

3-1

DATA-ADDRESSING MODES

Because the MOV instruction is a very 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.

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 always changes.¹ It is crucial to remember that a MOV instruction always *copies* the source data into the destination. The MOV never actually picks up the data and moves it. Also, note 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 on data-addressing modes. 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 Core2. The RIP relative addressing mode is not illustrated and is only available on the Pentium 4 and the Core2 when operated in the 64-bit mode. The data-addressing modes are as follows:

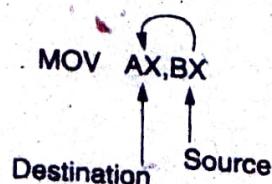
Register addressing

Register addressing transfers a copy of a byte or word from the source register or contents of a 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.) In the Pentium 4 operated in the 64-bit mode, any 64-bit register is also allowed. An example is the MOV RDX, RCX instruction that transfers a copy of the quadword contents of register RCX into register RDX.

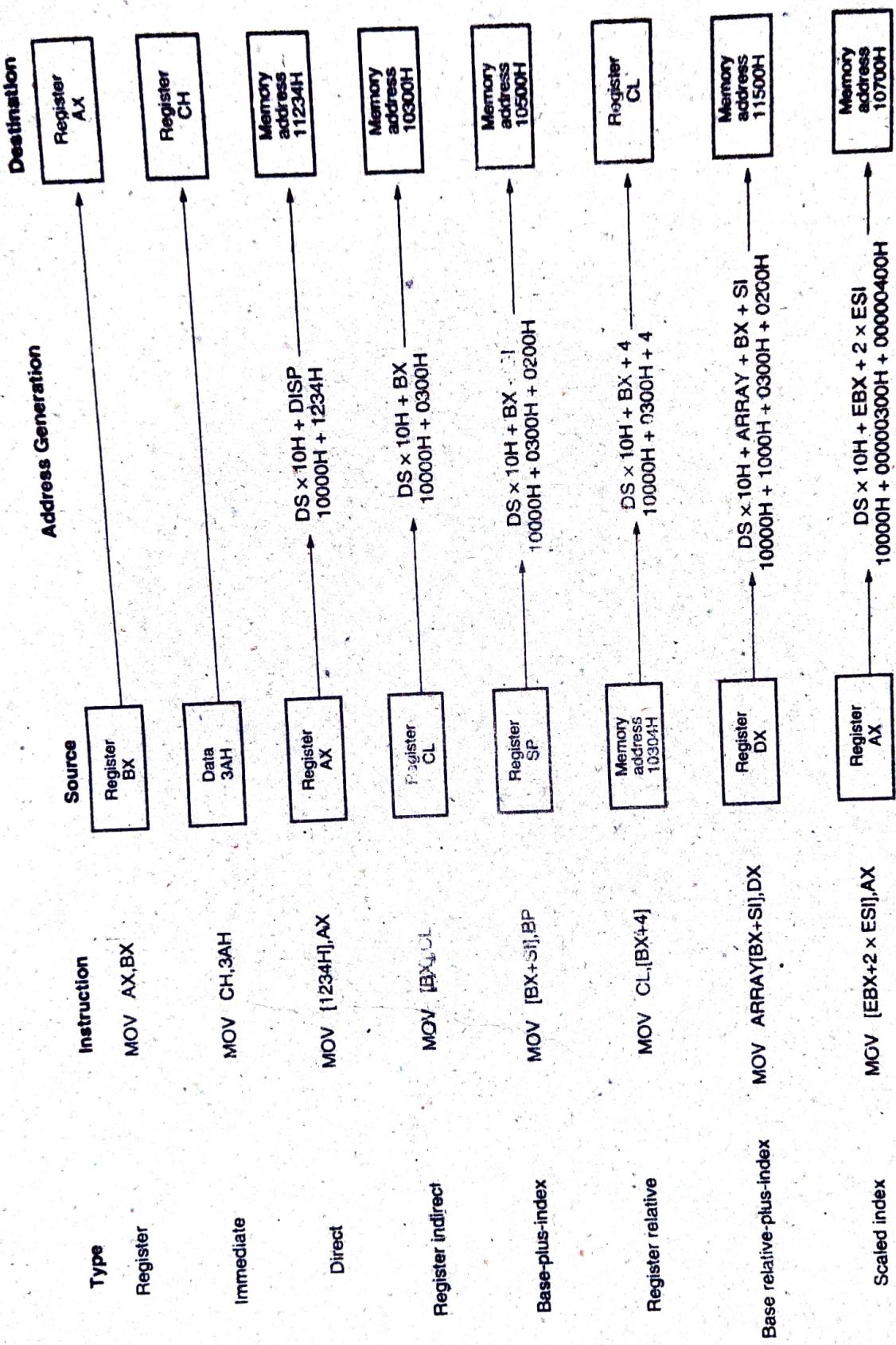
Immediate addressing

Immediate addressing transfers the source, an immediate byte, word, doubleword, or quadword 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

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



¹The 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-Core2 data-addressing modes.

Direct addressing

memory location. (Example: The MOV EBX, 12345678H instruction copies a doubleword-sized 12345678H into the 32-bit-wide EBX register.) In 64-bit operation of the Pentium 4 or Core2, only a MOV immediate instruction allows access to any location in the memory using a 64-bit linear address.

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 with 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.) The direct memory instructions in the 64-bit mode use a full 64-bit linear address.

Register indirect addressing

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 byte, word, or doubleword 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.) In 64-bit mode, the indirect address remains 32 bits in size, which means this form of addressing at present only allows access to 4G bytes of address space if the program operates in the 32-bit compatible mode. In the full 64-bit mode, any address is accessed using either a 64-bit address or the address contained in a register.

Base-plus-index addressing

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 two registers (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

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 32-bit register except ESP 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

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

