CHAPTER 2

The Microprocessor and its Architecture

Topics - a) Internal Microprocessor Architecture b) Real Mode Addressing
Addressing Mode: a) data Addressing Mode
b) Program Memory Addressing Mode
c) Stack memory Addressing mode

Text Book Used – The INTEL Microprocessors; Architecture, Programming and Interfacing By Barry B Brey (8thEdition)

Internal Microprocessor Architecture

Before a program is written or any instruction investigated, the internal configuration of the microprocessor must be known. This section of the chapter details the program-visible internal architecture of the 8086–Core2 microprocessors. Also detailed are the function and purpose of each of these internal registers. Note that in a multiple core microprocessor each core contains the same programming model. The only difference is that each core runs a separate task or thread simultaneously.

The Programming Model

The programming model of the 8086 through the Core2 is considered to be **program visible** because its registers are used during application programming and are specified by the instructions. Other registers, detailed later in this chapter, are considered to be **program invisible** because they are not addressable directly during applications programming, but may be used indirectly during system programming. Only the 80286 and above contain the program-invisible registers used to control and operate the protected memory system and other features of the microprocessor.

Figure 2–1 illustrates the programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions. The earlier 8086, 8088, and 80286 contain 16-bit internal architectures, a subset of the registers shown in Figure 2–1. The 80386 through the Core2 microprocessors contain full 32-bit internal architectures. The architectures of the earlier 8086 through the 80286 are fully upward-compatible to the 80386 through the Core2. The shaded areas in this illustration represent registers that are found in early versions of the 8086, 8088, or 80286 microprocessors and are provided on the 80386–Core2 microprocessors for compatibility to the early versions.

The programming model contains 8-, 16-, and 32-bit registers. The Pentium 4 and Core2 also contain 64-bit registers when operated in the 64-bit mode as illustrated in the programming model. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL and are referred to when an instruction is formed using these two-letter designations. For example, an ADD AL, AH instruction adds the 8-bit contents of AH to AL. (Only AL changes due to this instruction.) The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, SI, IP, FLAGS, CS, DS, ES, SS, FS, and GS.

Note that the first 4 registers contain a pair of 8-bit registers. An example is AX, which contains AH and AL. The 16-bit registers are referenced with the two-letter designations such as AX. For example, an ADD DX, CX instruction adds the 16-bit contents of CX to DX. (Only DX changes due to this instruction.) The extended 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, EIP, and EFLAGS. These 32-bit extended registers, and 16-bit registers FS and GS, are available only in the 80386 and above. The 16-bit registers are referenced by the designations FS or GS for the two new 16-bit registers, and by a three-letter designation for the 32-bit registers. For example, an ADD ECX, EBX instruction adds the 32-bit contents of EBX to ECX. (Only ECX changes due to this instruction.)

Some registers are general-purpose or multipurpose registers, while some have special purposes. The multipurpose registers include EAX, EBX, ECX, EDX, EBP, EDI, and ESI. These registers hold various data sizes (bytes, words, or doublewords) and are used for almost any purpose, as dictated by a program.

The 64-bit registers are designated as RAX, RBX, and so forth. In addition to the renaming of the registers for 64-bit widths, there are also additional 64-bit registers that are called R8 through R15. The 64-bit extensions have multiplied the available register space by more than 8 times in the Pentium 4 and the Core2 when compared to the original microprocessor architecture as indicated in the shaded area in Figure 2–1. An example 64-bit instruction is ADD RCX, RBX, instruction, which adds the 64-bit contents of RBX to RCX. (Only RCX changes due to this instruction.) One difference exists: these additional 64-bit registers (R8 through R15) are addressed as a byte, word, doubleword, or **quadword**, but only the rightmost 8 bits is a byte. R8 through R15 have no provision for directly addressing bits 8 through 15 as a byte. In the 64-bit mode, a legacy high byte register (AH, BH, CH, or DH) cannot be addressed in the same instruction with an R8 through R15 byte. Because legacy software does not access R8 through R15, this causes no problems with existing 32-bit programs, which function without modification.

