-bit number added to the 20-bit segment address to form the real mode memory address.
4. All instructions (code) are accessed by the combination of CS (segment address) plus IP or EIP (offset address).

5. Data are normally referenced through a combination of the DS (data segment), and either an offset address or the contents of a register that contains the offset address. The 8086 through the Pentium 4 use BX, DI, and SI as default offset registers for data if 16-bit registers are selected. The 80386 and above can use the 32-bit registers EAX, EBX, ECX, EDX, EDI, and ESI as default offset registers for data.
6. Protected mode operation allows memory above the first 1M byte to be accessed by the 80286 through the Pentium 4 microprocessors. This extended memory system (XMS) is accessed via a segment address plus an offset address, just as in the real mode. The difference is that the segment address is not held in the segment register. In the protected mode, the segment starting address is stored in a descriptor that is selected by the segment register.
7. A protected mode descriptor contains a base address, limit, and access rights byte. The base address locates the starting address of the memory segment; the limit defines the last location of the segment. The access rights byte defines how the memory segment is accessed via a program. The 80286 microprocessor allows a memory segment to start at any of its 16M bytes of memory using a 24-bit base address. The 80386 and above allow a memory segment to begin at any of its 4G bytes of memory using a 32-bit base address. The limit is a 16-bit number in the 80286 and a 20-bit number in the 80386 and above. This allows an 80286 memory segment limit of 64K bytes, and an 80386 and above memory segment limit of either 1M bytes (G = 0) or 4G bytes (G = 1).
8. The segment register contains three fields of information in the protected mode. The leftmost 13 bits of the segment register address one of 8192 descriptors from a descriptor table. The TI bit accesses either the global descriptor table (TI = 0) or the local descriptor table (TI = 1). The rightmost 2 bits of the segment register select the requested priority level for the memory segment access.
9. The program-invisible registers are used by the 80286 and above to access the descriptor tables. Each segment register contains a cache portion that is used in protected mode to hold the base address, limit, and access rights acquired from a descriptor. The cache allows the microprocessor to access the memory segment without again referring to the descriptor table until the segment register's contents are changed.
10. A memory page is 4K bytes in length. The linear address, as generated by a program, can be mapped to any physical address through the paging mechanism found within the 80386 through the Pentium 4 microprocessor.
11. Memory paging is accomplished through control registers CR0 and CR3. The PG bit of CR0 enables paging and the contents of CR3 addresses the page directory. The page directory contains up to 1024 page table addresses that are used to access paging tables. The page table contains 1024 entries that locate the physical address of a 4K-byte memory page.
12. The TLB (translation look-aside buffer) caches the 32 most recent page table translations. This precludes page table translation if the translation resides in the TLB, speeding the execution of software.

2–6 QUESTIONS AND PROBLEMS

1. What are program-visible registers?
2. The 80286 addresses registers that are 8- and _____ - bits wide.
3. The extended registers are addressable by which microprocessors?
4. The extended BX register is addressed as _____
5. Which register holds a count for some instructions?
6. What is the purpose of the IP/EIP register?
7. The carry flag bit is set by which arithmetic operations?
8. Will an overflow occur if a signed FFH is added to a signed 01H?
9. A number that contains 3 one bits is said to have _____ parity.
10. Which flag bit controls the INTR pin on the microprocessor?

11. Which microprocessors contain an FS segment register?
12. What is the purpose of a segment register in the real mode operation of the microprocessor?
13. In the real mode, show the starting and ending addresses of each segment located by the following segment register values:
    (a) 1000H
    (b) 1234H
    (c) 2300H
    (d) E000H
    (e) AB00H
14. Find the memory address of the next instruction executed by the microprocessor, when operated in the real mode, for the following CS:IP combinations:
    (a) CS = 1000H and IP = 2000H
    (b) CS = 2000H and IP = 1000H
    (c) CS = 2300H and IP = 1A00H
    (d) CS = 1A00H and IP = B000H
    (e) CS = 3456H and IP = ABCDH
15. Real mode memory addresses allow access to memory below which memory address?
16. Which register or registers are used as an offset address for string instruction destinations in the microprocessor?
17. Which 32-bit register or registers are used as an offset address for data segment data in the Pentium 4 microprocessor?
18. The stack memory is addressed by a combination of the _____ segment plus _____ offset.
19. If the base pointer (BP) addresses memory, the _____ segment contains the data.
20. Determine the memory location addressed by the following real mode 80286 register combinations:
    (a) DS = 1000H and DI = 2000H
    (b) DS = 2000H and SI = 1002H
    (c) SS = 2300H and BP = 3200H
    (d) DS = A000H and BX = 1000H
    (e) SS = 2900H and SP = 3A00H
21. Determine the memory location addressed by the following real mode Pentium 4 register combinations:
    (a) DS = 2000H and EAX = 00003000H
    (b) DS = 1A00H and ECX = 00002000H
    (c) DS = C000H and ESI = 0000A000H
    (d) SS = 8000H and ESP = 00009000H
    (e) DS = 1239H and EDX = 0000A900H
22. Protected mode memory addressing allows access to which area of the memory in the 80286 microprocessor?
23. Protected mode memory addressing allows access to which area of the memory in the Pentium 4 microprocessor?
24. What is the purpose of the segment register in protected mode memory addressing?
25. How many descriptors are accessible in the global descriptor table in the protected mode?
26. For an 80286 descriptor that contains a base address of A00000H and a limit of 1000H, what starting and ending locations are addressed by this descriptor?
27. For an 80486 descriptor that contains a base address of 01000000H, a limit of 0FFFFH, and G = 0, what starting and ending locations are addressed by this descriptor?
28. For a Pentium 4 descriptor that contains a base address of 00280000H, a limit of 00010H, and G = 1, what starting and ending locations are addressed by this descriptor?
29. If the DS register contains 0020H in a protected mode system, which global descriptor table entry is accessed?
30. If DS = 0103H in a protected mode system, the requested privilege level is _____.
31. If DS = 0105H in a protected mode system, which entry, table, and requested privilege level are selected?
32. What is the maximum length of the global descriptor table in the Pentium 4 microprocessor?

33. Code a descriptor that describes a memory segment that begins at location 210000H and ends at location 21001FH. This memory segment is a code segment that can be read. The descriptor is for an 80286 microprocessor.
34. Code a descriptor that describes a memory segment that begins at location 03000000H and ends at location 05FFFFFFH. This memory segment is a data segment that grows upward in the memory system and can be written. The descriptor is for an 80386 microprocessor.
35. Which register locates the global descriptor table?
36. How is the local descriptor table addressed in the memory system?
37. Describe what happens when a new number is loaded into a segment register when the microprocessor is operated in the protected mode.
38. What are the program-invisible registers?
39. What is the purpose of the GDTR?
40. How many bytes are found in a memory page?
41. What register is used to enable the paging mechanism in the 80386, 80486, Pentium, Pentium Pro, and Pentium 4 microprocessors?
42. How many 32-bit addresses are stored in the page directory?
43. Each entry in the page directory translates how much linear memory into physical memory?
44. If the microprocessor sends linear address 00200000H to the paging mechanism, which paging directory entry is accessed, and which page table entry is accessed?
45. What value is placed in the page table to redirect linear address 20000000H–30000000H?
46. What is the purpose of the TLB located within the 80486 microprocessor?
47. Using the Internet, write a short report that details the TLB. Hint: You might want to go to the Intel Web site and search for information.

CHAPTER 3

Addressing Modes

INTRODUCTION

Efficient software development for the microprocessor requires a complete familiarity with the addressing modes employed by each instruction. In this chapter, the MOV **(move data)** instruction is used to describe the data-addressing modes. The MOV instruction transfers bytes or words of data between registers, or between registers and memory in the 8086 through the 80286 and bytes, words, or doublewords in the 80386 and above. In describing the program memory-addressing modes, the CALL and JUMP instructions show how to modify the flow of the program.

The data-addressing modes include register, immediate, direct, register indirect, base-plus-index, register relative, and base relative-plus-index in the 8086 through the 80286 microprocessor. The 80386 and above also include a scaled-index mode of addressing memory data. The program memory-addressing modes include program relative, direct, and indirect. The operation of the stack memory is explained so that the PUSH and POP instructions are understood.

CHAPTER OBJECTIVES

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

1. Explain the operation of each data-addressing mode.
2. Use the data-addressing modes to form assembly language statements.
3. Explain the operation of each program memory-addressing mode.
4. Use the program memory-addressing modes to form assembly and machine language statements.
5. Select the appropriate addressing mode to accomplish a given task.
6. Detail the difference between addressing memory data using real mode and protected mode operation.
7. Describe the sequence of events that place data onto the stack or remove data from the stack.
8. Explain how a data structure is placed in memory and used with software.

3–1 DATA-ADDRESSING MODES

Because the MOV instruction is a common and flexible instruction, it provides a basis for the explanation of the data-addressing modes. Figure 3–1 illustrates the MOV instruction and defines the direction of data flow. The **source** is to the right and the **destination** is to the left, next to the opcode MOV. (An **opcode,** or operation code, tells the microprocessor which operation to perform.) This direction of flow, which is applied to all instructions, is awkward at first. We naturally assume that things move from left to right, whereas here they move from right to left. Notice that a comma always separates the destination from the source in an instruction. Also, note that memory-to-memory transfers are not allowed by any instruction except for the MOVS instruction.

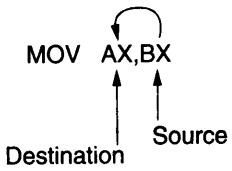


FIGURE 3–1 The MOV instruction showing the source, destination, and direction of data flow.

In Figure 3–1, the MOV AX,BX instruction transfers the word contents of the source register (BX) into the destination register (AX). The source never changes, but the destination usually changes.[1] It is essential to remember that a MOV instruction always *copies* the source data and into the destination. The MOV never actually picks up the data and moves it. Also, note that the flag register remains unaffected by most data transfer instructions. The source and destination are often called **operands.**

Figure 3–2 shows all possible variations of the data-addressing modes using the MOV instruction. This illustration helps to show how each data-addressing mode is formulated with the MOV instruction and also serves as a reference. Note that these are the same data-addressing modes found with all versions of the Intel microprocessor, except for the scaled-index-addressing mode, which is found only in the 80386 through the Pentium 4. The data-addressing modes are as follows:

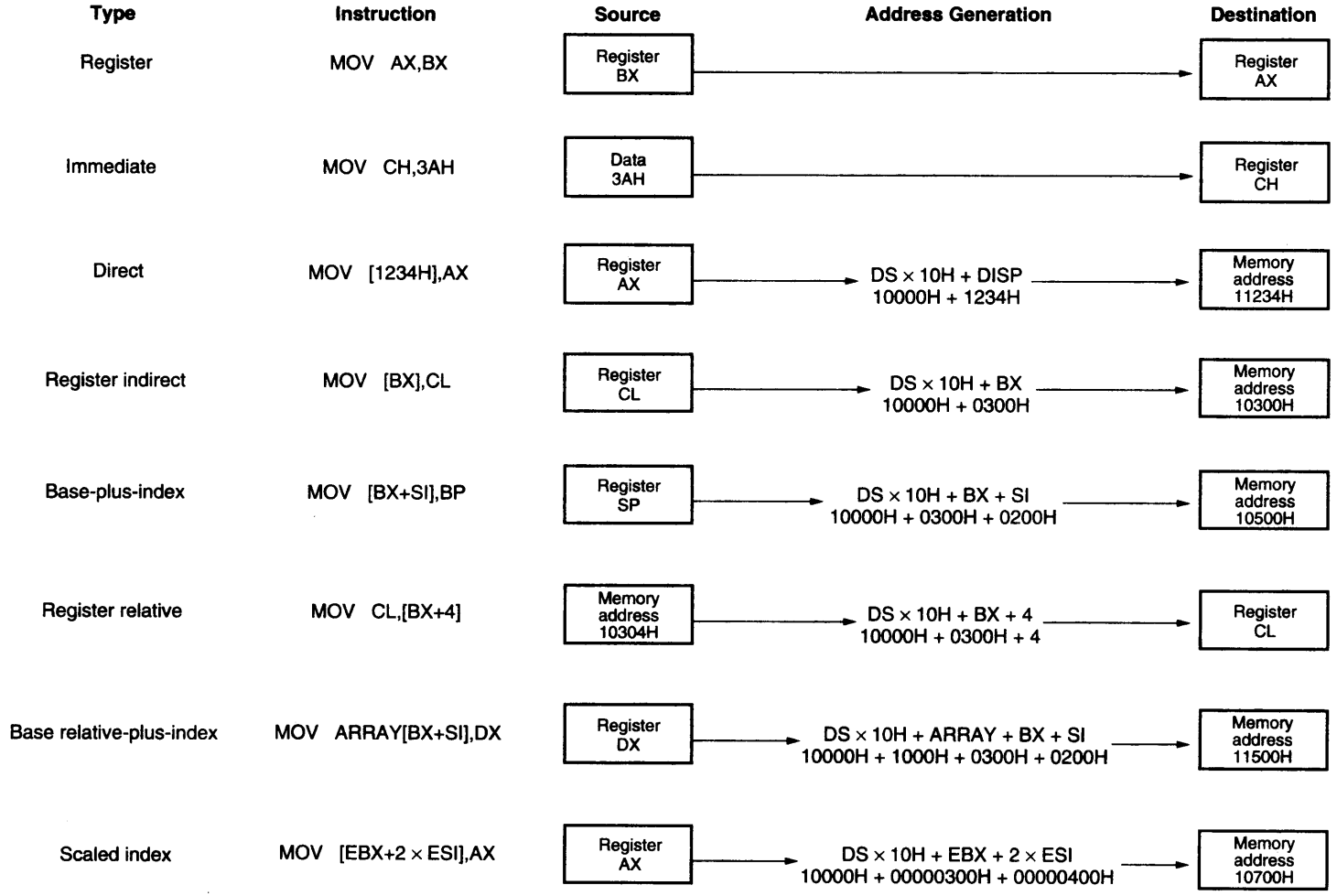
Register addressing — Transfers a copy of a byte or word from the source register or memory location to the destination register or memory location. (Example: the MOV CX,DX instruction copies the word-sized contents of register DX into register CX.) In the 80386 and above, a doubleword can be transferred from the source register or memory location to the destination register or memory location. (Example: the MOV ECX,EDX instruction copies the doubleword-sized contents of register EDX into register ECX.)

Immediate addressing — Transfers the source-immediate byte or word of data into the destination register or memory location. (Example: the MOV AL,22H instruction copies a byte-sized 22H into register AL.) In the 80386 and above, a doubleword of immediate data can be transferred into a register or memory location. (Example: the MOV EBX,12345678H instruction opies a doubleword-sized 12345678H into the 32-bit wide EBX register.)

Direct addressing — Moves a byte or word between a memory location and a register. The instruction set does not support a memory-to-memory transfer, except for the MOVS instruction. (Example: the MOV CX,LIST instruction copies the word-sized contents of memory location LIST into register CX.) In the 80386 and above, a doubleword-sized memory location can also be addressed. (Example: the MOV ESI,LIST instruction copies a 32-bit number, stored in four consecutive bytes of memory, from location LIST into register ESI.)

Register indirect addressing — Transfers a byte or word between a register and a memory location addressed by an index or base register. The index and base registers are BP, BX, DI, and SI. (Example: the MOV AX,[BX] instruction copies the word-sized data from the data segment offset address indexed by BX into register AX.) In the 80386 and above, a

[1]The exceptions are the CMP and TEST instructions, which never change the destination. These instructions are described in later chapters.



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

FIGURE 3–2 8086–Pentium 4 data-addressing modes.

byte, word, or double-word is transferred between a register and a memory location addressed by any register: EAX, EBX, ECX, EDX, EBP, EDI, or ESI. (Example: the MOV AL,[ECX] instruction loads AL from the data segment offset address selected by the contents of ECX.)

Base-plus-index addressing Transfers a byte or word between a register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI). (Example: the MOV [BX+DI],CL instruction copies the byte-sized contents of register CL into the data segment memory location addressed by BX plus DI.) In the 80386 and above, any register EAX, EBX, ECX, EDX, EBP, EDI, or ESI may be combined to generate the memory address. (Example: the MOV [EAX+EBX],CL instruction copies the byte-sized contents of register CL into the data segment memory location addressed by EAX plus EBX.)

