This tutorial taken from Berry B bray book for addressing modes understanding.Read only those parts related to lecture slides.You are not supposed or overburdened to understand anything extra not explained in class.Keep ur cool and enjoy ur learning experience.COA Instructor.

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

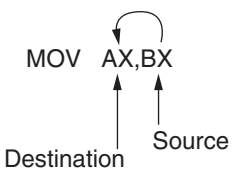
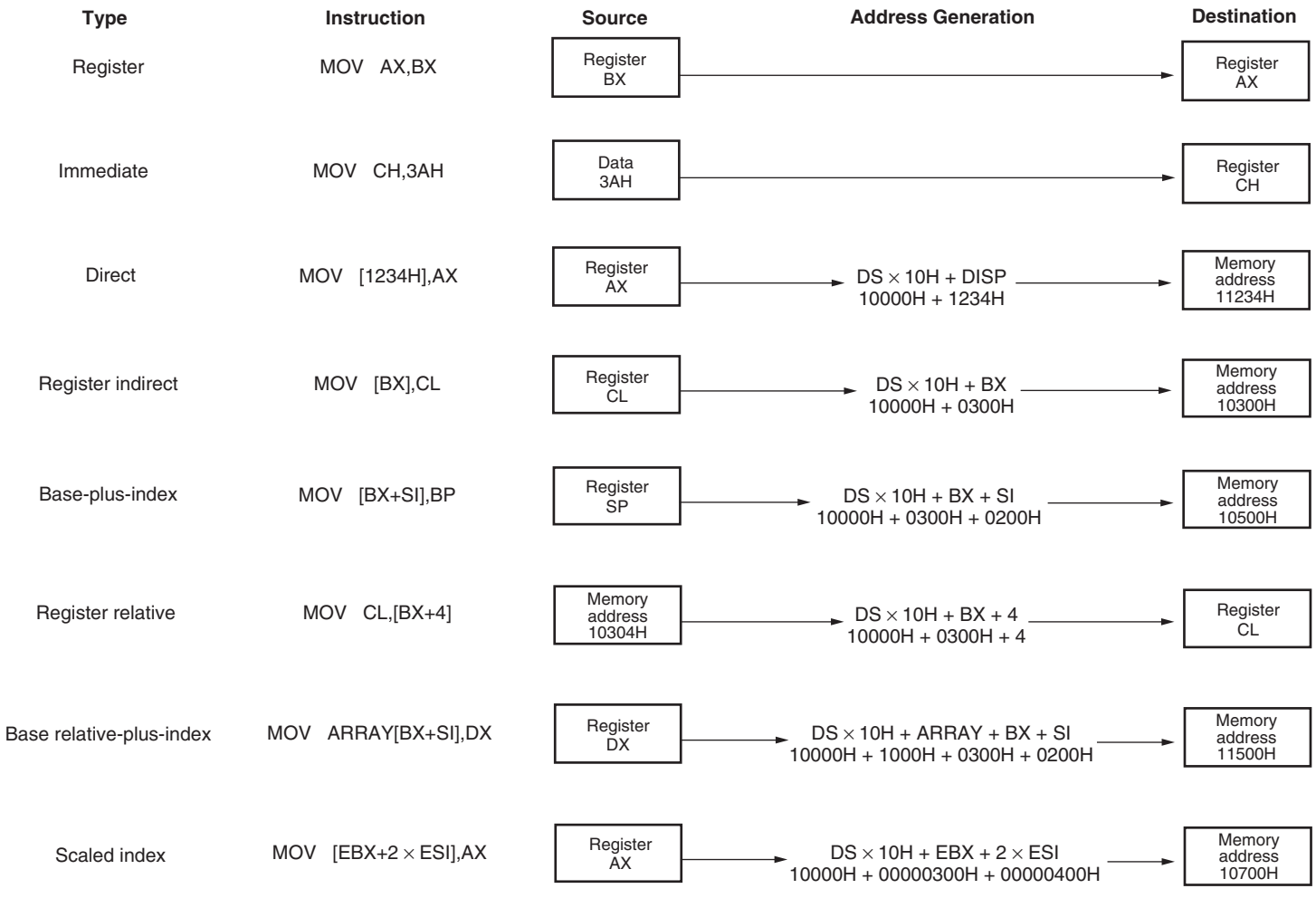


FIGURE 3–1 The MOV instruction showing the source, destination, and direction of data flow.