Table 2–1 shows the overrides used to access portions of a 64-bit register. To access the low-order byte of the R8 register, use R8B (where B is the low-order byte). Likewise, to access the low-order word of a numbered register, such as R10, use R10W in the instruction. The letter D is used to access a doubleword. An example instruction that copies the low-order doubleword from R8 to R11 is MOV R11D, R8D. There is no special letter for the entire 64-bit register.

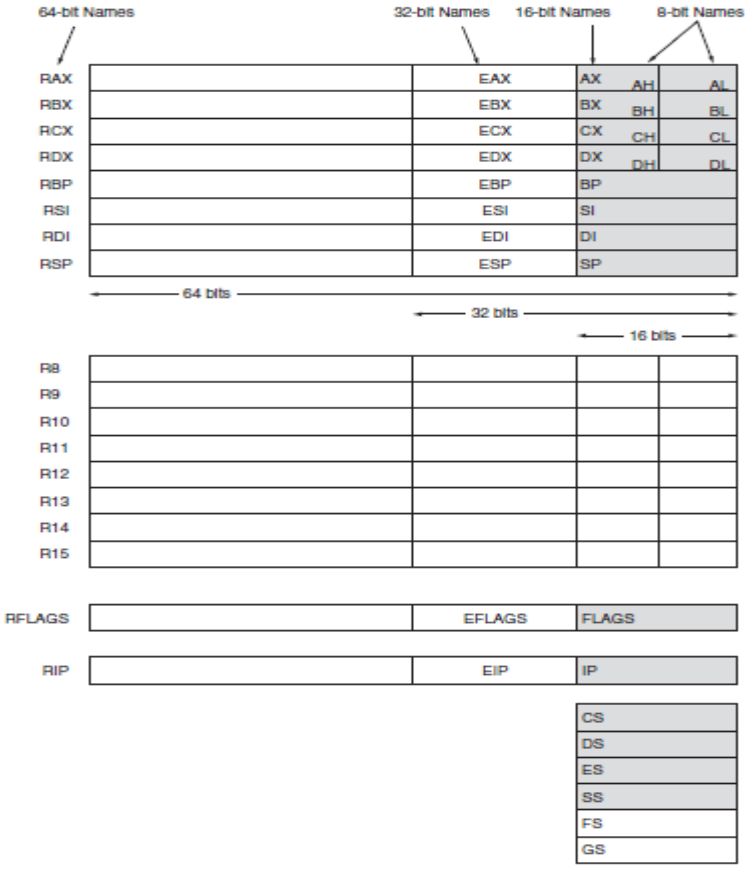


FIGURE 2–1 The programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions.

TABLE 2–1 Flat mode 64-bit access to numbered registers.

Register Size	*Override*	*Bits Accessed*	*Example*
8 bits	B	7–0	MOV R9B, R10B
16 bits	W	15–0	MOV R10W, AX
32 bits	D	31–0	MOV R14D, R15D
64 bits	—	63–0	MOV R13, R12

Multipurpose Registers

RAX RAX is referenced as a 64-bit register (RAX), a 32-bit register **(accumulator)** (EAX), a 16-bit register (AX), or as either of two 8-bit registers (AH and AL). Note that if an 8- or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered to be a multipurpose register. In the 80386 and above, the EAX register may also hold the offset address of a location in the memory system. In the 64-bit Pentium 4 and Core2, RAX holds a 64-bit offset address, which allows 1T (terra) byte of memory to be accessed through a 40-bit address bus. In the future, Intel plans to expand the address bus to 52 bits to address 4P (peta) bytes of memory.

RBX RBX is addressable as RBX, EBX, BX, BH, or BL. The BX register **(base index)** sometimes holds the offset address of a location in the memory system in all versions of the microprocessor. In the 80386 and above, EBX also can address memory data. In the 64-bit Pentium 4 and Core2, RBX can also address memory data.

RCX RCX, which is addressable as RCX, ECX, CX, CH, or CL, is a **(count)** general-purpose register that also holds the count for various instructions. In the 80386 and above, the ECX register also can hold the offset address of memory data. In the 64-bit Pentium 4, RCX can also address memory data. Instructions that use a count are the repeated string instructions (REP/REPE/REPNE); and shift, rotate, and LOOP/LOOPD instructions. The shift and rotate instructions use CL as the count, the repeated string instructions use CX, and the LOOP/LOOPD instructions use either CX or ECX. If operated in the 64-bit mode, LOOP uses the 64-bit RCX register for the loop counter.