Register relative addressing Moves a byte or word between a register and the memory location addressed by an index or base register plus a displacement. (Example: MOV AX,[BX+4] or MOV AX,ARRAY[BX]. The first instruction loads AX from the data segment address formed by BX plus 4. The second instruction loads AX from the data segment memory location in ARRAY plus the contents of BX.) The 80386 and above use any register to address memory. (Example: MOV AX,[ECX+4] or MOV AX,ARRAY[EBX]. The first instruction loads AX from the data segment address formed by ECX plus 4. The second instruction loads AX from the data segment memory location ARRAY plus the contents of EBX.)

Base relative-plus-index addressing Transfers a byte or word between a register and the memory location addressed by a base and an index register plus a displacement. (Example: MOV AX,ARRAY[BX+DI] or MOV AX,[BX+DI+4]. These instructions load AX from a data segment memory location. The first instruction uses an address formed by adding ARRAY, BX, and DI and the second by adding BX, DI, and 4.) In the 80386 and above, MOV EAX,ARRAY[EBX+ECX] loads EAX from the data segment memory location accessed by the sum of ARRAY, EBX, and ECX.

Scaled-index addressing Is available only in the 80386 through the Pentium 4 microprocessor. The second register of a pair of registers is modified by the scale factor of 2X, 4X, or 8X to generate the operand memory address. (Example: a MOV EDX,[EAX+4*EBX] instruction loads EDX from the data segment memory location addressed by EAX plus 4 times EBX.) Scaling allows access to word (2X), doubleword (4X), or quadword (8X) memory array data. Note that a scaling factor of 1X also exists, but it is normally implied and does not appear in the instruction. The MOV AL,[EBX+ECX] is an example in which the scaling factor is a one. Alternately, the instruction can be rewritten as MOV AL,[EBX+1*ECX]. Another example is a MOV AL,[2*EBX] instruction, which uses only one scaled register to address memory.

Register Addressing

Register addressing is the most common form of data addressing and, once the register names are learned, is the easiest to apply. The microprocessor contains the following 8-bit registers used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL. Also present are the following 16-bit registers: AX, BX, CX, DX, SP, BP, SI, and DI. In the 80386 and above, the extended 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. With register addressing, some MOV instructions, and the PUSH and POP instructions, also use the 16-bit segment registers (CS, ES, DS, SS, FS, and GS). It is important for instructions to use registers that are the same size. *Never* mix an 8-bit register with a 16-bit register, an 8-bit register with a 32-bit register, or a 16-bit register with 32-bit register because this is not allowed by the microprocessor and results in an error when assembled. This is

TABLE 3–1 Examples of the register-addressed instructions.

Assembly Language	*Size*	*Operation*
MOV AL,BL	8-bits	Copies BL into AL
MOV CH,CL	8-bits	Copies CL into CH
MOV AX,CX	16-bits	Copies CX into AX
MOV SP,BP	16-bits	Copies BP into SP
MOV DS,AX	16-bits	Copies AX into DS
MOV SI,DI	16-bits	Copies DI into SI
MOV BX,ES	16-bits	Copies ES into BX
MOV ECX,EBX	32-bits	Copies EBX into ECX
MOV ESP,EDX	32-bits	Copies EDX into ESP
MOV ES,DS	—	Not allowed (segment-to-segment)
MOV BL,DX	—	Not allowed (mixed sizes)
MOV CS,AX	—	Not allowed (the code segment register may not be the destination register)

even true when a MOV AX,AL or a MOV EAX,AL instruction may seem to make sense. Of course, the MOV AX,AL or MOV EAX,AL instruction is *not* allowed because these registers are of different sizes. Note that a few instructions, such as SHL DX,CL, are exceptions to this rule, as indicated in later chapters. It is also important to note that *none* of the MOV instructions affect the flag bits.

Table 3–1 shows many variations of register move instructions. It is impossible to show all combinations because there are too many. For example, just the 8-bit subset of the MOV instruction has 64 different variations. A segment-to-segment register MOV instruction is about the only type of register MOV instruction *not* allowed. Note that the code segment register is not normally changed by a MOV instruction because the address of the next instruction is found in both IP/EIP and CS. If only CS were changed, the address of the next instruction would be unpredictable. Therefore, changing the CS register with a MOV instruction is not allowed.

Figure 3–3 shows the operation of the MOV BX,CX instruction. Note that the source register's contents do not change, but the destination register's contents do change. The instruction moves *(copies)* a 1234H from register CX into register BX. This *erases* the old contents (76AFH) of register BX, but the contents of CX remain unchanged. The contents of the destination register or destination memory location change for all instructions

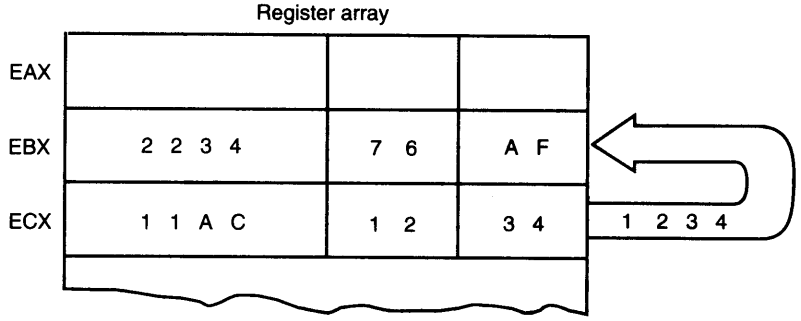


FIGURE 3–3 The effect of executing the MOV BX, CX instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.

except the CMP and TEST instructions. Note that the MOV BX,CX instruction does not affect the leftmost 16 bits of register EBX.

Example 3–1 shows a sequence of assembled instructions that copy various data between 8-, 16-, and 32-bit registers. As mentioned, the act of moving data from one register to another only changes the destination register, never the source. The last instruction in this example (MOV CS,AX) assembles without error, but causes problems if executed. If only the contents of CS change without changing IP, the next step in the program is unknown and therefore causes the program to go awry.

EXAMPLE 3–1

```
0000 8B C3              MOV AX,BX       ;copy contents of BX into AX
0002 8A CE              MOV CL,DH       ;copy the contents of DH into CL
0004 8A CD              MOV CL,CH       ;copy the contents of CH into CL
0006 66|8B C3           MOV EAX,EBX     ;copy the contents of EBX into EAX
0009 66|8B D8           MOV EBX,EAX     ;copy EAX into EBX, ECX, and EDX
000C 66|8B C8           MOV ECX,EAX
000F 66|8B D0           MOV EDX,EAX
0012 8C C8              MOV AX,CS       ;copy CS into DS
0014 8E D8              MOV DS,AX
0016 8E C8              MOV CS,AX       ;assembles, but will cause problems
```

Immediate Addressing

Another data-addressing mode is immediate addressing. The term *immediate* implies that the data immediately follow the hexadecimal opcode in the memory. Also note that immediate data are **constant data,** while the data transferred from a register are **variable data.** Immediate addressing operates upon a byte or word of data. In the 80386 through the Pentium 4 microprocessors, immediate addressing also operates on doubleword data. The MOV immediate instruction transfers a copy of the immediate data into a register or a memory location. Figure 3–4 shows the operation of a MOV EAX,13456H instruction. This instruction copies the 13456H from the instruction, located in the memory immediately following the hexadecimal opcode, into register 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.[2] The 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. In this text, the # symbol is not used for immediate data. The most common assemblers—Intel ASM, Microsoft MASM,[3] and Borland TASM[4]—do not use the # symbol for immediate data, but an older assembler used with some Hewlett-Packard logic development systems do, as may others.

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 that the data start with a 0. For example, to represent

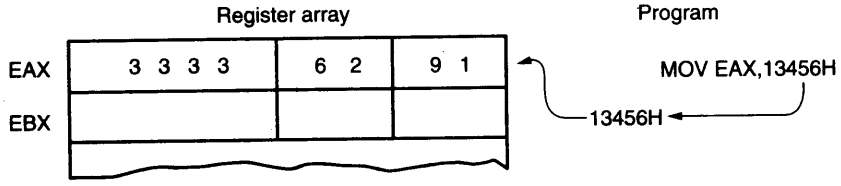


FIGURE 3–4 The operation of the MOV EAX,3456H instruction. This instruction copies the immediate data (13456H) into EAX.

[2]This is true for the assembler provided by Hewlett-Packard in some development systems.

[3]MASM (MACRO assembler) is a trademark of Microsoft Corporation.

[4]TASM (Turbo assembler) is a trademark of Borland Corporation.

TABLE 3–2 Examples of immediate addressing using the MOV instruction.

Assembly Language	*Size*	*Operation*
MOV BL,44	8-bits	Copies a 44 decimal (2CH) into BL
MOV AX,44H	16-bits	Copies a 0044H into AX
MOV SI,0	16-bits	Copies a 0000H into SI
MOV CH,100	8-bits	Copies a 100 decimal (64H) into CH
MOV AL,'A'	8-bits	Copies an ASCII A into AL
MOV AX,'AB'	16-bits	Copies an ASCII BA* into AX
MOV CL,11001110B	8-bits	Copies a 11001110 binary into CL
MOV EBX,12340000H	32-bits	Copies a 12340000H into EBX
MOV ESI,12	32-bits	Copies a 12 decimal into ESI
MOV EAX,100Y	32-bits	Copies a 100 binary into EAX

**Note:* This is not an error. The ASCII characters are stored as a BA, so care should be exercised when using a word-sized pair of ASCII characters.

a hexadecimal F2, a 0F2H is used in assembly language. In some assemblers (though not in MASM, TASM, or this text), hexadecimal data are represented with an 'h, as in MOV AX,#'h1234. 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. (An example is the MOV BH,'A' instruction, which moves an ASCII-coded A (41H) into register BH.) 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, or, in some assemblers, the letter Y. Table 3–2 shows many different variations of MOV instructions that apply immediate data.

Example 3–2 shows various immediate instructions in a short program that places a 0000H into the 16-bit registers AX, BX, and CX. This is followed by instructions that use register addressing to copy the contents of AX into registers SI, DI, and BP. This is a complete program that uses programming models for assembly and execution. The .MODEL .TINY statement directs the assembler to assemble the program into a single code segment. The .CODE statement or directive indicates the start of the code segment; the .STARTUP statement indicates the starting instruction in the program; and the .EXIT statement causes the program to exit to DOS. The END statement indicates the end of the program file. This program is assembled with MASM and executed with CodeView[5] (CV) to view its execution. Note that the most recent version of TASM will also accept MASM code. To store the program into the system use either the DOS EDIT program or Programmer's WorkBench[6] (PWB). Note that a TINY program always assembles as a command (.COM) program.

EXAMPLE 3–2

```
                        .MODEL TINY         ;choose single segment model
0000                    .CODE               ;indicate start of code segment

                        .STARTUP            ;indicate start of program

0100  B8 0000           MOV     AX,0        ;place 0000H into AX
0103  BB 0000           MOV     BX,0000H    ;place 0000H into BX
0106  B9 0000           MOV     CX,0        ;place 0000H into CX
```

[5]CodeView is a registered trademark of Microsoft Corporation.

[6]Programmer's WorkBench is a registered trademark of Microsoft Corporation.

```
0109  8B F0          MOV    SI,AX      ;copy AX into SI
010B  8B F8          MOV    DI,AX      ;copy AX into DI
010D  8B E8          MOV    BP,AX      ;copy AX into BP

                     .EXIT             ;exit to DOS
                     END               ;end of file
```

Each statement in a program consists of four parts or fields, as illustrated in Example 3–3. The leftmost field is called the *label* and it is used to store a symbolic name for the memory location that it represents. All labels must begin with a letter or one of the following special characters: @, $, _, or ?. A label may be of 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 that are explained as they appear. The next field is called the *opcode field;* it is designed to hold the instruction, or opcode. The MOV part of the move data instruction is an example of an opcode. To the right of the opcode field is the *operand field,* which contains information used by the opcode. For example, the MOV AL,BL instruction has the opcode MOV and operands AL and BL. Note that some instructions contain between zero and three operands. The final field, the *comment field,* contains a comment about an instruction or a group of instructions. A comment always begins with a semicolon (;).

EXAMPLE 3–3

```
LABEL          OPCODE   OPERAND     COMMENT

DATA1          DB       23H         ;define DATA1 as a byte of 23H
DATA2          DW       1000H       ;define DATA2 as a word of 1000H

START:         MOV      AL,BL       ;copy BL into AL
               MOV      BH,AL       ;copy AL into BH
               MOV      CX,200      ;copy 200 decimal into CX
```

When the program is assembled and the **list** (.LST) file is viewed, it appears as the program listed in Example 3–2. The hexadecimal number at the far left is the offset address of the instruction or data. This number is generated by the assembler. The number or numbers to the right of the offset address are the machine-coded instructions or data that are also generated by the assembler. For example, if the instruction MOV AX,0 appears in a file and it is assembled, it appears in offset memory location 0100 in Example 3–2. Its hexadecimal machine language form is B8 0000. The B8 is the opcode in machine language and the 0000 is the 16-bit wide data with a value of zero. When the program was written, only the MOV AX,0 was typed into the editor; the assembler generated the machine code and addresses, and stored the program in a file ending with the extension .LST. Note that all programs shown in this text are in the form generated by the assembler.

Direct Data Addressing

Most instructions can use the direct data-addressing mode. In fact, direct data addressing is applied to many instructions in a typical program. There are two basic forms of direct data addressing: (1) **direct addressing,** which applies to a MOV between a memory location and AL, AX, or EAX, and (2) **displacement addressing,** which applies to almost any instruction in the instruction set. In either case, the 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. A MOV instruction using this type of addressing is usually a 3-byte long instruction. (In the 80386 and above, a register size prefix may appear before the instruction, causing it to exceed three bytes in length.)

The MOV AL,DATA instruction, as represented by most assemblers, loads AL from data segment memory location DATA (1234H). Memory location DATA is a symbolic memory location, while the 1234H is the actual hexadecimal

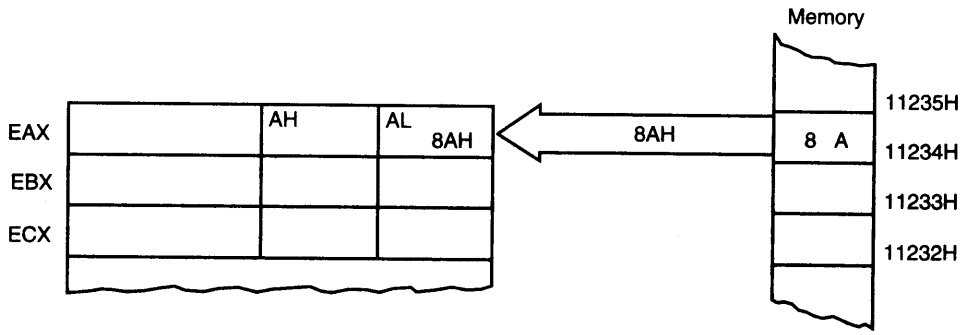


FIGURE 3–5 The operation of the MOV AL,[1234H] instruction when DS = 1000H.

location. With many assemblers, this instruction is represented as a MOV AL,[1234H][7] instruction. The [1234H] is an absolute memory location that is not allowed by all assembler programs. Note that this may need to be formed as MOV AL,DS:[1234H] with some assemblers, to show that the address is in the data segment. Figure 3–5 shows how this instruction transfers a copy of the byte-sized contents of memory location 11234H into AL. The effective address is formed by adding 1234H (the offset address) to 1000H (the data segment address of 1000H) in a system operating in the real mode.

Table 3–3 lists the three direct addressed instructions. These instructions often appear in programs, so Intel decided to make them special three-byte long instructions to reduce the length of programs. All other instructions that move data from a memory location to a register, called **displacement-addressed instructions,** require four or more bytes of memory for storage in a program.

Displacement Addressing. Displacement addressing is almost identical to direct addressing, except that the instruction is four bytes wide instead of three. In the 80386 through the Pentium 4, this instruction can be up to seven 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.

If the operation of the MOV CL,DS:[1234H] instruction is compared to that of the MOV AL,DS:[1234H] instruction of Figure 3–5, both basically perform the same operation except for the destination register (CL versus AL). Another difference only becomes apparent upon examining the assembled versions of these two instructions. The MOV AL,DS:[1234H] instruction is three bytes long and the MOV CL,DS:[1234H] instruction is four bytes long, as illustrated in Example 3–4. This example shows how the assembler converts these two instructions into hexadecimal machine language. You must include the segment register DS: in this example, before the [offset] part of the instruction. You may use any segment register, but, in most cases, data are stored in the data segment, so this example uses DS:[1234H].