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 $2\times$, $4\times$, or $8\times$ 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 four times EBX.) Scaling allows access to word ($2\times$), doubleword ($4\times$), or quadword ($8\times$) memory array data. Note that a scaling factor of $1\times$ also exists, but it is normally implied and does not appear explicitly 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.

RIP relative addressing

This addressing mode is only available to the 64-bit extensions on the Pentium 4 or Core2. This mode allows access to any location in the memory system by adding a 32-bit displacement to the 64-bit contents of the 64-bit instruction pointer. For example, if RIP = 1000000000H and a 32-bit displacement is 300H, the location accessed is 1000000300H. The displacement is signed so data located within $\pm 2G$ from the instruction is accessible by this addressing mode.

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 register names used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL. Also present are the following 16-bit register names: AX, BX, CX, DX, SP, BP, SI, and DI. In the 80386 and above, the extended 32-bit register names are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. In the 64-bit mode of the Pentium 4, the register names are: RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, and R8 through R15. With register addressing, some MOV instructions and the PUSH and POP instructions also use the 16-bit segment register names (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 a 32-bit register because this is not allowed by the microprocessor and results in an error when assembled. Likewise never mix 64-bit registers with any other size register. This is even true when a MOV AX, AL (MOV EAX, AL) instruction may seem to make sense. Of course, the MOV AX, AL or MOV EAX, AL instructions are not allowed because the 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. The flag bits are normally modified by arithmetic or logic instructions.

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 instruc-

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

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

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

start with a **0**. For example, to represent a hexadecimal F2, 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 letter 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 assembly language program that places 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 with MASM. 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 CodeView⁴ (CV) to view its execution. Note that the most recent version of TASM will also accept MASM code without any changes. To store the program into the system use the DOS EDIT program, Windows NotePad,⁵ or Programmer's WorkBench⁶ (PWB). Note that a TINY program always assembles as a command (.COM) program.

⁴CodeView is a registered trademark of Microsoft Corporation.

⁵NotePad is a registered trademark of Microsoft Corporation.