RDX RDX, which is addressable as RDX, EDX, DX, DH, or DL, is a **(data)** general-purpose register that holds a part of the result from a multiplication or part of the dividend before a division. In the 80386 and above, this register can also address memory data.

RBP RBP, which is addressable as RBP, EBP, or BP, points to a memory **(base pointer)** location in all versions of the microprocessor for memory data transfers.

RDI RDI, which is addressable as RDI, EDI, or DI, often addresses **(destination index)** string destination data for the string instructions.

RSI RSI is used as RSI, ESI, or SI. The source index register often **(source index)** addresses source string data for the string instructions. Like RDI, RSI also functions as a general-purpose register. As a 16-bit register, it is addressed as SI; as a 32-bit register, it is addressed as ESI; and as a 64-bit register, it is addressed as RSI.

R8 through R15 These registers are only found in the Pentium 4 and Core2 if 64-bit extensions are enabled. As mentioned, data in these registers are addressed as 64-, 32-, 16-, or 8-bit sizes and are of general purpose.

Most applications will not use these registers until 64-bit processors are common. Please note that the 8-bit portion is the rightmost 8-bit only; bits 8 to 15 are not directly addressable as a byte.

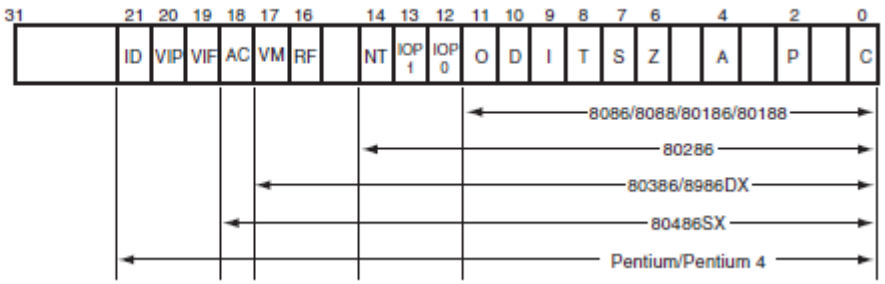


FIGURE 2–2 The EFLAG and FLAG register counts for the entire 8086 and Pentium microprocessor family.

Special-Purpose Registers. The special-purpose registers include RIP, RSP, and RFLAGS; and the segment registers include CS, DS, ES, SS, FS, and GS.

RIP RIP addresses the next instruction in a section of memory defined as **(instruction pointer)** a code segment. This register is IP (16 bits) when the microprocessor operates in the real mode and EIP (32 bits) when the 80386 and above operate in the protected mode. Note that the 8086, 8088, and 80286 do not contain an EIP register and only the 80286 and above operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction. In the 64-bit mode, RIP contains a 40-bit address at present to address a 1T flat address space.

RSP RSP addresses an area of memory called the stack. The stack memory **(stack pointer)** stores data through this pointer and is explained later in the text with the instructions that address stack data. This register is referred to as SP if used as a 16-bit register and ESP if referred to as a 32-bit register.

RFLAGS RFLAGS indicate the condition of the microprocessor and control its operation. Figure 2–2 shows the flag registers of all versions of the microprocessor. (Note the flags are upward-compatible from the 8086/8088 through the Core2 microprocessors.) The 8086–80286 contain a FLAG register (16 bits) and the 80386 and above contain an EFLAG register (32-bit extended flag register). The 64-bit RFLAGS contain the EFLAG register, which is unchanged in the 64-bit version.

The rightmost five flag bits and the overflow flag change after many arithmetic and logic instructions execute. The flags never change for any data transfer or program control operation. Some of the flags are also used to control features found in the microprocessor. Following is a list of each flag bit, with a brief description of their function. As instructions are introduced in subsequent chapters, additional detail on the flag bits is provided. The rightmost five flags and the overflow flag are changed by most arithmetic and logic operations, although data transfers do not affect them.