TABLE 3–3 Direct addressed instructions using AX and AL.

Assembly Language	*Size*	*Operation*
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 NEWS,AL	8-bits	Copies AL into data segment memory location NEWS
MOV THERE,AX	16-bits	Copies AX into data segment memory location THERE
MOV ES:[2000 H],AL	8-bits	Copies AL into extra data segment memory location 2000H

[7]This form may be used with MASM, but most often appears when a program is entered or listed by DEBUG, a debugging toll provided with DOS.

EXAMPLE 3–4

```
0000  A0 1234 R          MOV AL,DS:[1234H]
0003  8A 0E 1234 R       MOV CL,DS:[1234H]
```

Table 3–4 lists some MOV instructions, using the displacement form of direct addressing. Not all variations are listed because there are many MOV instructions of this type. The segment registers can be stored or loaded from memory.

Example 3–5 shows a short program using models that address information in the data segment. Note that the **data segment** begins with a .DATA statement to inform the assembler where the data segment begins. The model size is adjusted from .TINY, as shown in Example 3–3, to SMALL so that a data segment can be included. The **SMALL model** allows one data segment and one code segment. The SMALL model is often used whenever memory data are required for a program. A SMALL model program assembles as an execute (.EXE) program. Notice how this example allocates memory locations in the data segment by using the DB and DW directives. Here the .STARTUP statement not only indicates the start of the code, but it also loads the data segment register with the segment address of the data segment. If this program is assembled and executed with CodeView, the instructions can be viewed as they execute and change registers and memory locations.

EXAMPLE 3–5

```
                          .MODEL SMALL       ;select SMALL model
0000                      .DATA              ;indicate start of DATA segment

0000 10          DATA1    DB     10H         ;place 10H in DATA1
0001 00          DATA2    DB     0           ;place 0 in DATA2
0002 0000        DATA3    DW     0           ;place 0 in DATA3
0004 AAAA        DATA4    DW     0AAAAH      ;place AAAAH in DATA4

0000                      .CODE              ;indicate start of CODE segment
                          .STARTUP           ;indicate start of program

0017 A0 0000 R            MOV    AL,DATA1    ;copy DATA1 to AL
001A 8A 26 0001 R         MOV    AH,DATA2    ;copy DATA2 to AH
001E A3 0002 R            MOV     DATA3,AX    ;save AX at DATA3
0021 8B 1E 0004 R         MOV     BX,DATA4    ;load BX with DATA4

                           EXIT               ;exit to DOS
                           END                 ;end file
```

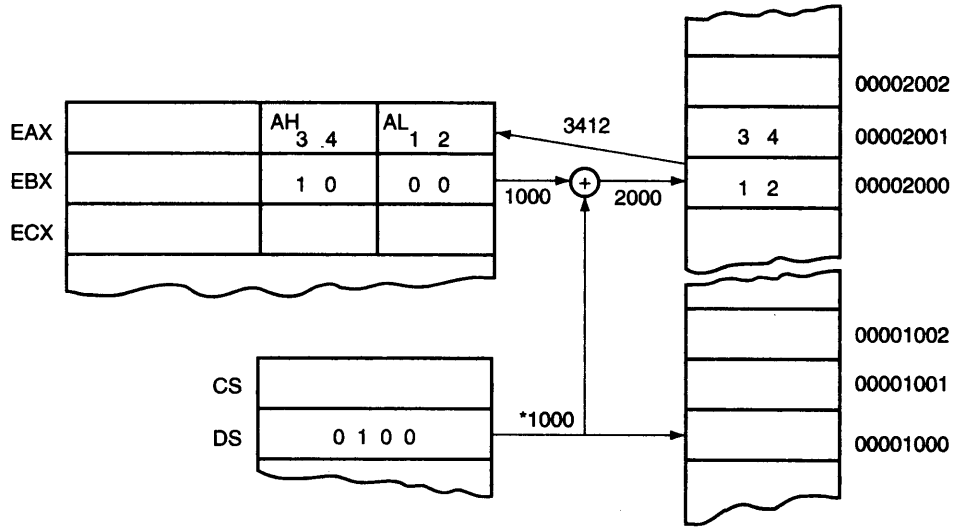
TABLE 3–4 Examples of direct data addressing using a displacement.

Assembly Language	*Size*	*Operation*
MOV CH,DOG	8-bits	Copies the byte contents of data segment memory location DOG into CH
MOV CH,[1000H]*	8-bits	Copies the byte contents of data segment 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

**Note:* This form of addressing is seldom used with most assemblers because an actual numeric offset address is rarely accessed.

Register Indirect Addressing

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. For example, if register BX contains a 1000H and the MOV AX,[BX] instruction executes, the word contents of data segment offset address 1000H are copied into register AX. If the microprocessor is operated in the real mode and DS = 0100H, this instruction addresses a word stored at memory bytes 2000H and 2001H, and transfers it into register AX (see Figure 3–6). Note that the contents of 2000H are moved into AL and the contents of 2001H are moved into AH. The [ ] symbols denote indirect addressing in assembly language. In addition to using the BP, BX, DI, and SI registers to indirectly address memory, the 80386 and above allow register indirect addressing with any extended register except ESP. Some typical instructions using indirect addressing appear in Table 3–5.



*After DS is appended with a 0.

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.

TABLE 3–5 Example of register indirect addressing.

Assembly Language	*Size*	*Operation*
MOV CX,[BX]	16-bits	Copies the word contents of the data segment memory location address 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 moves are not allowed except with string instructions

**Note:* Data addressed by BP are by default located in the stack segment, while all other indirect addressing modes use the data segment by default.

The **data segment** is used by default with register indirect addressing or any other addressing 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, the contents of the 32-bit 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 of the segment, as dictated by the access rights byte. An example 80386/80486/Pentium 4 instruction is MOV EAX,[EBX]. This instruction loads EAX with the doubleword-sized number stored at the data segment offset address indexed by EBX.

In some cases, indirect addressing requires specifying the size of the data are specified with the **special assembler directive** BYTE PTR, WORD PTR, or DWORD 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. Does the MOV [DI],10H instruction address a byte-, word-, or doubleword-sized memory location? The assembler can't determine the size of the 10H. 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 clearly identifies the memory location as doubleword-sized. The BYTE PTR, WORD PTR, and DWORD PTR directives are used only with instructions that address a memory location through a pointer or index register with immediate data, and for a few other instructions that are described in subsequent chapters.

Indirect addressing often allows a program to refer to tabular data located in the memory system. For example, suppose that you must create a table of information that contains 50 samples taken from memory location 0000:046C. Location 0000:046C contains a counter that is maintained by the personal computer's real-time clock. 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.

The sequence shown in Example 3–6 loads register BX with the starting address of the table and initializes the count, located in register CX, to 50. The OFFSET directive tells the assembler to load BX with the offset address of memory location TABLE, not the contents of TABLE. For example, the MOV BX,DATAS instruction copies the contents of memory location DATAS into BX, while the MOV BX,OFFSET DATAS instruction copies the offset address of DATAS into BX. When the OFFSET directive is used with the MOV instruction, the assembler calculates the offset address and then uses a MOV immediate instruction to load the address into the specified 16-bit register.

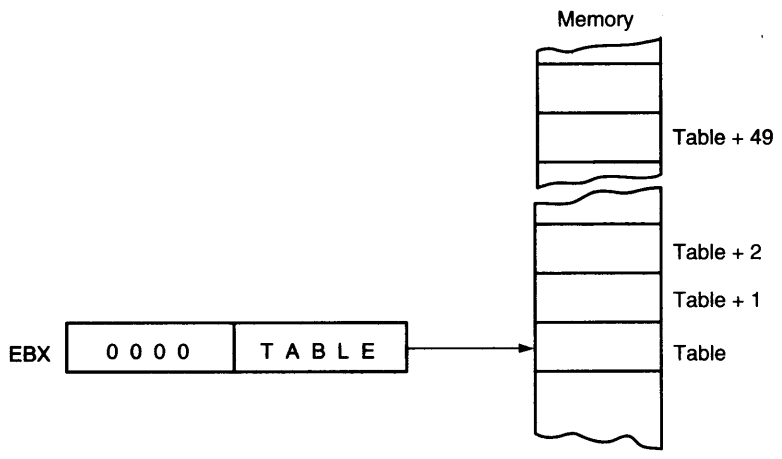


FIGURE 3–7 An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.

EXAMPLE 3–6

```
                        .MODEL SMALL                ;select SMALL model
0000                    .DATA                       ;start of DATA segment

0000  0032 [        DATAS  DW     50 DUP (?)        ;setup array of 50 bytes
              0000
                  ]

0000                    .CODE                       ;start of CODE segment
                        .STARTUP                    ;start of program

0017  B8 0000           MOV       AX,0
001A  8E C0             MOV       ES,AX             ;address segment 0000 with ES

001C  BB 0000 R         MOV       BX,OFFSET DATAS   ;address DATAS array
001F  B9 0032           MOV         CX,50           ;load counter with 50
0022                AGAIN:
0022  26:A1 046C        MOV       AX,ES:[046CH]     ;get clock value
0026  89 07             MOV       [BX],AX           ;save clock value in DATAS
0028  43                INC       BX                ;increment BX to next element
0029  E2 F7             LOOP      AGAIN             ;repeat 50 times

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

Once the counter and pointer are initialized, a repeat-until CX = 0 loop executes. Here, data are read from extra segment memory location 46CH with the MOV AX,ES:[046CH] instruction and stored in memory that is indirectly addressed by the offset address located in register BX. Next, BX is incremented (one is added to BX) to the next table location, and finally the LOOP instruction repeats the LOOP 50 times. The LOOP instruction decrements (subtracts one from) the counter (CX); if CX is not zero, LOOP causes a jump to memory location AGAIN. If CX becomes zero, no jump occurs and this sequence of instructions ends. This example copies the most recent 50 values from the clock into the memory array DATAS. This program will often show the same data in each location because the contents of the clock are changed only 18.2 times per second. To view the program and its execution, use the CodeView program. To use CodeView, type CV FILE.EXE or access it as DEBUG from the Programmer's WorkBench program under the RUN menu. Note that CodeView functions only with .EXE or .COM files. Some useful CodeView switches are /50 for a 50-line display and /S for use of high-resolution video displays in an application. To debug the file TEST.COM with 50 lines, type CV /50 TEST.COM at the DOS prompt.

Base-Plus-Index Addressing

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

In the 80386 and above, this type of addressing allows the combination of any two 32-bit extended registers except ESP. For example, the MOV DL,[EAX+EBX] instruction is an example using EAX (as the base) plus EBX (as the index). If the EBP register is used, the data are located in the stack segment instead of in the data segment.

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. In this example, BX = 1000H, DI = 0010H, and DS = 0100H, which translate into memory address 02010H. This instruction transfers a copy of the word from location 02010H into the DX register. Table 3–6 lists some instructions used for base-plus-index addressing. Note that the Intel assembler requires that 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.

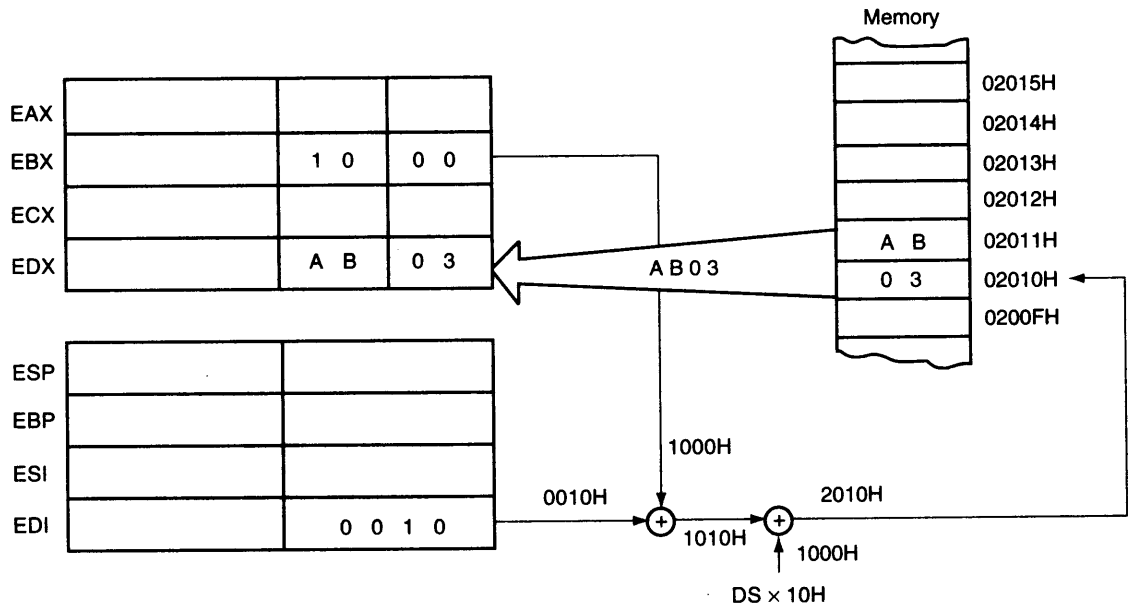


FIGURE 3–8 An example showing how the base-plus-index addressing mode functions for the MOV DX,[BX+DI] instruction. Notice that memory address 02010H is accessed because DS = 0100H, BX =100H, and DI = 0010H.

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

Assembly Language	*Size*	*Operation*
MOV CX,[BX+DI]	16-bits	Copies the word contents of the data segment memory location address 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 addresses by BX plus SI
MOV [BP+DI],AH	8-bits	Copies AH into the stack segment memory location addressed by BP plus DI

This text uses the first form in all example programs, but the second form can be used in many assemblers, including MASM from Microsoft. Instructions like MOV DI,[BX+DI] will assemble, but will not execute correctly.

Locating Array Data Using Base-plus-index Addressing. A major use of the base-plus-index addressing mode is to address elements in a memory array. Suppose that the elements in an array, located in the data segment at memory location ARRAY, must be accessed. 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.

A short program, listed in Example 3–7, moves array element 10H into array element 20H. Notice that the array element number, loaded into the DI register, addresses the array element. Also notice how the contents of the ARRAY have been initialized so that element 10H contains a 29H.

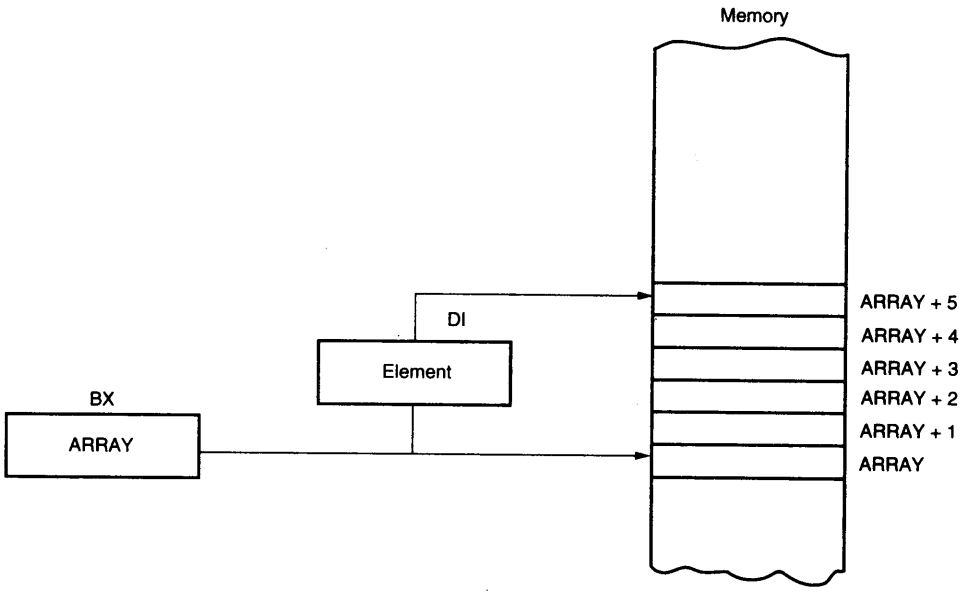


FIGURE 3–9 An example of the base-plus-index addressing mode. Here an element (DI) of an ARRAY (BX) is addressed.

EXAMPLE 3–7

```
                              .MODEL SMALL              ;select SMALL model
0000                          .DATA                     ;start of DATA segment