[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–Core2 data-addressing modes.

memory location. (Example: The MOV EBX, 12345678H instruction copies a doubleword-sized l2345678H 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

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 S1. (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×, 4×, or 8× 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×), doubleword (4×), or quadword (8×) memory array data. Note that a scaling factor of 1× 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 ±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 l6-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 instruction

TABLE 3–1 Examples of 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 R8B,CL	8 bits	Copies CL to the byte portion of R8 (64-bit mode)
MOV R8B,CH	8 bits	Not allowed
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 BP,R10W	16 bits	Copies R10 into BP (64-bit mode)
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 EDX,R9D	32 bits	Copies R9 into EDX (64-bit mode)
MOV RAX,RDX	64 bits	Copies RDX into RAX
MOV DS,CX	16 bits	Copies CX into DS
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)

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 by 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. This instruction moves *(copies)* a l234H 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 except the CMP and TEST instructions. Note that the MOV BX, CX instruction does not affect the leftmost 16 bits of register EBX.

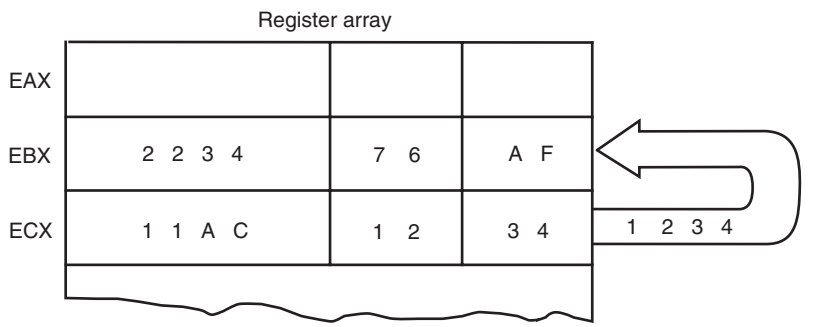


FIGURE 3–3 The effect of executing the MOV BX, CX instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.

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 changes only 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 contents of DH into CL
0004 8A CD          MOV CL,CH     ;copy contents of CH into CL
0006 66|8B C3       MOV EAX,EBX   ;copy contents of EBX into EAX
0009 66|8B D8       MOV EBX,EAX   ;copy contents of EAX into EBX
000C 66|8B C8       MOV ECX,EAX   ;copy contents of EAX into ECX
000F 66|8B D0       MOV EDX,EAX   ;copy contents of EAX into EDX
0012 8C C8          MOV AX,CS     ;copy CS into DS (two steps)
0014 8E D8          MOV DS,AX
0016 8E C8          MOV CS,AX     ;copy AX into CS (causes 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, whereas the data transferred from a register or memory location are variable data. Immediate addressing operates upon a byte or word of data. In the 80386 through the Core2 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. 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,[2] and Borland TASM[3]—do not use the # symbol for immediate data, but an older assembler used with some Hewlett-Packard logic development system does, as may others.

As mentioned, the MOV immediate instruction under 64-bit operation can include a 64-bit immediate number. An instruction such as MOV RAX,123456780A311200H is allowed in the 64-bit mode.

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

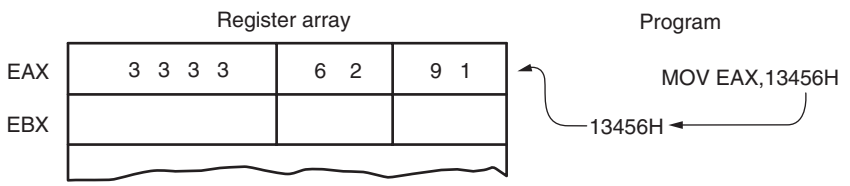


FIGURE 3–4 The operation of the MOV EAX,3456H instruction. This instruction copies the immediate data (13456H) into EAX.

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

[3]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 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 CodeView[4] (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 WorkBench[6] (PWB). Note that a TINY program always assembles as a command (.COM) program.