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

EXAMPLE 3-2

```

0000      .MODEL TINY           ;choose single segment model
          .CODE                ;start of code segment
          .STARTUP              ;start of program
0100 B8 0000    MOV AX,0        ;place 0000H into AX
0103 BB 0000    MOV BX,0        ;place 0000H into BX
0106 B9 0000    MOV CX,0        ;place 0000H into CX
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 program

```

opxnd

Each statement in an assembly language 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 to the right 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

<u>Label</u>	<u>Opcode</u>	<u>Operand</u>	<u>Comment</u>
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 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 with the extension .LST. Note that all programs shown in this text are in the form generated by the assembler.

EXAMPLE 3-4

```

int MyFunction(int temp)
{
    asm
    {
        mov eax,temp
        add eax,20h
        mov temp,eax
    }
    return temp; // return a 32-bit integer
}

```

Programs are also written using the inline assembler in some Visual C++ programs. Example 3-4 shows a function in a Visual C++ program that includes some code written with the inline assembler. This function adds 20H to the number returned by the function. Notice that the assembly code accesses C++ variable temp and all of the assembly code is placed in an `_asm` code block. Many examples in this text are written using the inline assembler within a C++ program.

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. In 64-bit operation, the direct-addressing instructions are also used with a 64-bit linear address, which allows access to any memory location.

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 3 bytes in length.)

The `MOV AL,DATA` instruction, as represented by most assemblers, loads AL from the data segment memory location DATA (1234H). Memory location DATA is a symbolic memory location, while the 1234H is the actual hexadecimal 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) and 10000H (the data segment address of 1000H times 10H) in a system operating in the real mode.

Table 3-3 lists the direct-addressed instructions. These instructions often appear in programs, so Intel decided to make them special 3-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 4 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 4 bytes wide instead of 3. In the 80386 through the Pentium 4,

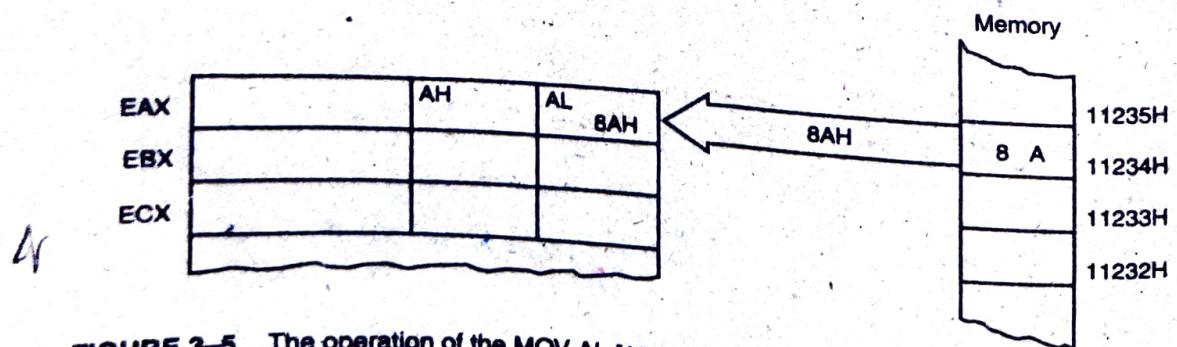


FIGURE 3-5 The operation of the `MOV AL,[1234H]` instruction when `DS = 1000H`.

⁷This form may be typed into a MASM program, but it most often appears when the debugging tool is executed.

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

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

*Note: The 80386-Pentium 4 at times use more than 3 bytes of memory for 32-bit instructions.

this instruction can be up to 7 bytes wide if both 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, we see that 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 3 bytes long and the MOV CL,DS:[1234H] instruction is 4 bytes long, as illustrated in Example 3-5. 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].

