TABLE 3-1 Examples of register-addressed instructions.

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AL,BL</td>
<td>8 bits</td>
<td>Copies BL into AL</td>
</tr>
<tr>
<td>MOV CH,CL</td>
<td>8 bits</td>
<td>Copies CL into CH</td>
</tr>
<tr>
<td>MOV RB,CL</td>
<td>8 bits</td>
<td>Copies CL, to the byte portion of RB (64-bit mode)</td>
</tr>
<tr>
<td>MOV RB,CH</td>
<td>8 bits</td>
<td>Not allowed</td>
</tr>
<tr>
<td>MOV AX,CX</td>
<td>16 bits</td>
<td>Copies CX into AX</td>
</tr>
<tr>
<td>MOV SI,BP</td>
<td>16 bits</td>
<td>Copies BP into SI</td>
</tr>
<tr>
<td>MOV DS,AX</td>
<td>16 bits</td>
<td>Copies AX into DS</td>
</tr>
<tr>
<td>MOV BP,SI</td>
<td>16 bits</td>
<td>Copies SI into BP (64-bit mode)</td>
</tr>
<tr>
<td>MOV SI,DI</td>
<td>16 bits</td>
<td>Copies DI into SI</td>
</tr>
<tr>
<td>MOV BX,ES</td>
<td>16 bits</td>
<td>Copies ES into BX</td>
</tr>
<tr>
<td>MOV ECX,EBX</td>
<td>32 bits</td>
<td>Copies EBX into ECX</td>
</tr>
<tr>
<td>MOV ESP,EDI</td>
<td>32 bits</td>
<td>Copies EDX into ESP</td>
</tr>
<tr>
<td>MOV EDX,EAX</td>
<td>32 bits</td>
<td>Copies RB into EDX (64-bit mode)</td>
</tr>
<tr>
<td>MOV RAX,RDX</td>
<td>64 bits</td>
<td>Copies RDX into RAX</td>
</tr>
<tr>
<td>MOV DS,CX</td>
<td>16 bits</td>
<td>Copies CX into DS</td>
</tr>
<tr>
<td>MOV ES,DS</td>
<td>—</td>
<td>Not allowed (segment-to-segment)</td>
</tr>
<tr>
<td>MOV BL,DX</td>
<td>—</td>
<td>Not allowed (mixed sizes)</td>
</tr>
<tr>
<td>MOV CS,AX</td>
<td>—</td>
<td>Not allowed (the code segment register may not be the destination register)</td>
</tr>
</tbody>
</table>

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/ESP 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 1234H from register CX into register BX. This erases the old contents (76A1H) 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 bitmost 16 bits of register EBX.

![Figure 3-3](Image)

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 or the Core2 processors, immediate addressing also operates on doublewords. 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,%13456H instruction is an example. Most assemblers do not use the % symbol, but represent immediate data as in the MOV AX,3456H instruction. In this test, the % symbol is not used for immediate data. The most common assemblers—Intel ASM, Microsoft, and Borland TASM—do not use the % symbol for immediate data, but an older assembler used with some Hewlett-Packard logic development systems, does so.

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](Image)
TABLE 3-2 Examples of Immediate addressing using the MOV instruction.

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV BL,44</td>
<td>Copies 44 decimal (2CH) into BL</td>
</tr>
<tr>
<td>MOV AX,44H</td>
<td>Copies 0044H into AX</td>
</tr>
<tr>
<td>MOV SI,0</td>
<td>Copies 0000H into SI</td>
</tr>
<tr>
<td>MOV CX,100</td>
<td>Copies 100 decimal (64H) into CH</td>
</tr>
<tr>
<td>MOV AL,'A'</td>
<td>Copies ASCII A into AL</td>
</tr>
<tr>
<td>MOV AH,1</td>
<td>Not allowed in 64-bit mode, but allowed in 32- or 16-bit modes</td>
</tr>
<tr>
<td>MOV AX,1BH</td>
<td>Copies ASCII 'A' into AX</td>
</tr>
<tr>
<td>MOV CL,11001110B</td>
<td>Copies 11001110 binary into CL</td>
</tr>
<tr>
<td>MOV EBX,12340000H</td>
<td>Copies 12340000H into EBX</td>
</tr>
<tr>
<td>MOV ESI,12</td>
<td>Copies 12 decimal into ESI</td>
</tr>
<tr>
<td>MOV EAX,100B</td>
<td>Copies 100 binary into EAX</td>
</tr>
<tr>
<td>MOV RCX,100H</td>
<td>Copies 100H into RCX</td>
</tr>
</tbody>
</table>

*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,41234H. 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 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 language 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 directs 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® (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®, or Programmer's WorkBench® (PWB). Note that a TINY program always assembles in a command (.COM) program.