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

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

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

EXAMPLE 3–2

```
                    .MODEL TINY             ;choose single segment model
0000                .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
```

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

```
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 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 (l6-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,

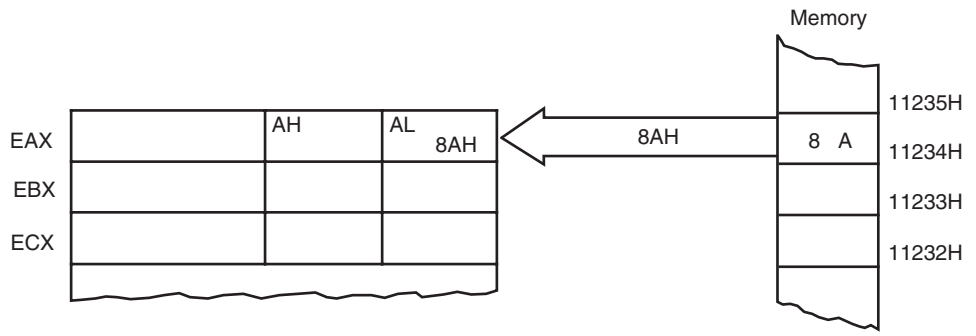


FIGURE 3–5 The operation of the MOV AL,[1234H] instruction when DS = 1000H.

[7]This 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.

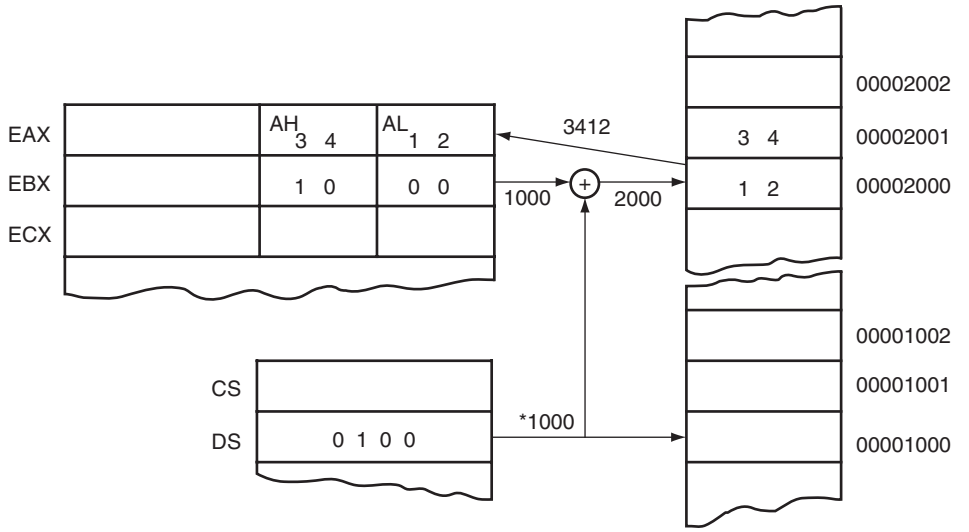
segment address of the data segment. If this program is assembled and executed with CodeView, the instructions can be viewed as they execute and change registers and memory locations.

EXAMPLE 3–6

```
                        .MODEL SMALL      ;choose small model
0000                    .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 0AAAAH         ;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;            exit to DOS
                        END;              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*
MOV CX,[BX]	16 bits	Copies the word contents of the data segment memory location addressed by BX into CX
MOV [BP],DL*	8 bits	Copies DL into the stack segment memory location addressed by BP
MOV [DI],BH	8 bits	Copies BH into the data segment memory location addressed by DI
MOV [DI],[BX]	—	Memory-to-memory transfers are not allowed except with string instructions
MOV AL,[EDX]	8 bits	Copies the byte contents of the data segment memory location addressed by EDX into AL
MOV ECX,[EBX]	32 bits	Copies the doubleword contents of the data segment memory location addressed by EBX into ECX
MOV RAX,[RDX]	64 bits	Copies the quadword contents of the memory location address by the linear address located in RDX into RAX (64-bit mode)

*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.

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 OWORD 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,DATAS instruction copies the contents of memory location DATAS into BX, while the MOV BX,OFFSET DATAS instruction copies the offset address 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 in the specified 16-bit register.