EXAMPLE 3-5

0000 A0 1234 R	MOV AL,DS:[1234H]
0003 BA 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 3-6 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 file. 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

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

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

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 3-6

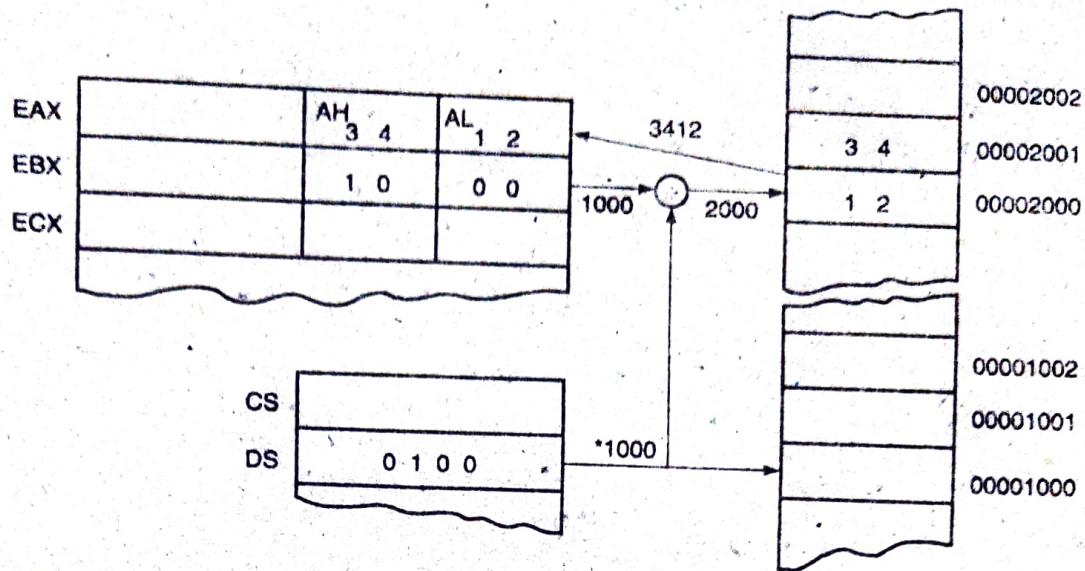
```

0000          .MODEL SMALL      ;choose small model
              .DATA           ;start data segment
0000 10      DATA1 DB 10H    ;place 10H into DATA1
0001 00      DATA2 DB 0     ;place 00H into DATA2
0002 0000    DATA3 DW 0     ;place 0000H into DATA3
0004 AAAA    DATA4 DW 0AAAHH ;place AAAAH into DATA4
0000          .CODE           ;start code segment
              .STARTUP        ;start program
0017 A00000 R MOV AL,DATA1   ;copy DATA1 into AL
001A 8A 26 0001 R MOV AH,DATA2 ;copy DATA2 into AH
001E A3 0002 R  MOV DATA3,AX ;copy AX into DATA3
0021 8B 1E 0004 R  MOV BX,DATA4 ;copy DATA4 into BX
                    .EXIT;
                    END;          exit to DOS
                                end program listing

```

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 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. If a Pentium 4 or Core2 is available that operates in the 64-bit mode, any 64-bit register is used to hold a 64-bit linear address. In the 64-bit mode, the segment registers serve no purpose in addressing a location in the flat model.



*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.

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 fault. When using a 32-bit register to address memory in the real mode, the contents of the 32-bit register must never

TABLE 3–5 Examples of register indirect addressing.

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

*Note: Data addressed by BP or EBP are by default in the **stack segment**, while other indirect addressed instructions use the **data segment** by default.

THE INTEL MICROPROCESSORS

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-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 the 64-bit mode, the segment registers are not used in the address calculation because the register contains the actual linear memory address.

In some cases, indirect addressing requires specifying the size of the data. The size is specified by the **special assembler directive** BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR. These directives indicate the size of the memory data addressed by the memory pointer (PTR). For example, the MOV AL,[DI] instruction is clearly a byte-sized move instruction, but the MOV [DI],10H instruction is ambiguous. Does the MOV [DI],10H instruction address a byte-, word-, doubleword-, or quadword-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, DWORD PTR, and QWORD PTR directives are used only with instructions that address a memory location through a pointer or index register with immediate data, and for a few other instructions that are described in subsequent chapters. Another directive that is occasionally used is the QWORD PTR, where a QWORD is a quadword (64-bits mode). If programs are using the SIMD instructions, the QWORD PTR, an octal word, is also used to represent a 128-bit-wide number.