0000  0010 [        ARRAY     DB    16 DUP (?)          ;setup ARRAY
           00
              ]
0010  29                      DB    29H                 ;sample data at element 10H
0011  001E [                  DB    30 DUP (?)
           00
              ]
0000                          .CODE                     ;start of CODE segment
                              .STARTUP                  ;start of program

0017  BB 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 of file
```

Register Relative Addressing

Register relative addressing is similar to base-plus-index addressing and displacement addressing. In register relative addressing, the 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. In this example, BX = 0100H and DS = 0200H, so the address generated is the sum of DS × 10H, BX, and the displacement of 1000H or 03100H. Remember that BX, DI, or SI addresses the data segment and BP addresses the stack segment. In the 80386 and above, the displacement can be a 32-bit number and the register can

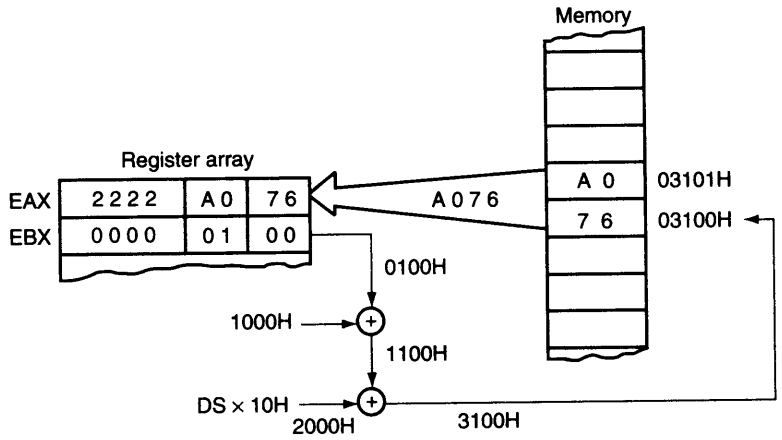


FIGURE 3–10 The operation of the MOV AX,[BX+1000H] instruction, when BX = 0100H and DS = 0200H.

TABLE 3–7 Examples of register relative addressing.

Assembly Language	*Size*	*Operation*
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 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 the sum of SET_IT and BX into DI

be any 32-bit register except the ESP register. Remember that the size of a real mode segment is 64K bytes long. Table 3–7 lists a few instructions that use register relative addressing.

The displacement can be a number added to the register within the [ ], as in the MOV AL,[DI+2] instruction, or it can be a displacement subtracted from the register, as in MOV AL,[SI–1]. A displacement also can be an offset address appended to the front of the [ ], as in MOV AL,DATA[DI]. Both forms of displacements also can appear simultaneously, as in the MOV AL,DATA[DI+3] instruction. In all cases, both forms of the displacement add to the base, or base and index register within the [ ]. In the 8086–80286 microprocessors, the value of the displacement is limited to a 16-bit signed number with a value ranging between +32,767 (7FFFH) and –32,768 (8000H); in the 80386 and above, a 32-bit displacement is allowed with a value ranging between +2,147,483,647 (7FFFFFFFH) and –2,147,483,648 (80000000H).

Addressing Array Data with Register Relative. It is possible to address array data with register relative addressing, such as one does 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.

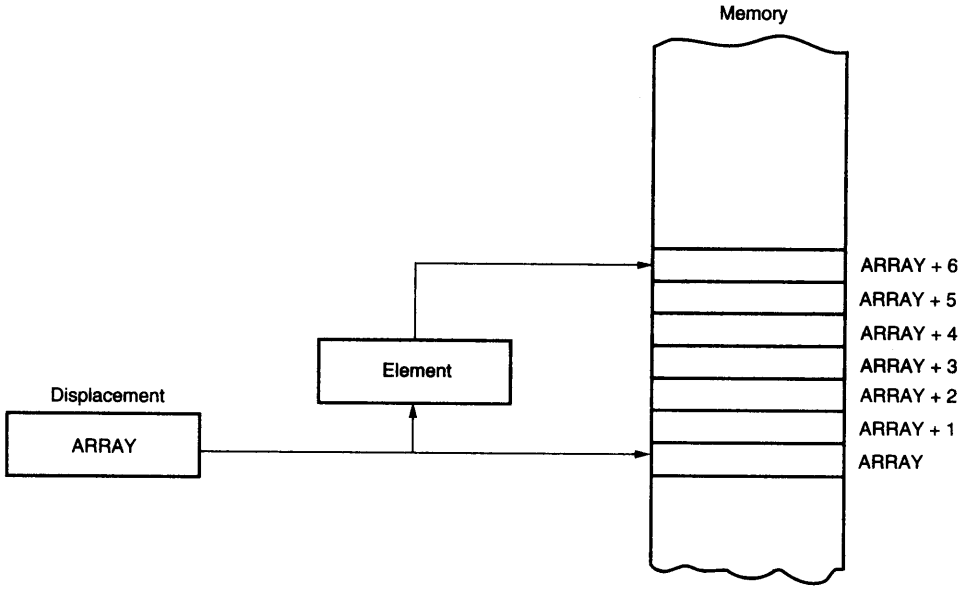


FIGURE 3–11 Register relative addressing used to address an element of ARRAY. The displacement addresses the start of ARRAY, and DI accesses an element.

Example 3–8 shows how this new addressing mode can transfer the contents of array element 10H into array element 20H. Notice the similarity between this example and Example 3–7. The main difference is that, in Example 3–8, register BX is not used to address memory area ARRAY; instead, ARRAY is used as a displacement to accomplish the same task.

EXAMPLE 3–8

```
                        .MODEL SMALL            ;select SMALL model
0000                    .DATA                   ;start of DATA segment

0000  0010 [      ARRAY DB    16 DUP (?)        ;setup ARRAY
            00
              ]
0010  29                DB    29H               ;sample data at element 10H
0011  001E [            DB    30 DUP (?)
            00
              ]
0000                    .CODE                   ;start of CODE segment
                        .STARTUP                ;start of program

0017  BF 0010           MOV   DI,10H            ;address element 10H
001A  8A 85 0000 R      MOV   AL,ARRAY[DI]      ;get element 10H
001E  BF 0020           MOV   DI,20H            ;address element 20H
0021  88 85 0000 R      MOV   ARRAY[DI],AL      ;save in element 20H

                        .EXIT                   ;exit to DOS
                        END                     ;end of file
```

Base Relative-Plus-Index Addressing

The base relative-plus-index addressing mode is similar to the base-plus-index addressing mode, but it adds a displacement, besides using 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. Base relative-plus-index addressing is the least-used addressing mode. Figure 3–12 shows how data are referenced if the instruction executed by the microprocessor is a MOV AX,[BX+SI+100H]. The displacement of 100H adds to BX and SI to form the offset address within the data segment. Registers BX = 0020H, SI = 0010H, and DS = 1000H, so the effective address for this instruction is 10130H–the sum of these registers plus a displacement of 100H. This addressing mode is too complex for frequent use in a program. Some typical instructions using base relative-plus-index addressing appear in Table 3–8. Note that with the 80386 and above, the effective address is generated by the sum of two 32-bit registers plus a 32-bit displacement.

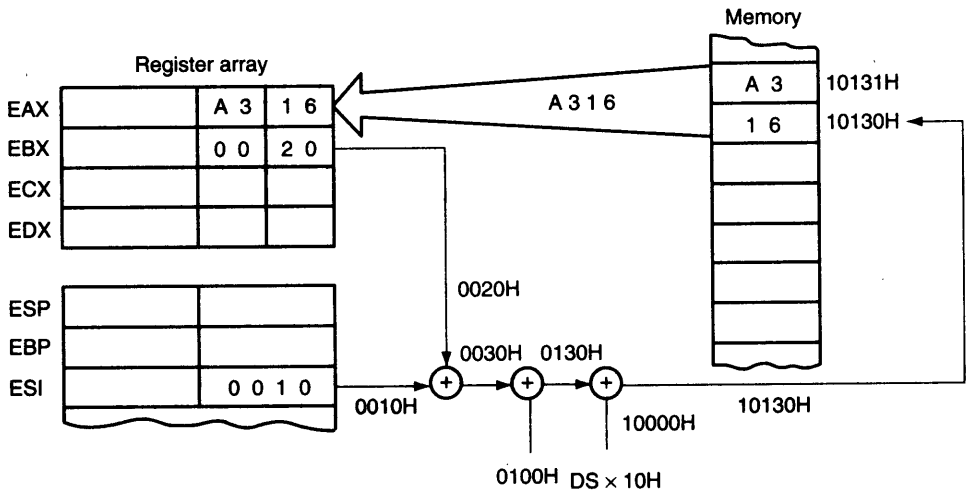


FIGURE 3–12 An example of base relative-plus-index addressing using a MOV AX,[BX+SI+100H] instruction. Note: DS = 1000H.

TABLE 3–8 Example base relative-plus-index instructions.

Assembly Language	*Size*	*Operation*
MOV DH,[BX+DI+20H]	8-bits	Copies the byte contents of the data segment memory location addressed by the sum of BX, DI, and 20H into DH
MOV AX,FILE[BX+DI]	16-bits	Copies the word contents of the data segment memory location addressed by the sum of FILE, BX, and DI into AX
MOV LIST[BP+DI],CL	8-bits	Copies CL into the stack segment memory location addressed by the sum of LIST, BP, and DI
MOV LIST[BP+SI+4],DH	8-bits	Copies DH into the stack segment memory location addressed by the sum of LIST, BP, SI, and 4

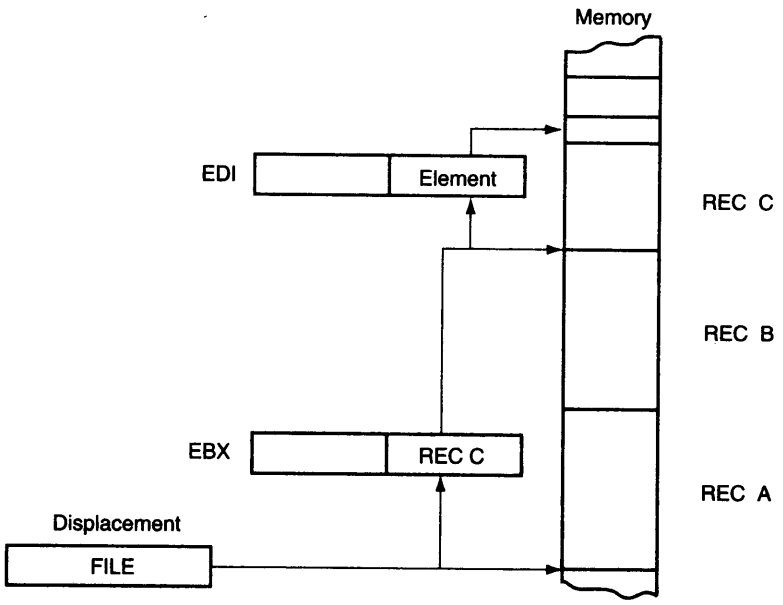


FIGURE 3–13 Base relative-plus-index addressing used to access a FILE that contains multiple records (REC).

Addressing Arrays with Base Relative-plus-index. Suppose that a file of many records exists in memory and each record contains many elements. This displacement addresses the file, the base register addresses a record, and the index register addresses an element of a record. Figure 3–13 illustrates this very complex form of addressing.

Example 3–9 provides a program that copies element 0 of record A into element 2 of record C by using the base relative-plus-index mode of addressing. This example FILE contains four records and each record contains 10 elements. Notice how the THIS BYTE statement is used to define the label FILE and RECA as the same memory location.

EXAMPLE 3–9

```
                        .MODEL SMALL                    ;SMALL model
0000                    .DATA                           ;start of DATA segment

0000  = 0000            FILE  EQU     THIS BYTE         ;assign FILE to this byte

0000  000A [            RECA  DB      10 DUP (?)        ;reserve 10 bytes for RECA
              00
                 ]
000A  000A [            RECB  DB      10 DUP (?)        ;reserve 10 bytes for RECB
              00
                 ]
0014  000A [            RECC  DB      10 DUP (?)        ;reserve 10 bytes for RECC
              00
                 ]
001E  000A [            RECD  DB      10 DUP (?)        ;reserve 10 bytes for RECD
              00
                 ]

0000                          .CODE                     ;start of CODE segment
                              .STARTUP                  ;start of program
```

```
0017  BB 0000 R          MOV     BX,OFFSET RECA      ;address RECA
001A  BF 0000            MOV     DI,0                ;address element 0
001D  8A 81 0000 R       MOV     AL,FILE[BX+DI]      ;get data
0021  BB 0014 R          MOV     BX,OFFSET RECC      ;address RECC
0024  BF 0002            MOV     DI,2                ;address element 2
0027  88 81 0000 R       MOV     FILE[BX+DI],AL      ;save data

                         .EXIT                       ;exit to DOS
                         END                         ;end of file
```

Scaled-Index Addressing

Scaled-index addressing is the last type of data-addressing mode discussed. This data-addressing mode is unique to the 80386 through the Pentium 4 microprocessors. Scaled-index addressing 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, or 8X. A scaling factor of 1X is implied and need not be included in the assembly language instruction (MOV AL,[EBX+ECX]). A scaling factor of 2X is used to address word-sized memory arrays, a scaling factor of 4X is used with doubleword-sized memory arrays, and a scaling factor of 8X is used with quadword-sized memory arrays.

An example instruction is MOV AX,[EDI+2*ECX]. This instruction uses a scaling factor of 2X, which multiplies the contents of ECX by 2 before adding it to the EDI register to form the memory address. If ECX contains a 00000000H, word-sized memory element 0 is addressed; if ECX contains a 00000001H, word-sized memory element 1 is accessed, and so forth. This scales the index (ECX) by a factor of 2 for a word-sized memory array. Refer to Table 3–9 for some examples of scaled-index addressing. As you can imagine, there are an extremely large number of the scaled-index addressed register combinations. Scaling is also applied to instructions that use a single indirect register to access memory. The MOV EAX,[4*EDI] is a scaled-index instruction that uses one register to indirectly address memory.

Example 3–10 shows a sequence of instructions that uses scaled-index addressing to access a word-sized array of data called LIST. Note that the offset address of LIST is loaded into register EBX with the MOV EBX,OFFSET LIST instruction. Once EBX addresses array LIST, the elements (located in ECX) of 2, 4, and 7 of this word-wide array are added, using a scaling factor of 2 to access the elements. This program stores the 2 at element 2 into elements 4 and 7. Also notice the .386 directive to select the 80386 microprocessor. This directive must follow the .MODEL statement for the assembler to process 80386 instructions for DOS. If the 80486 is in use, the .486 directive appears after the .MODEL statement; if the Pentium, Pentium Pro, Pentium II, Pentium III,

TABLE 3–9 Examples of scaled-index addressing. (Not for 8086/ 8088/80286 readers).

Assembly Language	*Size*	*Operation*
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 plus 4 times ECX into EAX

or Pentium 4 is in use, the .586 directive appears after .MODEL. If the microprocessor selection directive appears before the .MODEL statement, the microprocessor executes instructions in the 32-bit mode, which is not compatible with DOS.

EXAMPLE 3–10

```
                                .MODEL SMALL            ;select SMALL model
                                .386                    ;use the 80386
0000                            .DATA                   ;start of DATA segment
0000  0000 0001 0002  LIST      DW   0,1,2,3,4          ;define array list
      0003 0004
000A  0005 0006 0007            DW   5,6,7,8,9
      0008 0009

0000                            .CODE                   ;start of CODE segment
                                .STARTUP                ;start of program
0010  66| BB 00000000 R         MOV  EBX,OFFSET LIST    ;address array LIST

0016  66| B9 00000002           MOV  ECX,2              ;get element 2
001C  67& 8B 04 4B              MOV  AX,[EBX+2*ECX]

0020  66| B9 00000004           MOV  ECX,4              ;store in element 4
0026  67& 89 04 4B              MOV  [EBX+2*ECX],AX

002A  66| B9 00000007           MOV  ECX,7              ;store in element 7
0030  67& 89 04 4B              MOV  [EBX+2*ECX],AX

                                .EXIT                   ;exit to DOS
                                END                     ;end of file
```

Data Structures

A data structure is used to specify how information is stored in a memory array and can be quite useful with applications that use arrays. It is best to think of a data structure as a template for data. The start of a structure is identified with the STRUC assembly language directive and the end with the ENDS statement. A typical data structure is defined and used three times in Example 3–11. Notice that the name of the structure appears with the STRUC and with ENDS statement.

EXAMPLE 3–11

```
                        ;Define INFO data structure
0057                    INFO    STRUC