C (carry) Carry holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions, as dictated by some programs and procedures. This is especially true of the DOS function calls.

P (parity) Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is the count of ones in a number expressed as even or odd. For example, if a number contains three binary one bits, it has odd parity. If a number contains no one bits, it has even parity. The parity flag finds little application in modern programming and was implemented in early Intel microprocessors for checking data in data communications environments. Today parity checking is often accomplished by the data communications equipment instead of the microprocessor.

A (auxiliary carry) The auxiliary carry holds the carry (half-carry) after addition or the borrow after subtraction between bit positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z (zero) The zero flag shows that the result of an arithmetic or logic operation is zero. If Z = 1, the result is zero; if Z = 0, the result is not zero. This may be confusing, but that is how Intel decided to name this flag.

S (sign) The sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If S=0 , the sign bit (leftmost bit of a number) is set or negative; if S=1, the sign bit is cleared or positive.

T (trap) The trap flag enables trapping through an on-chip debugging feature. (A program is debugged to find an error or bug.) If the T flag is enabled (1), the microprocessor interrupts the flow of the program on

conditions as indicated by the debug registers and control registers. If the T flag is a logic 0, the trapping (debugging) feature is disabled. The Visual C++ debugging tool uses the trap feature and debug registers to debug faulty software. Setting trap flag puts the microprocessor into single step mode for debugging.

I (interrupt) The interrupt flag controls the operation of the INTR (interrupt request) input pin. If I=1, the INTR pin is enabled; if I=0, the INTR pin is disabled. The state of the I flag bit is controlled by the STI (set I flag) and CLI (clear I flag) instructions.

D (direction) The direction flag selects either the increment or decrement mode for the DI and/or SI registers during string instructions. If D=1, the registers are automatically decremented; if D=0, the registers are automatically incremented. The D flag is set with the STD (set direction) and cleared with the CLD (clear direction) instructions.

O (overflow) Overflows occur when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. For example, if 7FH ( ) is added—using an 8-bit addition—to 01H ( ), the result is 80H (–128). This result represents an overflow condition indicated by the overflow flag for signed addition. For unsigned operations, the overflow flag is ignored.

IOPL IOPL is used in protected mode operation to select the privilege **(I/O privilege level)** level for I/O devices. If the current privilege level is higher or more trusted than the IOPL, I/O executes without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs, causing execution to suspend. Note that an IOPL of 00 is the highest or most trusted and an IOPL of 11 is the lowest or least trusted.

NT (nested task) The nested task flag indicates that the current task is nested within another task in protected mode operation. This flag is set when the task is nested by software.

RF (resume) The resume flag is used with debugging to control the resumption of execution after the next instruction.

VM (virtual mode) The VM flag bit selects virtual mode operation in a protected mode system. A virtual mode system allows multiple DOS memory partitions that are 1M byte in length to coexist in the memory system. Essentially, this allows the system program to execute multiple DOS programs. VM is used to simulate DOS in the modern Windows environment.

AC The alignment check flag bit activates if a word or doubleword is **(alignment check)** addressed on a non-word or non-doubleword boundary. Only the 80486SX microprocessor contains the alignment check bit that is primarily used by its companion numeric coprocessor, the 80487SX, for synchronization.

VIF The VIF is a copy of the interrupt flag bit available to the Pentium– **(virtual interrupt)** Pentium 4 microprocessors.

VIP (virtual VIP provides information about a virtual mode interrupt for the **interrupt pending)** Pentium–Pentium 4 microprocessors. This is used in multitasking environments to provide the operating system with virtual interrupt flags and interrupt pending information.

ID (identification) The ID flag indicates that the Pentium–Pentium 4 microprocessors support the CPUID instruction. The CPUID instruction provides the system with information about the Pentium microprocessor, such as its version number and manufacturer.