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 in DOS 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 3-7 loads register BX with the starting address of the table and it 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,DATA\$ instruction copies the contents of memory location DATA\$ into BX, while the MOV BX,OFFSET DATA\$ instruction copies the offset address DATA\$ 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 in the specified 16-bit register.

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

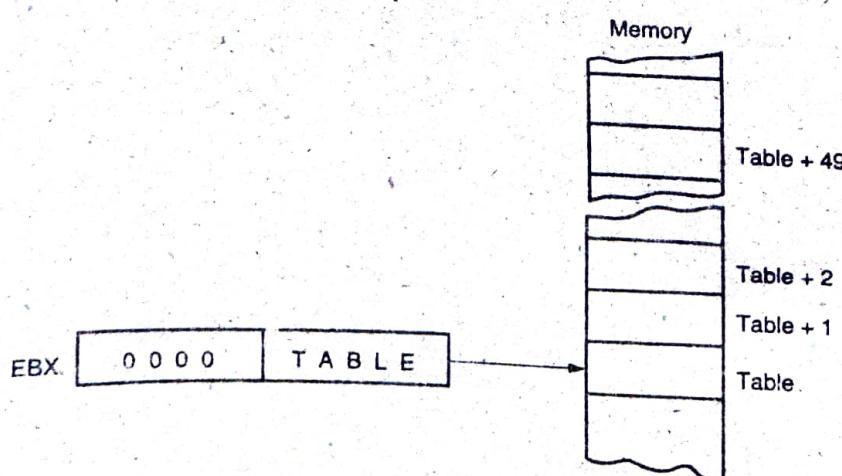


TABLE 3-10 Examples of indirect program memory addressing.

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

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

TABLE	DW	LOC0
	DW	LOC1
	DW	LOC2
	DW	LOC3

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 1000H and a JMP BX instruction executes, the microprocessor jumps to offset address 1000H in the current code segment.

If a relative register holds the address, the jump is also considered to be an indirect jump. For example, JMP [BX] refers to the memory location within the data segment at the offset address contained in BX. At this offset address is a 16-bit number 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 3-14. 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 16-bit-wide jump table.

EXAMPLE 3-14

```
0000 BB 0004
0003 FF A7 23A1 R
```

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

3-3

STACK MEMORY-ADDRESSING MODES

The stack plays an important role in all microprocessors. It holds data temporarily and stores the return addresses used by procedures. The stack memory is an 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