0000  0020  [           NAMES   DB    32 DUP (?)        ;32 bytes for name
              00
                ]
0020  0020  [           STREET  DB    32 DUP (?)        ;32 bytes for street
              00
                ]
0040  0010  [           CITY    DB    16 DUP (?)        ;16 bytes for city
              00
                ]
0050  0002  [           STATE   DB     2 DUP (?)        ;2 bytes for state
              00
                ]
0052  0005  [           ZIP     DB     5 DUP (?)        ;5 bytes for zip-code
              00
                ]
                        INFO    ENDS
```

```
0000 42 6F 62 20 53 6D NAME1  INFO   <'Bob Smith','123 Main Street','Wanda','OH','44444'>
        69 74 68
        0017 [
               00
                  ]
        31 32 33 20 4D
        61 69 6E 20 53 74
        72 65 65 74
        0011 [
               00
                  ]
        57 61 6E 64 61
        000B [
               00
                  ]
        4F 48 34 34 34
        34 34

0057 53 74 65 76 65 20  NAME2 INFO <'Steve Doe','222 Mouse Lane','Miller','PA','18100'>
        44 6F 65
        0017 [
               00
                  ]
        32 32 32 20 4D
        6F 75 73 65 20 4C
        61 6E 65
        0012 [
               00
                  ]
        4D 69 6C 6C 65
        72
        000A [
               00
                  ]
        50 41 31 38 31
        30 30

00AE 42 65 6E 20 44 6F  NAME3 INFO <'Jim Dover','303 Main Street','Orender','CA','90000'>
        76 65 72
        0017 [
               00
                  ]
        33 30 33 20 4D
        61 69 6E 20 53 74
        72 65 65 74
        0011 [
               00
                  ]
        4F 72 65 6E 64
        65 72
        0009 [
               00
                  ]
        43 41 39 30 30
        30 30
```

The data structure in Example 3–11 defines five fields of information. The first is 32 bytes long and holds a name; the second is 32 bytes long and holds a street address; the third is 16 bytes long for the city; the fourth is 2 bytes long for the state; the fifth is 5 bytes long for the ZIP Code. Once the structure is defined (INFO), it can be filled, as illustrated, with names and addresses. Three examples of uses for INFO are illustrated. Note that literals are surrounded with apostrophes and the entire field is surrounded with < > symbols when the data structure is used to define data.

When data are addressed in a structure, use the structure name and the field name to select a field from the structure. For example, to address the STREET in NAME2, use the operand NAME2.STREET, where the name of the structure is first followed by a period and then by the name of the field. Likewise, use NAME3.CITY to refer to the city in structure NAME3.

EXAMPLE 3–12

```
                        ;Clear names in array NAME1

0000  B9 0020                     MOV   CX,32
0003  B0 00                       MOV   AL,0
0005  BE 0000 R                  MOV   SI,OFFSET NAME1.NAMES
0008  F3/AA                       REP   STOSB

                        ;Clear street in array NAME2

000A  B9 0020                     MOV   CX,32
000D  B0 00                       MOV   AL,0
0010  BE 0077 R                  MOV   SI,OFFSET NAME2.STREET
0013  F3/AA                       REP   STOSB

                       ;Clear zip-code in array NAME3

0015  B9 0005                     MOV   CX,5
0018  B0 00                       MOV   AL,0
001A  BE 0100 R                  MOV   SI,OFFSET NAME3.ZIP
001D  F3/AA                       REP   STOSB
```

A short sequence of instructions appears in Example 3–12 that clears the name field in structure NAME1, the address field in structure NAME2, and the ZIP Code field in structure NAME3. The function and operation of the instructions in this program are defined in later chapters in the text. You may wish to refer to this example once these instructions are learned.

3–2 PROGRAM MEMORY-ADDRESSING MODES

Program memory-addressing modes, used with the JMP and CALL instructions, consist of three distinct forms: direct, relative, and indirect. This section introduces these three addressing forms, using the JMP instruction to illustrate their operation.

Direct Program Memory Addressing

Direct program memory addressing is what many early microprocessors used for all jumps and calls. Direct program memory addressing is also used in high-level languages, such as the BASIC language GOTO and GOSUB instructions. The microprocessor uses this form of addressing, but not as often as relative and indirect program memory addressing are used.

The instructions for direct program memory addressing store the address with the opcode. For example, if a program jumps to memory location 10000H for the next instruction, the address (10000H) is stored following the opcode in the memory. Figure 3–14 shows the direct intersegment JMP instruction and the four bytes required to store the address 10000H. 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.) The direct jump is often called a *far jump* because it can jump to any memory location for the next instruction. In the real mode, a far jump accesses any loca-

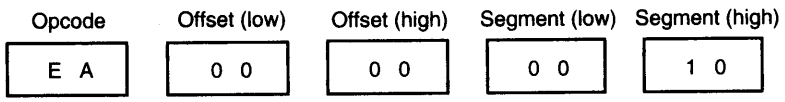


FIGURE 3–14 The 5-byte machine language version of a JMP [10000H] instruction.

tion within the first 1M byte of memory by changing both CS and IP. In protected mode operation, the far jump accesses a new code segment descriptor from the descriptor table, allowing it to jump to any memory location in the entire 4G-byte address range in the 80386 through Pentium 4 microprocessors.

The only other instruction that uses 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

Relative program memory addressing is 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)." For example, if a JMP instruction skips the next two bytes of memory, the address in relation to the instruction pointer is a 2 that adds to the instruction pointer. This develops the address of the next program instruction. An example of the relative JMP instruction is shown in Figure 3–15. Notice that the JMP instruction is a one-byte instruction, with a one-byte or a two-byte displacement that adds to the instruction pointer. A one-byte displacement is used in short jumps, and a two-byte displacement is used with near jumps and calls. Both types are considered to be intrasegment jumps. (An **intrasegment jump** is a jump anywhere within the current code segment.) In the 80386 and above, the displacement can also be a 32-bit value, allowing them to use relative addressing to any location within their 4G-byte code segments.

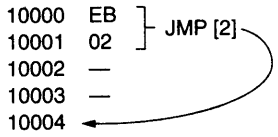


FIGURE 3–15 A JMP [2] instruction. This instruction skips over the two bytes of memory that follow the JMP instruction.

Relative JMP and CALL instructions contain either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference. (The 80386 and above can have an 8-bit or 32-bit displacement.) All assemblers automatically calculate the distance for the displacement and select the proper one-, two- or four-byte form. If the distance is too far for a two-byte displacement in an 8086 through 80286 microprocessor, some assemblers use the direct jump. An 8-bit displacement *(short)* has a jump range of between +127 and –128 bytes from the next instruction, while a 16-bit displacement *(near)* has a range of ±32K bytes. In the 80386 and above, a 32-bit displacement allows a range of ±2G bytes. The 32-bit displacement can only be used in the protected mode.

Indirect Program Memory Addressing

The microprocessor allows several forms of program indirect memory addressing for the JMP and CALL instructions. Table 3–10 lists some acceptable program indirect jump instructions, which can use any 16-bit register (AX, BX, CX, DX, SP, BP, DI, or SI); any relative register ([BP], [BX], [DI], or [SI]); and any relative register with a displacement. In the 80386 and above, an extended register can also 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 16-bit register holds the address of a JMP instruction, the jump is near. For example, if the BX register contains a 1000H and a JMP BX instruction executes, the microprocessor jumps to offset address 1000H in the current code segment.

TABLE 3–10 Examples of indirect program memory addressing.

Assembly Language	*Operation*
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 memory 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 addressed by TABLE plus BX

If a relative register holds the address, the jump is also considered to be an indirect jump. For example, a 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 that is 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. This jump table is referenced by the short program of Example 3–13. In this example, the BX register is loaded with a 4 so, when it combines in the JMP TABLE[BX] instruction with TABLE, the effective address is the contents of the second entry in the jump table.

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

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.

EXAMPLE 3–13

```
                        ;Using indirect addressing for a jump
                        ;
0000  BB 0004                   MOV BX,4                ;address LOC2
0003  FF A7 23A1 R              JMP TABLE[BX]           ;jump to LOC2
```

3–3 STACK MEMORY-ADDRESSING MODES

The stack plays an important role in all microprocessors. It holds data temporarily and stores return addresses for procedures. The stack memory is a LIFO **(last-in, first-out)** memory, which describes the way that data are stored and removed from the stack. Data are placed onto the stack with a **PUSH instruction** and removed with a **POP instruction.** The CALL instruction also uses the stack to hold the return address for procedures and a RET (return) instruction to remove the return address from the stack.

The stack memory is maintained by two registers: the stack pointer (SP or ESP) and the stack segment register (SS). Whenever a word of data is pushed onto the stack [see Figure 3–17(a)], the high-order 8 bits are placed in the location addressed by SP – 1. The low-order 8 bits are placed in the location addressed by SP – 2. The SP is then decremented by 2 so that the next word of data is stored in the next available stack memory location. The SP/ESP register always points to an area of memory located within the stack segment. The SP/ESP register adds to SS ∞ 10H to form the stack memory address in the real mode. In protected mode operation, the SS register holds a selector that accesses a descriptor for the base address of the stack segment.

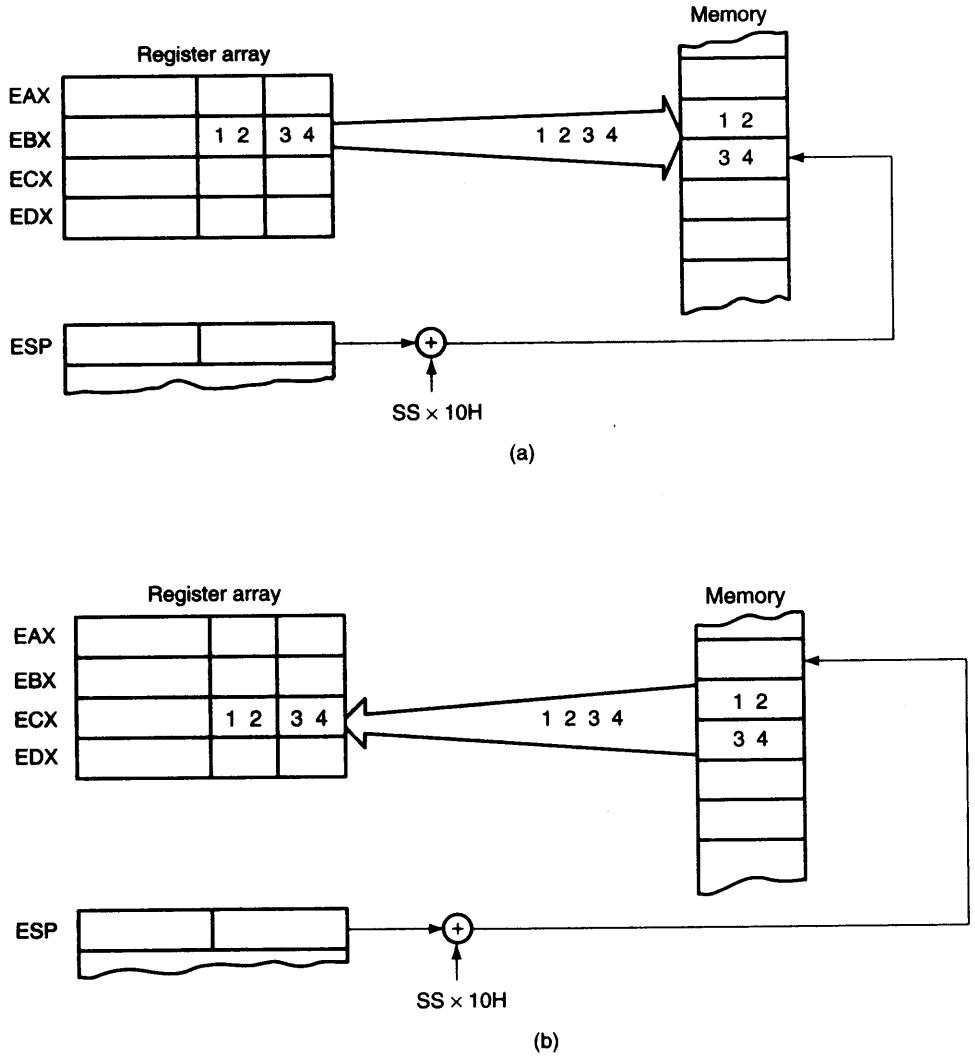


FIGURE 3–17 The PUSH and POP instructions. (a) PUSH BX places the contents of BX onto the stack, (b) POP CX removes data from the stack and places them into CX. Both instructions are shown after execution.

Whenever data are popped from the stack [see Figure 3–17(b)], the low-order 8 bits are removed from the location addressed by SP. The high-order 8 bits are removed from the location addressed by SP + 1. The SP register is then incremented by 2. Table 3–11 lists some of the PUSH and POP instructions available to the microprocessor. Note that PUSH and POP always store or retrieve words of data—never bytes—in the 8086 through the 80286 microprocessors. The 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 the 80386 and above, from any 32-bit extended register. Data may be popped off the stack into any 16-bit register or any segment register except CS. The reason that data may not be popped from the stack into CS is that this only changes part of the address of the next instruction.

TABLE 3–11 Example PUSH and POP instructions.

Assembly Language	*Operation*
POPF	Removes a word from the stack and places it into the flags
PUSHF	Copies the flags onto the stack
PUSH AX	Copies AX to the stack
POP BX	Removes a word from the stack and places it into BX
PUSH DS	Copies DS to the stack
PUSH 1234H	Copies a 1234H to the stack
POP CS	Illegal instruction
PUSH WORD PTR [BX]	Copies a word from the data segment memory location addressed by BX onto the stack
PUSH EDI	Copies EDI to the stack

The PUSHA and POPA instructions either push or pop all of the registers, except the segment registers, on the stack. These instructions are not available on the early 8086/8088 microprocessors. The push immediate instruction is also new to the 80286 through the Pentium microprocessors. Note the examples in Table 3–11, which show the order of the registers transferred by the PUSHA and POPA instructions. The 80386 and above also allow extended registers to be pushed or popped.

Example 3–14 lists a short program that pushes the contents of AX, BX, and CX onto the stack. The first POP retrieves the value that was pushed onto the stack from CX and places it into AX. The second POP places the original value of BX into CX. The last POP places the original value of AX into BX.

EXAMPLE 3–14

```
                        .MODEL TINY             ;select TINY model
0000                    .CODE                   ;start CODE segment
                        .STARTUP                ;start of program
0100  B8 1000           MOV     AX,1000H        ;load test data
0103  BB 2000           MOV     BX,2000H
0106  B9 3000           MOV     CX,3000H

0109  50                PUSH    AX              ;1000H to stack
010A  53                PUSH    BX              ;2000H to stack
010B  51                PUSH    CX              ;3000H to stack