EXAMPLE 3-2

```
DATA1 DB 20H ;define DATA1 as a byte of 20H
DATA2 DW 1000H ;define DATA2 as a word of 1000H
START: MOV AL,AL ;copy AL into AL
        MOV BL,AL ;copy BL into BL
        MOV CX,200 ;copy CX 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 0000H 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) {
    int eax, edx, ecx;
    mov eax, temp
    add eax, 20h
    mov temp, eax
    return temp; // return a 32-bit integer
}
```
Programs are also written using the inline assembler using 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] 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 1000H (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, 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 AL,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**

```
0060 AD 1234 m MOV AL,DS:[1234H]
0063 DA 0B 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. 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...

---

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

*This form may be typed into a MASM program, but it may often appear when the debugging tool is executed.
TABLE 3-4 Examples of direct data addressing using a displacement.

<table>
<thead>
<tr>
<th>Assembly Language</th>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV CH,DOG</td>
<td>8 bits</td>
<td>Copies the byte contents of data segment memory location DOG into CH</td>
</tr>
<tr>
<td>MOV CH,D$[1000H]**</td>
<td>8 bits</td>
<td>Copies the byte contents of data segment memory offset address 1000H into CH</td>
</tr>
<tr>
<td>MOV ES,DATA6</td>
<td>16 bits</td>
<td>Copies the word contents of data segment memory location DATA6 into ES</td>
</tr>
<tr>
<td>MOV DATA7,BP</td>
<td>16 bits</td>
<td>Copies BP into data segment memory location DATA7</td>
</tr>
<tr>
<td>MOV NUMBER,SP</td>
<td>16 bits</td>
<td>Copies SP into data segment memory location NUMBER</td>
</tr>
<tr>
<td>MOV DATA1,1AX</td>
<td>32 bits</td>
<td>Copies EAX into data segment memory location DATA1</td>
</tr>
<tr>
<td>MOV EDI,NUM1</td>
<td>32 bits</td>
<td>Copies the doubleword contents of data segment memory location NUM1 into EDI</td>
</tr>
</tbody>
</table>

*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
0000 DATA
0001 00 DINTAL DB 10H
0002 0000 DATA3 DW 000H
0003 AAAA DATA4 DB 000H
0004 CODE
0017 A00000 R MOV AL,DINTAL
0018 6A 26 0001 R MOV AH,DINTAL
0021 68 0004 R MOV BX,DINTAL
EXIT: CALL 0DS END;
```

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.

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, EBX 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 exceed 2GB.

TABLE 3-5 Examples of register indirect addressing.

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

*Note: Data addressed by BP or EBX are by default in the stack segment, while other indirect addressed instructions use the data segment by default.
exceed 0000H/FFFFH. 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 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 indicated 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],16H instruction is ambiguous. Does the MOV [DI],16H instruction address a byte-, word-, doubleword-, or quadword-sized memory location? The assembler can determine the size of the 16H. The instruction MOV BYTE PTR [DI],16H clearly designates the location addressed by DI as a byte-sized memory location. Likewise, the MOV DWORD PTR [DI],16H 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-bit mode). If programs are using the SIMD instructions, theOWORD 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 0000:0046C. Location 0000:0046C 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 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,OFFSET 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.

**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 = 010H, 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 DL,[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.

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

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

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 3-8, 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 29H.

EXAMPLE 3-8

```
0000 .MODEL SMALL          ;Select small model
0000 .DATA               ;Start data segment
0000 0010 [     ARRAY DB 16 DUP(?); Set array of 16 bytes
                  00 ]
0010 29 ;DB 29H          ;Element 10H
0011 001H ;DB 20 dup(?); Element 20H
0060 .CODE               ;Start code segment
0060 .STARTUP
0017 BS 0000 R ;MOV BL OFFSET ARRAY ;Address array
001A EF 001D ;MOV DL,1EH ;Address element 10H
001D EA 01 ;MOV AL,[BX+DI] ;Get element 10H
001E BF 0020 ;MOV DL,20H ;Address element 20H
0022 88 01 ;MOV [BX+DI],AL ;Save in element 20H
0023 .EXIT ;Exit to DOS
0024 END ;End program
```