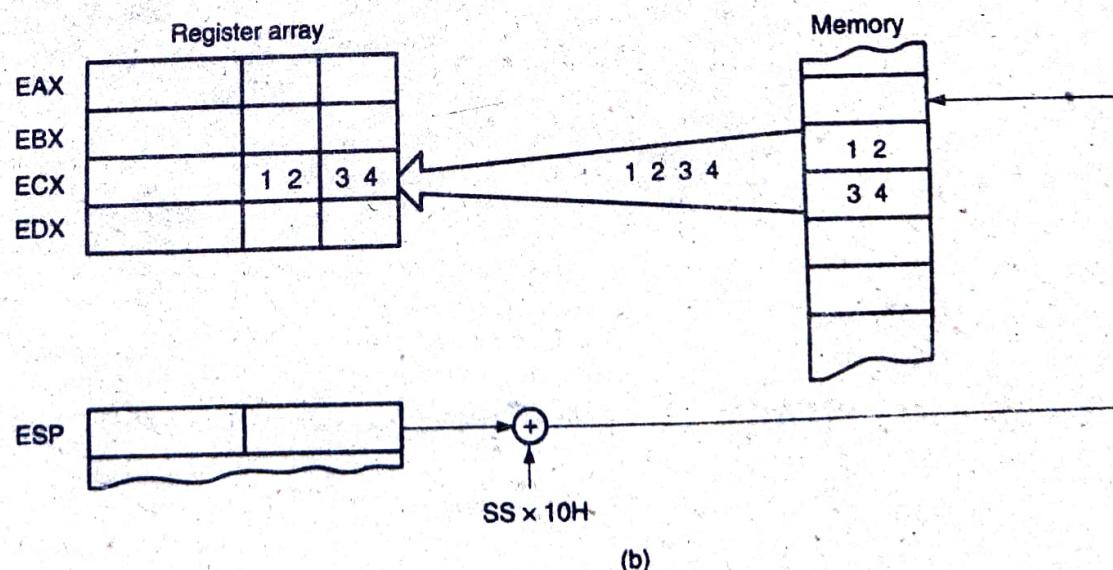
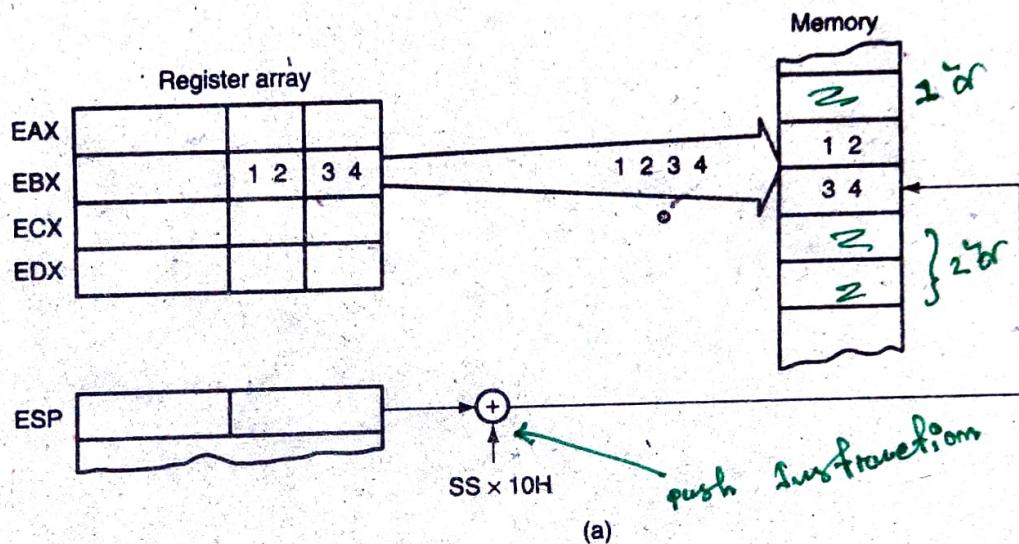


FIGURE 3-17 The **PUSH** and **POP** instructions: (a) **PUSH BX** places the contents of BX onto the stack; (b) **POP CX** removes data from the stack and places them into CX. Both instructions are shown after execution.

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.

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 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 register or any segment register except CS. The reason that data may not be popped from the stack

TABLE 3-11 Example PUSH and POP instructions.

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

into CS is that this only changes part of the address of the next instruction. In the Pentium 4 or Core2 operated in 64-bit mode, the 64-bit registers can be pushed or popped from the stack, but they are 8 bytes in length.

The PUSHA and POPA instructions either push or pop all of the registers, except segment registers, onto the stack. These instructions are not available on the early 8086/8088 processors. The push immediate instruction is also new to the 80286 through the Core2 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. The 64-bit mode for the Pentium 4 and Core2 does not contain a PUSHA or POPA instruction.

Example 3-15 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 value of AX into BX.

EXAMPLE 3-15

	.MODEL TINY	;select tiny model
0000	.CODE	;start code segment
	.STARTUP	;start 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 CBX
010E 5B	POP BX	;1000H to BX
	.exit	;exit to DOS
	end	;end program

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. The 80386 through the Pentium 4 microprocessors have an additional addressing mode called scaled-index addressing.
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 Core2 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 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–Core2 microprocessors.
5. The 64-bit mode for the Pentium 4 and Core2 microprocessors use the same addressing modes as the Pentium 4 or Core2 in 32-bit mode, except the registers contain a linear address