010C  58                POP     AX              ;3000H to AX
010D  59                POP     CX              ;2000H to CX
010E  5B                POP     BX              ;1000H to BX
                        .EXIT                   ;exit to DOS
                        END                     ;end of file
```

3–4 SUMMARY

1. The data-addressing modes include register, immediate, direct, register indirect, base-plus-index, register relative, and base relative-plus-index addressing. In the 80386 through the Pentium 4 microprocessors, an additional addressing mode, called scaled-index addressing, exists.
2. The program memory-addressing modes include direct, relative, and indirect addressing.

3. Table 3–12 lists all real mode data-addressing modes available to the 8086 through the 80286 microprocessors. Note that the 80386 and above use these modes, plus the many defined through this chapter. In the protected mode, the function of the segment register is to address a descriptor that contains the base address of the memory segment.
4. The 80386 through Pentium 4 microprocessors have additional addressing modes that allow the extended registers EAX, EBX, ECX, EDX, EBP, EDI, and ESI to address memory. Although these addressing modes

TABLE 3–12 Example real mode data-addressing modes.

Assembly Language	*Address Generation*
MOV AL,BL	8-bit register addressing
MOV AX,BX	16-bit register addressing
MOV DS,CX	Segment register addressing
MOV AL,LIST	(DS x 10H) + LIST
MOV CH,DATA1	(DS x 10H) + DATA1
MOV ES,DATA2	(DS x 10H) + DATA2
MOV AL,12	Immediate data of 12H
MOV AL,[BP]	(SS x 10H) + BP
MOV AL,[BX]	{DS x 10H) + BX
MOV AL,[DI]	(DS x 10H) + DI
MOV AL,[SI]	(DS x 10H) + SI
MOV AL,[BP+2]	(SS x 10H) + BP + 2
MOV AL,[BX–4]	(DS x 10H) + BX – 4
MOV AL,[DI+1000H]	(DS x 10H) + DI + 1000H
MOV AL,[SI+300H]	(DS x 10H) + SI + 300H
MOV AL,LIST[BP]	(SS x 10H) + LIST + BP
MOV AL,LIST[BX]	(DS x 10H) + LIST + BX
MOV AL,LIST[DI]	(DS x 10H) + LIST + DI
MOV AL,LIST[SI]	(DS x 10H) + LIST + SI
MOV AL,LIST[BP+2]	(SS x 10H) + LIST + BP + 2
MOV AL,LIST[BX–6]	(DS x 10H) + LIST + BX – 6
MOV AL, LIST[DI+100H]	(DS x 10H) + LIST + DI + 100H
MOV AL,LIST[SI+200H]	(DS x 10H) + LIST + SI + 200H
MOV AL,[BP+DI]	(SS x 10H) + BP + DI
MOV AL,[BP+SI]	(SS x 10H) + BP + SI
MOV AL,[BX+DI]	(DS x 10H) + BX + DI
MOV AL,[BX+SI]	(DS x 10H) + BX + SI
MOV AL,[BP+DI+4]	(SS x 10H) + BP + DI +4
MOV AL,[BP+SI–8]	(SS x 10H) + BP + SI – 8
MOV AL,[BX+DI+10H]	(DS x 10H) + BX + DI + 10H
MOV AL,[BX+ SI–10H]	(DS x 10H) + BX + SI – 10H
MOV AL,LIST[BP+DI]	(SS x 10H) + LIST + BP + DI
MOV AL,LIST[BP+SI]	(SS x 10H) + LIST + BP + SI
MOV AL,LIST[BX+DI]	(DS x 10H) + LIST + BX + DI
MOV AL,LIST[BX+SI]	(DS x 10H) + LIST + BX + SI
MOV AL,LIST[BP+DI+2]	(SS x 10H) + LIST + BP + DI + 2
MOV AL,LIST[BP+SI–7]	(SS x 10H) + LIST + BP + SI – 7
MOV AL,LIST[BX+DI+3]	(DS x 10H) + LIST + BX + DI + 3
MOV AL,LIST[BX+SI–2]	(DS x 10H) + LIST + BX + SI – 2

are too numerous to list in tabular form, in general, any of these registers function in the same way as those listed in Table 3–12. For example, the MOV AL,TABLE[EBX+2*ECX+10H] is a valid addressing mode for the 80386–Pentium 4 microprocessors.

5. The MOV instruction copies the contents of the source operand into the destination operand. The source never changes for any instruction.
6. 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). 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 Pentium 4 microprocessors, the extended registers also are used for register addressing; they consist of EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. Also available to the 80386 and above are the FS and GS segment registers.
7. 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. In the 80386 and above, a doubleword immediate data may also be loaded into a 32-bit register or memory location.
8. 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 one segment, the code segment, and is assembled as a command (.COM) program. If the SMALL model is used, the program uses a code and data segment, and assembles as an execute (.EXE) program. Other model sizes and their attributes are listed in Appendix A.
9. Direct addressing occurs in two forms in the microprocessor: (1) direct addressing and (2) displacement addressing. 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 three bytes of memory, while displacement addressing requires four bytes. Note that some of these instructions in the 80386 and above may require additional bytes in the form of prefixes for register and operand sizes.
10. 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.
11. 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.
12. Register relative addressing uses either a base or index register, plus a displacement to access memory data.
13. 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.
14. Scaled-index addressing is unique to the 80386 through the Pentium 4. The second of two registers (index) is scaled by a factor of 2X, 4X, or 8X 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.
15. Data structures are templates for storing arrays of data, and are addressed by array name and field. For example, array NUMBER and field TEN of array NUMBER is addressed as NUMBER.TEN.
16. Direct program memory addressing is allowed with the JMP and CALL instructions to any location in the memory system. With this addressing mode, the offset address and segment address are stored with the instruction.
17. Relative program addressing allows a JMP or CALL instruction to branch forward or backward in the current code segment by ±32K 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 ±2G bytes. The 32-bit displacement can be used only in protected mode.

18. 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.
19. 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. The PUSHA and POPA instructions transfer AX, CX, DX, BX, BP, SP, SI, and DI between the stack and these registers. In the 80386 and above, the extended register and extended flags can also be transferred between registers and the stack. A PUSHFD stores the EFLAGS, while a PUSHF stores the FLAGS.
20. Example 3–15 shows many of the addressing modes presented in the chapter. This example program fills the ARRAY1 from locations 0000:0000–0000:0009. It then fills ARRAY2–0 through 9. Finally, it exchanges the contents of ARRAY1 element 2 with ARRAY2 element 3.

EXAMPLE 3–15

```
                        .MODEL SMALL              ;select SMALL model
0000                    .DATA                     ;start of DATA segment

0000  000A [      ARRAY1 DB     10 DUP (?)        ;reserve for ARRAY1
            00
              ]
000A  000A [      ARRAY2 DB     10 DUP (?)        ;reserve for ARRAY2
            00
              ]
0000                      .CODE                   ;start of CODE segment
                          .STARTUP                ;start of program

0017  B8 0000             MOV   AX,0              ;segment ES is 0000H
001A  8E C0               MOV   ES,AX

001C  BF 0000             MOV   DI,0              ;address element 0
001F  B9 000A             MOV   CX,10             ;count of 10
0022                LAB1:
0022  26:8A 05            MOV   AL,ES:[DI]        ;copy data
0025  88 85 0000 R        MOV   ARRAY1[DI],AL     ;into ARRAY1
0029  47                  INC   DI
002A  E2 F6               LOOP  LAB1

002C  BF 0000             MOV   DI,0              ;address element 0
002F  B9 000A             MOV   CX,10             ;count of 10
0032  B0 00               MOV   AL,0              ;initial value
0034                LAB2:
0034  88 85 000A R        MOV   ARRAY2[DI],AL     ;fill ARRAY2
0038  FE C0               INC   AL
003A  47                  INC   DI
003B  E2 F7               LOOP  LAB2

003D  BF 0003             MOV   DI,3              ;exchange array data
0040  8A 85 0000 R        MOV   AL,ARRAY1[DI]
0044  8A A5 000B R        MOV   AH,ARRAY2[DI+1]
0048  88 A5 0000 R        MOV   ARRAY1[DI],AH
004C  88 85 000B R        MOV   ARRAY2[DI+1],AL