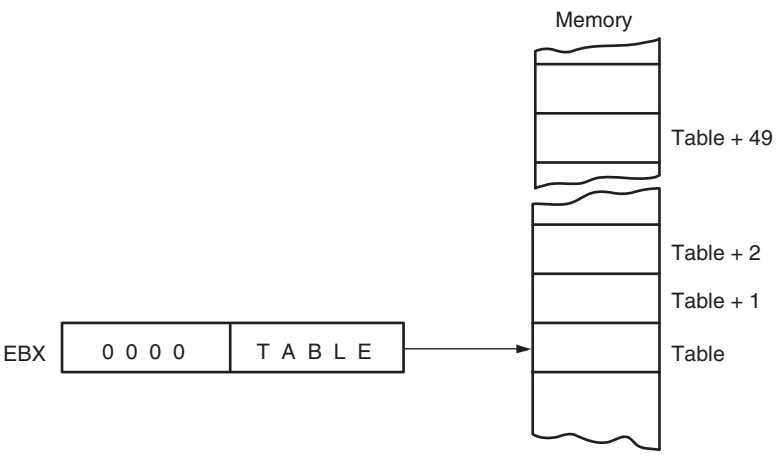


FIGURE 3–7 An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.

EXAMPLE 3–7

```
                          .MODEL SMALL          ;select small model
0000                      .DATA                 ;start data segment

0000 0032 [       DATAS  DW  50 DUP(?) ;setup array of 50 words
          0000
              ]
0000                      .CODE                 ;start code segment
                          .STARTUP              ;start program
0017 B8 0000              MOV  AX,0
001A 8E C0                MOV  ES,AX            ;address segment 0000 with ES
001C B8 0000 R            MOV  BX,OFFSET DATAS  ;address DATAS array with BX
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 43                   INC BX
002A E2 F6                LOOP AGAIN            ;repeat 50 times

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

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 (1 is added to BX) twice to address the next word in the table. Finally, the LOOP instruction repeats the LOOP 50 times. The LOOP instruction decrements (subtracts 1 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 XXXX.EXE, where XXXX.EXE is the name of the program that is being debugged. You can also 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 /S 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, whereas 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.

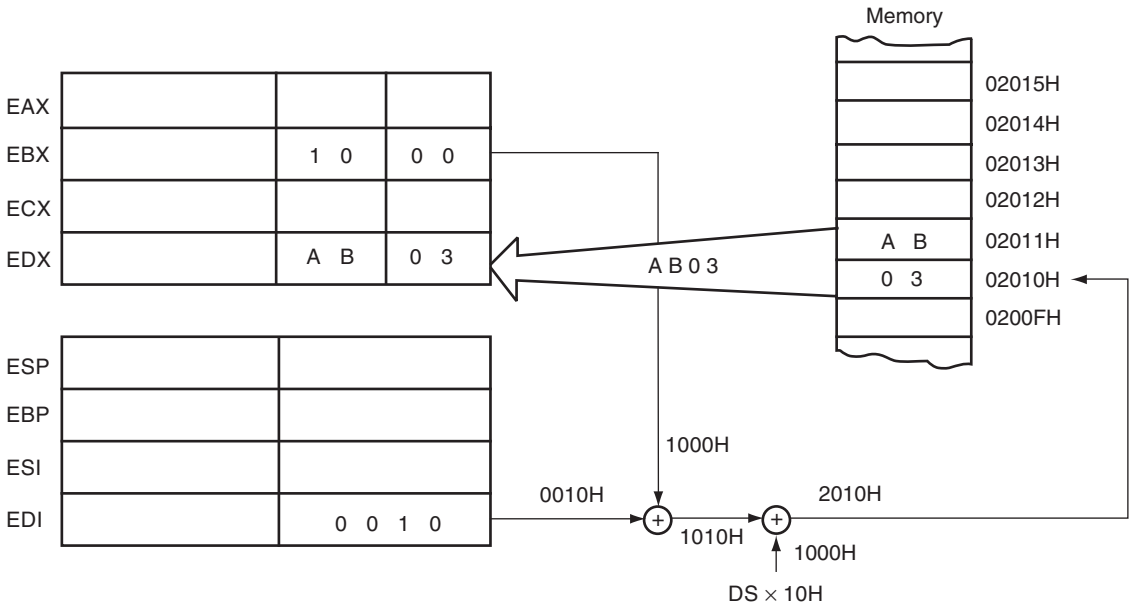


FIGURE 3–8 An example showing how the base-plus-index addressing mode functions for the MOV DX,[BX+DI] instruction. Notice that memory address 02010H is accessed because DS = 0100H, BX = 100H, and DI = 0010H.

Table 3–6 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. 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

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