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+100H] instruction. In this example, BX = 0100H and DS = 0200H, so the address generated is the sum of DS × 0H, BX, and the displacement of 100H, which addresses location 03100H. Remember that BX, DI, or SI addresses the data segment and BP addresses the stack segment. In the 8086 and above, the displacement can be a 32-bit number and the register can 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 is a number added to the register within the [ ] as in the MOV AL,[DI+1] 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 MOV AL,[DATA][DI+1]. Both forms of the displacement add to the base or base plus index register within the [ ] symbols. In the 8086–80286 microprocessors, the value of the displacement is limited to a 16-bit signed number with a value ranging between +32767 (FFFFH) and −32768 (8000H); in the 80386 and above, a 32-bit displacement is allowed with a value ranging between +2,147,483,647 (7FFFFFFF) and −2,147,483,648 (80000000H).

### Table 3–7 Examples of register relative addressing.

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

### Example 3–9

- MOV AX,[DI+100H]  
- MOV ARRAY[SI],BL  
- MOV LIST[BH+2],CL  
- MOV DI,SET_[BX]  
- MOV DI,EAX+10H  
- MOV ARRAY[EAX],EAX  
- MOV ARRAY[RBX],AL  
- MOV ARRAY[RCX],EAX

### Example 3–10

- MOV AX,[BX+100H]  
- MOV ARRAY[SI],BL  
- MOV LIST[BH+2],CL  
- MOV DI,SET_[BX]  
- MOV DI,EAX+10H  
- MOV ARRAY[EAX],EAX  
- MOV ARRAY[RBX],AL  
- MOV ARRAY[RCX],EAX

### Example 3–11

- ARRAY + 6  
- ARRAY + 5  
- ARRAY + 4  
- ARRAY + 3  
- ARRAY + 2  
- ARRAY + 1  
- ARRAY

### 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.

Example 3–9 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 3–8. The main difference is that, in Example 3–9, register BX is not used to address memory ARRAY; instead, ARRAY is used as a displacement to accomplish the same task.
TABLE 3-9  Examples of scaled-index addressing.

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

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 Core2 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 00000004H, 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 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. In the 64-bit mode, an instruction such as MOV RAX,[8*EDI] might appear in a program.

Example 3-11 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 directive 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 is in use, then use .586; and if the Pentium Pro, Pentium II, Pentium III, Pentium 4, or Core2 is in use, then use the .686 directive. If the microprocessor selection directive appears before the .MODEL statement, the microprocessor executes instructions in the 32-bit protected mode, which must execute in Windows.

EXAMPLE 3-11

```assembly
MODEL SMALL
DATA
allow small model
SELECT small model
; start data segment
; start data segment
; define array LIST
CODE
; code segment
; start code segment
; address array LIST
; address element 2
; get element 2
; address element 4
; get element 4
; address element 7
; get element 7
; exit to DOS
end
```

RIP Relative Addressing

This form of addressing uses the 64-bit instruction pointer register in the 64-bit mode to address a linear location in the flat memory model. The inline assembler program available to Visual C++ does not contain any way of using this addressing mode or any other 64-bit addressing mode. The Microsoft Visual C++ does not at present support developing 64-bit assembly code. The instruction pointer is normally addressed using a * as in *+34, which is 34 bytes ahead in a program. When Microsoft finally places an inline assembler into Visual C++ for the 64-bit mode, this most likely will be the way that RIP relative addressing will appear.

One source is Intel, which does produce a compiler with an inline assembler for 64-bit code (http://www.intel.com/cd/software/products/asmo-na/eng/compilers/cwint279582.htm).

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 assembler language directive and the end with the ENDS statement. A typical data structure is defined and used three times in Example 3-12. Notice that the name of the structure appears with the STRUC and with ENDS statement. The example shows the data structure as it was typed without the assembled version.

EXAMPLE 3-12

```assembly
; define the INFO data structure
INFO
; start info block
; define the structure
; define the field names
NAME DS DB 32 dup(?) ; reserve bytes for the name
ADDRESS DS DB 32 dup(?) ; reserve bytes for the address
CITY DS DB 16 dup(?) ; reserve bytes for the city
STATE DS DB 2 dup(?) ; reserve bytes for the state
ZIP DS DB 5 dup(?) ; reserve bytes for the zip code
INFO ENDS
```

The data structure in Example 3-12 defines five fields of information. The first 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 example 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.