                          .EXIT                   ;exit to DOS
                          END                     ;end of file
```

3–5 QUESTIONS AND PROBLEMS

1. What do the following MOV instructions accomplish?
   (a) MOV AX,BX
   (b) MOV BX,AX
   (c) MOV BL,CH
   (d) MOV ESP,EBP
   (e) MOV AX,CS
2. List the 8-bit registers that are used for register addressing.
3. List the 16-bit registers that are used for register addressing.
4. List the 32-bit registers that are used for register addressing in the 80386 through the Pentium 4 microprocessors.
5. List the 16-bit segment registers used with register addressing by MOV, PUSH, and POP.
6. What is wrong with the MOV BL,CX instruction?
7. What is wrong with the MOV DS,SS instruction?
8. Select an instruction for each of the following tasks:
   (a) copy EBX into EDX
   (b) copy BL into CL
   (c) copy SI into BX
   (d) copy DS into AX
   (e) copy AL into AH
9. Select an instruction for each of the following tasks:
   (a) move a 12H into AL
   (b) move a 123AH into AX
   (c) move a 0CDH into CL
   (d) move a 1000H into SI
   (e) move a 1200A2H into EBX
10. What special symbol is sometimes used to denote immediate data?
11. What is the purpose of the .MODEL TINY statement?
12. What assembly language directive indicates the start of the CODE segment?
13. What is a label?
14. The MOV instruction is placed in what field of a statement?
15. A label may begin with what characters?
16. What is the purpose of the .EXIT directive?
17. Does the .MODEL TINY statement cause a program to assemble an execute program?
18. What tasks does the .STARTUP directive accomplish in the small memory model?
19. What is a displacement? How does it determine the memory address in a MOV [2000H],AL instruction?
20. What do the symbols [ ] indicate?
21. Suppose that DS = 0200H, BX = 0300H, and DI = 400H. Determine the memory address accessed by each of the following instructions, assuming real mode operation:
   (a) MOV AL,[1234H]
   (b) MOV EAX,[BX]
   (c) MOV [DI],AL
22. What is wrong with a MOV [BX],[DI] instruction?
23. Choose an instruction that requires BYTE PTR.
24. Choose an instruction that requires WORD PTR.
25. Choose an instruction that requires DWORD PTR.
26. Explain the difference between the MOV BX,DATA instruction and the MOV BX,OFFSET DATA instruction.

27. Suppose that DS = 1000H, SS = 2000H, BP = 1000H, and DI = 0100H. Determine the memory address accessed by each of the following instructions, assuming real mode operation:
    (a) MOV AL,[BP+DI]
    (b) MOV CX,[DI]
    (c) MOV EDX,[BP]
28. What, if anything, is wrong with a MOV AL,[BX][SI] instruction?
29. Suppose that DS = 1200H, BX = 0100H, and SI = 0250H. Determine the address accessed by each of the following instructions, assuming real mode operation:
    (a) MOV [100H],DL
    (b) MOV [SI+100H],EAX
    (c) MOV DL,[BX+100H]
30. Suppose that DS = 1100H, BX = 0200H, LIST = 0250H, and SI = 0500H. Determine the address accessed by each of the following instructions, assuming real mode operation:
    (a) MOV LIST[SI],EDX
    (b) MOV CL,LIST[BX+SI]
    (c) MOV CH,[BX+SI]
31. Suppose that DS = 1300H, SS = 1400H, BP = 1500H, and SI = 0100H. Determine the address accessed by each of the following instructions, assuming real mode operation:
    (a) MOV EAX,[BP+200H]
    (b) MOV AL,[BP+SI–200H]
    (c) MOV AL,[SI–0100H]
32. Which base register addresses data in the stack segment?
33. Suppose that EAX = 00001000H, EBX = 00002000H, and DS = 0010H. Determine the addresses accessed by the following instructions, assuming real mode operation:
    (a) MOV ECX,[EAX+EBX]
    (b) MOV [EAX+2*EBX],CL
    (c) MOV DH,[EBX+4*EAX+1000H]
34. Develop a data structure that has five fields of one word each named F1, F2, F3, F4, and F5 with a structure name of FIELDS.
35. Show how field F3 of the data structure constructed in question 34 is addressed in a program.
36. What are the three program memory-addressing modes?
37. How many bytes of memory store a far direct jump instruction? What is stored in each of the bytes?
38. What is the difference between an intersegment and intrasegment jump?
39. If a near jump uses a signed 16-bit displacement, how can it jump to any memory location within the current code segment?
40. The 80386 and above use a _____-bit displacement to jump to any location within the 4G byte code segment.
41. What is a far jump?
42. If a JMP instruction is stored at memory location 100H within the current code segment, it cannot be a _____ jump if it is jumping to memory location 200H within the current code segment.
43. Show which JMP instruction assembles (short, near, or far) if the JMP THERE instruction is stored at memory address 10000H and the address of THERE is:
    (a) 10020H
    (b) 11000H
    (c) 0FFFEH
    (d) 30000H
44. Form a JMP instruction that jumps to the address pointed to by the BX register.
45. Select a JMP instruction that jumps to the location stored in memory at the location table. Assume that it is a near JMP.

46. How many bytes are stored on the stack by PUSH instructions?
47. Explain how the PUSH [DI] instruction functions.
48. What registers are placed on the stack by the PUSHA instruction? In what order?
49. What does the PUSHAD instruction accomplish?
50. Which instruction places the EFLAGS on the stack in the Pentium 4 microprocessor?

CHAPTER 4

Data Movement Instructions

INTRODUCTION

This chapter concentrates on the data movement instructions. The data movement instructions include MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LAHF, SAHF, and the string instructions MOVS, LODS, STOS. The latest data transfer instruction implemented on the Pentium Pro through Pentium 4 is the CMOV (conditional move) instruction. The data movement instructions are presented first because they are more commonly used in programs and easy to understand.

The microprocessor requires an assembler program, which generates machine language, because machine language instructions are too complex to efficiently generate by hand. This chapter describes the assembly l-anguage syntax and some of its directives. [This text assumes that the user is developing software on an IBM personal computer or clone. It is recommended that the Microsoft MACRO assembler (MASM) be used as the development tool, but the Intel Assembler (ASM), Borland Turbo assembler (TASM), or similar software function equally as well. The most recent version of TASM completely emulates the MASM program. This text presents information that functions with the Microsoft MASM assembler, but most programs assemble without modification with other assemblers. Appendix A explains the Microsoft assembler and provides detail on the linker program and Programmer's WorkBench.] Those interested only in 8086 may ignore text in italics.

CHAPTER OBJECTIVES

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

1. Explain the operation of each data movement instruction with applicable addressing modes.
2. Explain the purposes of the assembly language pseudo-operations and key words such as ALIGN, ASSUME, DB, DD, DW, END, ENDS, ENDP, EQU, .MODEL, OFFSET, ORG, PROC, PTR, SEGMENT.
3. Select the appropriate assembly language instruction to accomplish a specific data movement task.
4. Determine the symbolic opcode, source, destination, and addressing mode for a hexadecimal machine language instruction.
5. Use the assembler to set up a data segment, stack segment, and code segment.
6. Show how to set up a procedure using PROC and ENDP.
7. Explain the difference between memory models and full-segment definitions for the MASM assembler.

4–1 MOV REVISITED

The MOV instruction, introduced in Chapter 3, explains the diversity of 8086–Pentium 4 addressing modes. In this chapter, the MOV instruction introduces the machine language instructions available with various addressing modes and instructions. Machine code is introduced because it may occasionally be necessary to interpret machine language programs generated by an assembler. Interpretation of the machine's native language (**machine language**) allows debugging or modification at the machine language level. Occasionally, machine language patches are made by using the DEBUG program available with DOS, which requires some knowledge of machine language. Conversion between machine and assembly language instructions is illustrated in Appendix B.

Machine Language

Machine language is the **native binary code** that the microprocessor understands and uses as its instructions to control its operation. Machine language instructions for the 8086 through the Pentium 4 vary in length from one to as many as thirteen bytes. Although machine language appears complex, there is order to this microprocessor's machine language. There are well over 100,000 variations of machine language instructions, which means that there is no complete list of these variations. Because of this, some binary bits in a machine language instruction are given, and the remainder are determined for each variation of the instruction.

Instructions for the 8086 through the 80286 are 16-bit mode instructions that take the form found in Figure 4–1(a). The 16-bit mode instructions are compatible with the 80386 and above if they are programmed to operate in the 16-bit instruction mode, but they may be prefixed, as shown in Figure 4–1(b). The 80386 and above assume that all instructions are 16-bit mode instructions when the machine is operated in the *real mode.* In the *protected mode,* the upper byte of the descriptor contains the D-bit that selects either the 16- or 32-bit instruction mode. At present, only Windows XP, Windows 95, Windows 98, and OS/2 operate in the 32-bit instruction mode. The 32-bit mode instructions are in the form shown in Figure 4–1(b). These instructions occur in the 16-bit instruction mode by the use of prefixes, which are explained later in this chapter.

The first two bytes of the 32-bit instruction mode format are called **override prefixes** because they are not always present. The first modifies the size of the operand address used by the instruction and the second modifies the register size. If the 80386 through the Pentium II operate as 16-bit instruction mode machines (real or protected mode) and a 32-bit register is used, the **register-size prefix** (66H) is appended to the front of the instruction. If operated in the 32-bit instruction mode (protected mode only) and a 32-bit register is used, the register-size prefix is absent. If a 16-bit register appears in an instruction in the 32-bit instruction mode, the register-size prefix is present to select a 16-bit register. The **address size-prefix** (67H) is used in a similar fashion, as explained later in this chapter. The prefixes toggle the size of the register and operand address from 16-bit to 32-bit or from 32-bit to 16-bit for the prefixed instruction. Note that the 16-bit instruction mode uses 8- and 16-bit registers and addressing

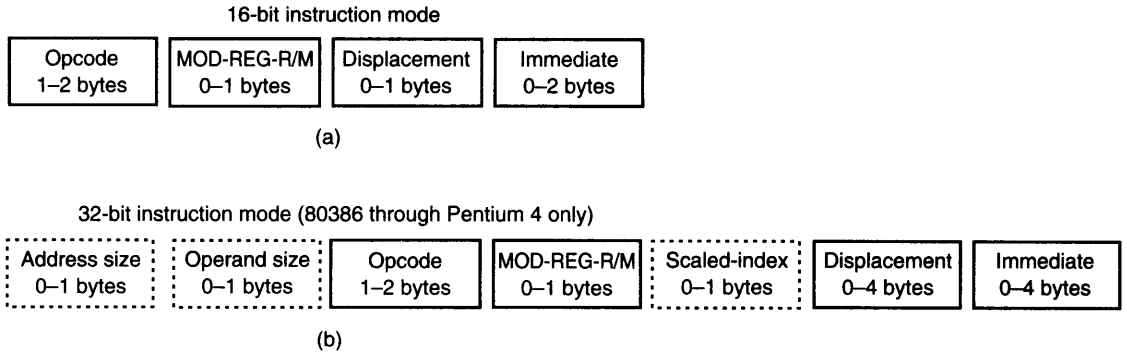


FIGURE 4–1 The formats of the 8086–Pentium 4 instructions. (a) The 16-bit form and (b) the 32-bit form.

modes, while the 32-bit instruction mode uses 8- and 32-bit registers and addressing modes by default. The prefixes override these defaults so that a 32-bit register can be used in the 16-bit mode or a 16-bit register can be used in the 32-bit mode. The **mode of operation** (16 or 32 bits) should be selected to conform with the application at hand. If 8- and 32-bit data pervade the application, the 32-bit mode should be selected; likewise, if 8- and 16-bit data pervade, the 16-bit mode should be selected. Normally, mode selection is a function of the operating system. (Remember that DOS can operate only in the 16-bit mode, however.)

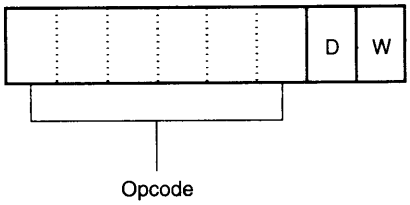


FIGURE 4–2 Byte 1 of many machine language instructions, showing the position of the D- and W-bits.

The Opcode. The **opcode** selects the operation (addition, subtraction, move, and so on) that is performed by the microprocessor. The opcode is either one or two bytes long for most machine language instructions. Figure 4–2 illustrates the general form of the first opcode byte of many, but *not* all, machine language instructions. Here, the first six bits of the first byte are the binary opcode. The remaining two bits indicate the **direction** (D)—not to be confused with the instruction mode bit (16/32) or direction flag bit (used with string instructions)—of the data flow, and indicate whether the data are a byte or a word (W). In the 80386 and above, words and doublewords are both specified when W = 1. The instruction mode and register-size prefix (66H) determine whether W represents a word or a doubleword.

If the direction bit (D) = 1, data flow *to* the register REG field from the R/M field located in the second byte of an instruction. If the D-bit = 0 in the opcode, data flow to the R/M field *from* the REG field. If the W-bit = 1, the data size is a *word* or *doubleword;* if the W-bit = 0, the data size is always a *byte.* The W-bit appears in most instructions, while the D-bit appears mainly with the MOV and some other instructions. Refer to Figure 4–3 for the binary bit pattern of the second opcode byte (reg-mod-r/m) of many instructions. Figure 4–3 shows the location of the MOD (mode), REG (register), and R/M (register/memory) fields.

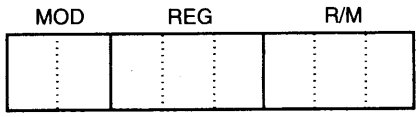


FIGURE 4–3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.

MOD Field. The MOD field specifies the addressing mode (MOD) for the selected instruction. The MOD field selects the type of addressing and whether a displacement is present with the selected type. Table 4–1 lists the operand forms available to the MOD field for 16-bit instruction mode, unless the operand address-size override prefix (67H) appears. If the MOD field contains an 11, it selects the register addressing mode. Register addressing uses the R/M field to specify a register instead of a memory location. If the MOD field contains a 00, 01, or 10, the R/M field selects one of the data memory-addressing modes. When MOD selects a data memory addressing mode, it indicates that the addressing mode contains no displacement (00), an 8-bit sign-extended displacement (01), or a 16-bit displacement (10). The MOV AL,[DI] instruction is an example that shows no displacement, a MOV AL,[DI+2] instruction uses an 8-bit displacement (+ 2), and a MOV AL,[DI+1000H] instruction uses a 16-bit displacement (+ 1000H).

All 8-bit displacements are **sign-extended** into 16-bit displacements when the microprocessor 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. To sign-extend a number, its sign-bit is copied to the next higher-order byte, which generates either a 00H or an FFH in the next higher-order byte. Some assembler programs do not use the 8-bit displacements.

In the 80386 through the Pentium 4 microprocessors, the MOD field may be the same as shown in Table 4–1; if the instruction mode is 32-bits, the MOD field is as it appears in Table 4–2. The MOD field is interpreted as selected by the address-size override prefix or the operating mode of the microprocessor. This change in the

TABLE 4–1 MOD field for the 16-bit instruction mode.

MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	16-bit displacement
11	R/M is a register

TABLE 4–2 MOD field for the 32-bit instruction mode (80386–Pentium 4 only).

MOD	*Function*
00	No displacement
01	8-bit sign-extended displacement
10	32-bit displacement
11	R/M is a register

interpretation of the MOD field and instruction supports many of the numerous additional addressing modes allowed in the 80386 through the Pentium 4. The main difference is that when the MOD field is a 10, this causes the 16-bit displacement to become a 32-bit displacement to allow any protected mode memory location (4G bytes) to be accessed. The 80386 and above only allow an 8- or 32-bit displacement when operated in the 32-bit instruction mode, unless the address-size override prefix appears. Note that if an 8-bit displacement is selected, it is sign-extended into a 32-bit displacement by the microprocessor.

Register Assignments. Table 4–3 lists the register assignments for the REG field and the R/M field (MOD = 11). This table contains three lists of register assignments: one is used when the W-bit = 0 (bytes), and the other two are used when the W-bit = 1 (words or doublewords). Note that doubleword registers are only available to the 80386 through the Pentium 4.

TABLE 4–3 REG and R/M (when MOD = 11) assignments.

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

Suppose that a 2-byte instruction, 8BECH, appears in a machine language program. Because neither a 67H (operand address-size override prefix) nor a 66H (register-size override prefix) appears as the first byte, the first byte is the opcode. If the microprocessor is operated in the 16-bit instruction mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2, as illustrated in Figure 4–4. The opcode is 100010. If you refer to Appendix B, which lists the machine language instructions, you will find that this is the opcode for a MOV instruction. Notice that both the D and W bits are a logic 1, which means that a word moves into the destination register specified in the REG field. The REG field contains a 101, indicating register BP, so the MOV instruction moves data into register BP.

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

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

Because the MOD field contains a 11, the R/M field also indicates a register. Here, R/M = 100 (SP); therefore, this instruction moves data from SP into BP and is written in symbolic form as a MOV BP,SP instruction.

Suppose that a 668BE8H instruction appears in an 80386 or above, operated in the 16-bit instruction mode. The first byte (66H) is the register-size override prefix that selects 32-bit register operands for the 16-bit instruction mode. The remainder of the instruction indicates that the opcode is a MOV with a source operand of EAX and a destination operand of EBP. This instruction is a MOV EBP,EAX. The same instruction becomes a MOV BP,AX instruction in the 80386 and above if it is operated in the 32-bit instruction mode because the register-size override prefix selects a 16-bit register. Luckily, the assembler program keeps track of the register- and address-size prefixes and the mode of operation. Recall that if the .386 switch is placed before the .MODEL statement, the 32-bit mode is selected; if it is placed after the .MODEL statement, the 16-bit mode is selected.

R/M Memory Addressing. If the MOD field contains a 00, 01, or 10, the R/M field takes on a new meaning. Table 4–4 lists the memory-addressing modes for the R/M field when MOD is a 00, 01, or 10 for the 16-bit instruction mode.

All of the 16-bit addressing modes presented in Chapter 3 appear in Table 4–4. The displacement, discussed in Chapter 3, is defined by the MOD field. If MOD = 00 and R/M = 101, the addressing mode is [DI]. If MOD = 01 or 10, the addressing mode is [DI+33H], or LIST [DI+22H] for the 16-bit instruction mode. This example uses LIST, 33H, and 22H as arbitrary values for the displacement.

TABLE 4–4 16-bit R/M memory-addressing modes.

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

**Note:* See text section, Special Addressing Mode.

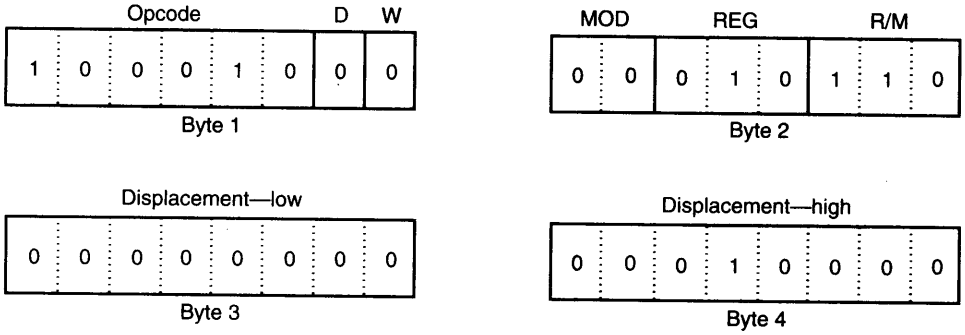
Figure 4–5 illustrates the machine language version of the 16-bit instruction MOV DL,[DI] or instruction (8A15H). This instruction is two bytes long and has an opcode 100010, D = 1 (to REG from R/M), W = 0 (byte), MOD = 00 (no displacement), REG = 010 (DL), and R/M = 101 ([DI]). If the instruction changes to MOV DL,[DI+1], the MOD field changes to 01 for an 8-bit displacement, but the first two bytes of the instruction otherwise remain the same. The instruction now becomes 8A5501H instead of 8A15H. Notice that the 8-bit displacement appends to the first two bytes of the instruction to form a three-byte instruction instead of two bytes. If the instruction is again changed to a MOV DL,[DI+1000H], the machine language form becomes a 8A750010H. Here, the 16-bit displacement of 1000H (coded as 0010H) appends the opcode.

Special Addressing Mode. There is a special addressing mode that does not appear in Tables 4–2, 4–3, or 4–4. It occurs whenever 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. The first instruction moves the contents of register DL into data segment memory location 1000H. The second instruction moves register DL into symbolic data segment memory location NUMB.

Whenever an instruction has only a displacement, the MOD field is always a 00 and the R/M field is always a 110. As shown in the tables, the instruction contains no displacement and uses addressing mode [BP]. You

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

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



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

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

cannot actually use addressing mode [BP] without a displacement in machine language. The assembler takes care of this by using an 8-bit displacement (MOD = 01) of 00H whenever the [BP] addressing mode appears in an instruction. This means that the [BP] addressing mode assembles as a [BP+0], even though a [BP] is used in the instruction. The same special addressing mode is also available to the 32-bit mode.

Figure 4–6 shows the binary bit pattern required to encode the MOV [1000H],DL instruction 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 4–7 shows the actual form of the MOV [BP],DL instruction. Notice that this is a three-byte instruction with a displacement of 00H.

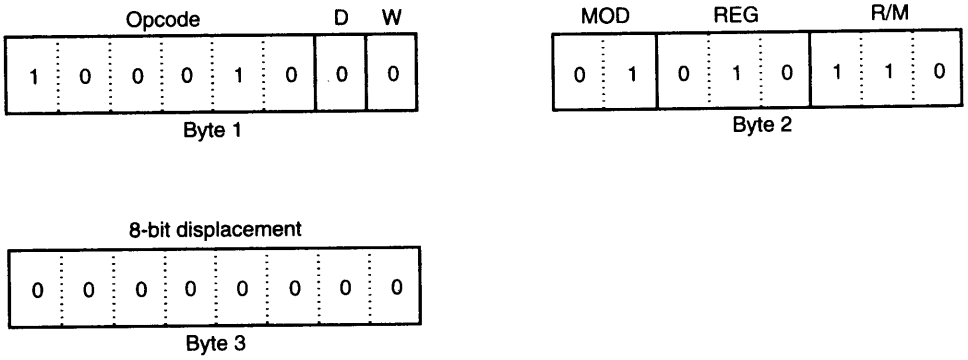
32-bit Addressing Modes. The 32-bit addressing modes found in the 80386 and above are obtained by either running these machines in the 32-bit instruction mode or in the 16-bit instruction mode by using the address-size prefix 67H. Table 4–5 shows the coding for R/M used to specify the 32-bit addressing modes. Notice that when R/M = 100, an additional byte appears in the instruction called a **scaled-index byte.** The scaled-index byte indicates the additional forms of scaled-index addressing that do not appear in Table 4–5. The scaled-index byte is mainly used when two registers are added to specify the memory address in an instruction. Because the scaled-index byte is added to the instruction, there are seven bits in the opcode and eight bits in the scaled-index byte to define. This means that a scaled-index instruction has 2^{15} (32K) possible combinations. There are over 32,000 different variations of the MOV instruction alone in the 80386 through the Pentium 4 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

TABLE 4–5 32-bit addressing modes selected by R/M.

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

**Note:* See text section, Special Addressing Mode.

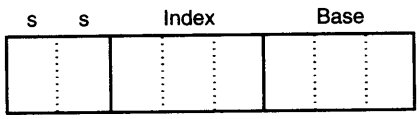


Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 00H

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

the 80386 and above use a 32-bit address. The leftmost two bits select a scaling factor (multiplier) of 1X, 2X, 4X, or 8X. Note that a scaling factor of 1X is implicit if none is used in an instruction that contains two 32-bit indirect address registers. The index and base fields both contain register numbers, as indicated in Table 4–3 for 32-bit registers.

The instruction MOV EAX,[EBX+4*ECX] is encoded as 67668B048BH. Notice that both the *address size* (67H) and *register size* (66H) override prefixes appear in the instruction. This coding (67668B048BH) is used when the 80386 and above microprocessors are operated in the 16-bit instruction mode for this instruction. If the microprocessor operates in the 32-bit instruction mode, both prefixes disappear and the instruction becomes an 8B048BH instruction. The use of the prefixes depends on the mode of operation of the microprocessor. Scaled-index addressing can also use a single register multiplied by a scaling factor. An example is the MOV AL,[2*ECX] instruction. The contents of the data segment location addressed by two times ECX is copied into AL.

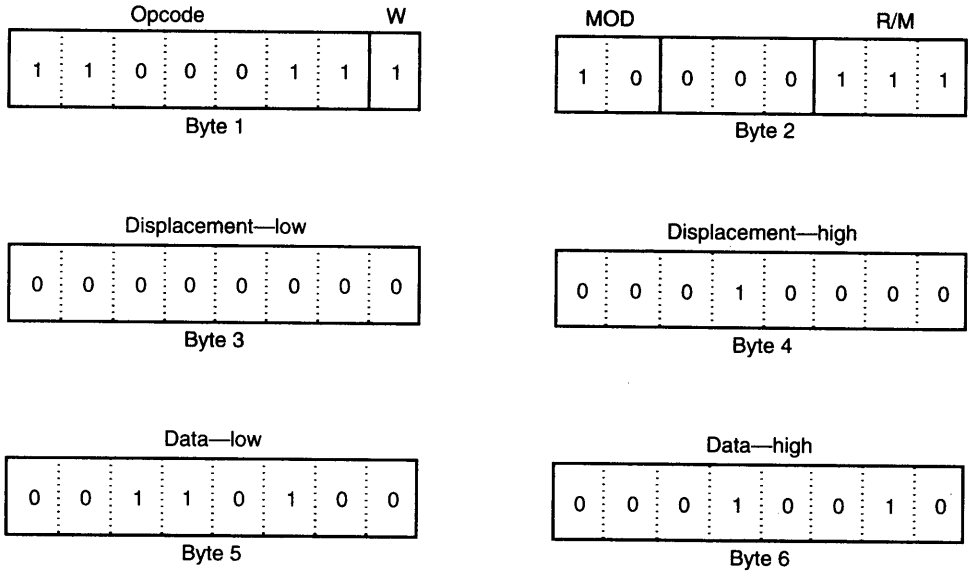


ss
00 = × 1
01 = × 2
10 = × 4
11 = × 8

FIGURE 4–8 The scaled-index byte.

An Immediate Instruction. Suppose that the MOV WORD PTR [BX+1000H],1234H instruction is chosen as an example of a 16-bit instruction using immediate addressing. This instruction moves a 1234H into the word-sized memory location addressed by the sum of 1000H, BX, and DS × 10H. This six-byte instruction uses two bytes for the opcode, W, MOD, and R/M fields. Two of the six bytes are the data of 1234H; two of the six bytes are the displacement of 1000H. Figure 4–9 shows the binary bit pattern for each byte of this instruction.

This instruction, in symbolic form, includes WORD PTR. The 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 in the instruction. Likewise, if the instruction uses a doubleword of immediate data, the DWORD PTR directive replaces BYTE PTR. Most instructions that refer to memory through a pointer do not need the BYTE PTR, WORD PTR, or DWORD PTR directives. These directives are necessary only when it is



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

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

not clear whether the operation is a byte or a word. The MOV [BX],AL instruction is clearly a byte move; the MOV [BX],1 instruction is not exact, and could therefore be a byte-, word-, or doubleword-sized move. Here, the instruction must be coded as MOV BYTE PTR [BX],1, MOV WORD PTR [BX],1, or MOV DWORD PTR [BX],1. If not, the assembler flags it as an error because it cannot determine the intent of this instruction.

Segment MOV Instructions. If the contents of a segment register are moved by the MOV, PUSH, or POP instructions, a special set of register bits (REG field) selects the segment register (see Table 4–6).

Figure 4–10 shows a MOV BX,CS instruction converted to binary. The opcode for this type of MOV instruction is different for the prior MOV instructions. Segment registers can be moved between any 16-bit register or 16-bit memory location. For example, the MOV [DI],DS instruction stores the contents of DS into the memory location addressed by DI in the data segment. 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 then move it to a segment register.

Although this discussion has not been a complete coverage of machine language coding, it should give you a good start in machine language programming. Remember 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. With the microprocessor and its over 100,000 instruction variations, let us hope that an assembler is available for the conversion because the process is very time-consuming, although not impossible.

TABLE 4–6 Segment register selection.

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

**Note:* MOV CS,R/M(16) and POP CS are not allowed by the microprocessor. The FS and GS segments are only available to the 80386–Pentium 4 microprocessors.

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

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

4–2 PUSH/POP

The PUSH and POP instructions are important instructions that *store* and *retrieve* data from the LIFO (last-in, first-out) stack memory. The microprocessor has six forms of the PUSH and POP instructions: register, memory, immediate, segment register, flags, and all registers. The PUSH and POP immediate and the PUSHA and POPA (all registers) forms are not available in the earlier 8086/8088 microprocessors, but are available to the 80286 through the Pentium 4.

Register addressing allows the contents of any 16-bit register to be transferred to or from the stack. In the 80386 and above, the 32-bit extended registers and flags (EFLAGS) can also be pushed or popped from the stack. Memory addressing PUSH and POP instructions store the contents of a 16-bit memory location (or 32-bits in the 80386 and above) 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 the contents of any segment register to be pushed onto the stack or removed from the stack (CS may be pushed, but data from the stack may never be popped into CS). The flags may be pushed or popped from that stack, and the contents of all the registers may be pushed or popped.

Push

The 8086–80286 PUSH instruction always transfers two bytes of data to the stack; the 80386 and above transfer two or four bytes, depending on the register or size of the memory location. The source of the data may be any internal 16- or 32-bit register, immediate data, any segment register, or any two bytes of memory data. There is also

a PUSHA instruction that copies the contents of the internal register set, except the segment registers, to the stack. The PUSHA (**push all**) instruction copies the registers to the stack in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The value for SP that is pushed onto the stack is whatever it was before the PUSHA instruction executes. The PUSHF (**push flags**) instruction copies the contents of the flag register to the stack. The PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set found in the 80386 through the Pentium 4.

Whenever data are pushed onto the stack, the first (most-significant) data byte moves into the stack segment memory location addressed by SP – 1. The second (least-significant) data byte moves into the stack segment memory location addressed by SP – 2. After the data are stored by a PUSH, the contents of the SP register decrement by 2. The same is true for a doubleword push, except that four bytes are moved to the stack memory (most-significant byte first), after which the stack pointer decrements by 4. Figure 4–11 shows the operation of the PUSH AX instruction. This instruction copies the contents of AX onto the stack where address SS:[SP – 1] = AH, SS:[SP – 2] = AL, and afterwards SP = SP – 2.

The PUSHA instruction pushes all the internal 16-bit registers onto the stack, as illustrated in Figure 4–12. This instruction 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. The PUSHA instruction is very useful when the entire register set (microprocessor environment) of the 80286 and above must be saved during a task. The PUSHAD instruction places the 32-bit register set on the stack in the 80386 through the Pentium 4. PUSHAD requires 32 bytes of stack storage space.

The PUSH immediate data instruction has two different opcodes, but in both cases, a 16-bit immediate number moves onto the stack; if PUSHD is used, a 32-bit immediate datum is pushed. If the value of the immediate data are 00H–FFH, the opcode is a 6AH; if the data are 0100H–FFFFH, the opcode is 68H. The PUSH 8 in-

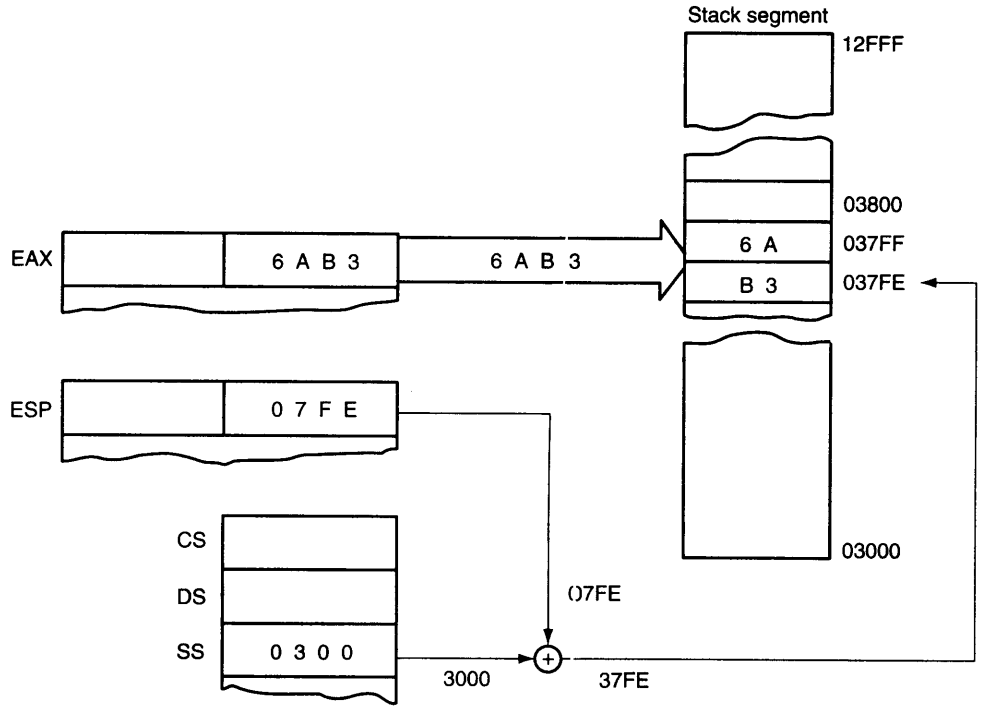


FIGURE 4–11 The effect of the PUSH AX instruction on ESP and stack memory location 37FFH and 37FEH. This instruction is shown at the point after execution.

struction, which pushes a 0008H onto the stack, assembles as a 6A08H. The PUSH 1000H instruction assembles as 680010H. Another example of PUSH immediate is the PUSH 'A' instruction, which pushes a 0041H onto the stack. Here, the 41H is the ASCII code for the letter A.

Table 4–7 lists the forms of the PUSH instruction that include PUSHA and PUSHF. Notice how the instruction set is used to specify different data sizes with the assembler.

Pop

The POP instruction performs the inverse operation of a PUSH instruction. The POP instruction removes data from the stack and places it into the target 16-bit register, segment register, or a 16-bit memory location. In the 80386 and above, a POP can also remove 32-bit data from the stack and use a 32-bit address. The POP instruction is not available as an immediate POP. The POPF (**pop flags**) instruction removes a 16-bit number from the stack and places it into the flag register; the POPFD removes a 32-bit number from the stack and places it into the extended flag register. The POPA (**pop all**) instruction removes 16 bytes of data from the stack and places it into the following registers, in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX. This is the reverse order from the way they were placed on the stack by the PUSHA instruction, causing the same data to return to the same registers. In the 80386 and above, a POPAD instruction reloads the 32-bit registers from the stack.

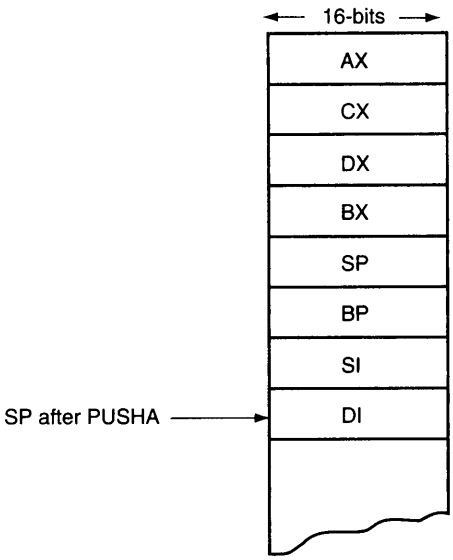


FIGURE 4–12 The operation of the PUSHA instruction, showing the location and order of stack data.

Suppose that a POP BX instruction executes. The first byte of data removed from the stack (the memory location addressed by SP in the stack segment) moves into register BL. The second byte is removed from stack segment memory location SP + 1 and is placed into register BH. After both bytes are removed from the stack, the SP register increments by 2. Figure 4–13 shows how the POP BX instruction removes data from the stack and places them into register BX.

The opcodes used for the POP instruction and all of its variations appear in Table 4–8. Note that a POP CS instruction is not a valid instruction in the instruction set. If a POP CS instruction executes, only a portion of the

TABLE 4–7 The PUSH instructions.

Symbolic	*Example*	*Note*
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR [BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR [EBX]	32-bit pointer
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH ','	8-bit immediate
PUSHW imm16	PUSHW 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGs

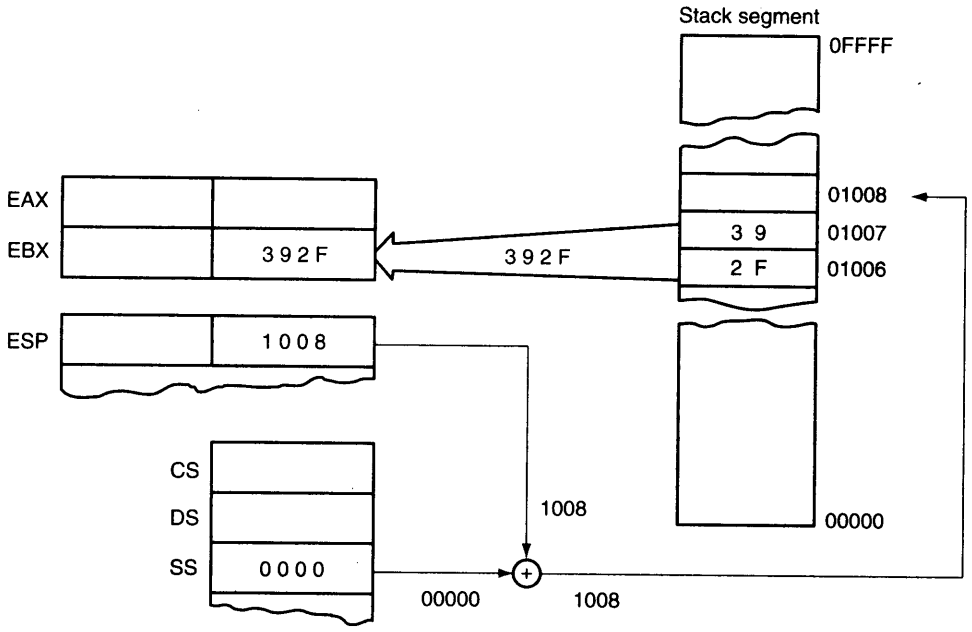


FIGURE 4–13 The POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.

TABLE 4–8 The POP instructions.

Symbolic	*Example*	*Note*
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP seg	POP FS	Segment register
POPA	POPA	Pop all 16-bit registers
POPAD	POPAD	Pop all 32-bit registers
POPF	POPF	Pop flags
POPFD	POPFD	Pop EFLAGs

address (CS) of the next instruction changes. This makes the POP CS instruction unpredictable and therefore not allowed.

Initializing the Stack

When the stack area is initialized, load both the stack segment (SS) register and the stack pointer (SP) register. It is normal to designate an area of memory as the stack segment by loading SS with the bottom location of the stack segment.

For example, if the stack segment is to reside in memory locations 10000H–1FFFFH, load SS with a 1000H. (Recall that the rightmost end of the stack segment register is appended with a 0H for real mode addressing.) To start the stack at the top of this 64K-byte stack segment, the stack pointer (SP) is loaded with a

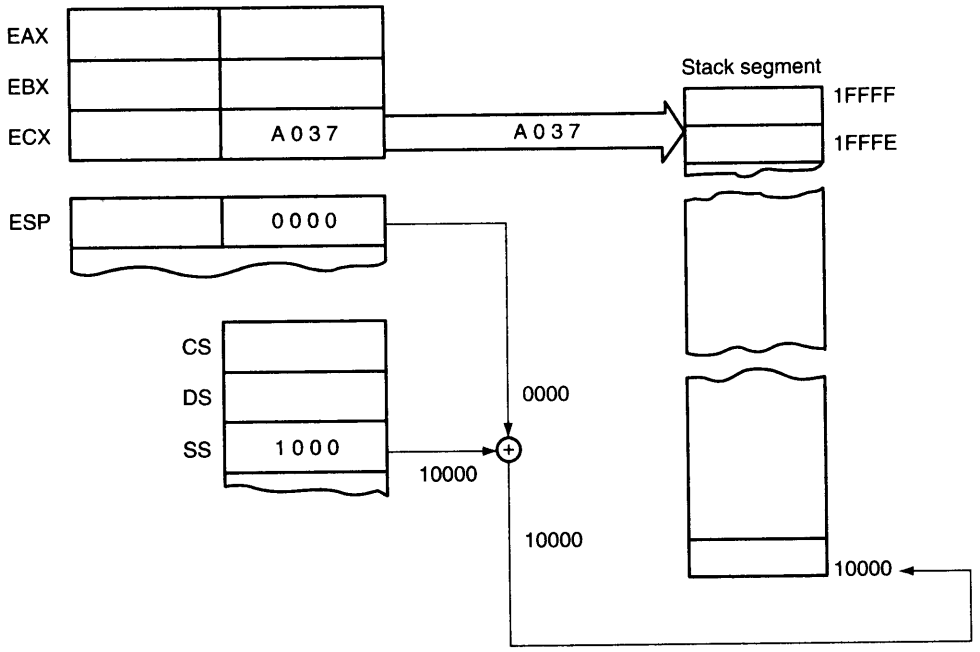


FIGURE 4–14 The PUSH CX instruction, showing the cyclical nature of the stack segment. This instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.

0000H. Likewise, to address the top of the stack at location 10FFFH, use a value of 1000H in SP. Figure 4–14 shows how this value causes data to be pushed onto the top of the stack segment with a PUSH CX instruction. Remember that all segments are cyclic in nature—that is, the top location of a segment is contiguous with the bottom location of the segment.

In assembly language, a stack segment is set up as illustrated in Example 4–1. The first statement identifies the start of the stack segment and the last statement identifies the end of the stack segment. The assembler and linker programs place the 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.

EXAMPLE 4–1

```
0000                    STACK_SEG       SEGMENT STACK

0000 0100[                      DW      100H DUP (?)
             ????
                    ]
0200                    STACK_SEG       ENDS
```

An alternative method for defining the stack segment is used with one of the memory models for the MASM assembler only (refer to Appendix A). Other assemblers do not use models; if they do, the models are not exactly the same as with MASM. Here, the .STACK statement, followed by the number of bytes allocated to the stack, defines the stack area (see Example 4–2). The function is identical to Example 4–1. The .STACK statement also initializes both SS and SP. Note that this text uses memory models that are designed for the Microsoft Macro Assembler program MASM.