Segment Registers. Additional registers, called segment registers, generate memory addresses when combined with other registers in the microprocessor. There are either four or six segment registers in various versions of the microprocessor. A segment register functions differently in the real mode when compared to the protected mode operation of the microprocessor. Details on their function in real and protected mode are provided later in this chapter. In the 64-bit flat model, segment registers have little use

in a program except for the code segment register. Following is a list of each segment register, along with its function in the system:

CS (code) The code segment is a section of memory that holds the code (programs and procedures) used by the microprocessor. The code segment register defines the starting address of the section of memory holding code. In real mode operation, it defines the start of a 64Kbyte section of memory; in protected mode, it selects a descriptor that describes the starting address and length of a section of memory holding code. The code segment is limited to 64K bytes in the 8088–80286, and 4G bytes in the 80386 and above when these microprocessors operate in the protected mode. In the 64-bit mode, the code segment register is still used in the flat model, but its use differs from other programming modes as explained in Section 2-5.

DS (data) The data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address. As with the code segment and other segments, the length is limited to 64K bytes in the 8086–80286, and 4G bytes in the 80386 and above.

ES (extra) The extra segment is an additional data segment that is used by some of the string instructions to hold destination data.

SS (stack) The stack segment defines the area of memory used for the stack. The stack entry point is determined by the stack segment and stack pointer registers. The BP register also addresses data within the stack segment.

FS and GS The FS and GS segments are supplemental segment registers available in the 80386–Core2 microprocessors to allow two additional memory segments for access by programs. Windows uses these segments for internal operations, but no definition of their usage is available.

REAL MODE MEMORY ADDRESSING

The 80286 and above operate in either the real or protected mode. Only the 8086 and 8088 operate exclusively in the real mode. In the 64-bit operation mode of the Pentium 4 and Core2, there is no real mode operation. This section of the text details the operation of the microprocessor in the real mode. **Real mode operation** allows the microprocessor to address only the first 1M byte of memory space—even if it is the Pentium 4 or Core2 microprocessor. Note that the first 1M byte of memory is called the **real memory**, **conventional memory**, or **DOS memory** system. The DOS operating system requires that the microprocessor operates in the real mode. Windows does not use the real mode.

Real mode operation allows application software written for the 8086/8088, which only contains 1M byte of memory, to function in the 80286 and above without changing the software. The upward compatibility of software is partially responsible for the continuing success of the Intel family of microprocessors. In all cases, each of these microprocessors begins operation in the real mode by default whenever power is applied or the microprocessor is reset. Note that if the Pentium 4 or Core2 operate in the 64-bit mode, it cannot execute real mode applications; hence, DOS applications will not execute in the 64-bit mode unless a program that emulates DOS is written for the 64-bit mode.

Segments and Offsets

A combination of a segment address and an offset address accesses a memory location in the real mode. All real mode memory addresses must consist of a segment address plus an offset address. The **segment address**, located within one of the segment registers, defines the beginning address of any 64K-byte memory segment. The **offset address** selects any location within the 64K byte memory segment. Segments in the real mode always have a length of 64K bytes.

Figure 2–3 shows how the **segment plus offset** addressing scheme selects a memory location. This illustration shows a memory segment that begins at location 10000H and ends at location IFFFFH—64K bytes in length. It also shows how an offset address, sometimes called a **displacement**, of F000H selects location 1F000H in the memory system. Note that the offset or displacement is the distance above the start of the segment, as shown in Figure 2–3.

The segment register in Figure 2–3 contains 1000H, yet it addresses a starting segment at location 10000H. In the real mode, each segment register is internally appended with a **0H** on its rightmost end. This forms a 20-bit memory address, allowing it to access the start of a segment. The microprocessor must generate a 20-bit memory address to access a location within the first 1M of memory. For example, when a segment register contains 1200H, it addresses a 64K-byte memory segment beginning at location 12000H. Likewise, if a segment register contains 1201H, it addresses a memory segment beginning at location 12010H. Because of the internally appended 0H, real mode segments can begin only at a l6-byte boundary in the memory system. This l6-byte boundary is often called a **paragraph**. Because a real mode segment of memory is 64K in length, once the beginning address is known, the **ending address** is found by adding FFFFH. For example, if a segment register contains 3000H, the first address of the segment is 30000H, and the last address is or 3FFFFH. Table 2–2 shows several examples of segment register contents and the starting and ending addresses of the memory segments selected by each segment address.

The offset address, which is a part of the address, is added to the start of the segment to address a memory location within the memory segment. For example, if the segment address is 1000H and the offset address is 2000H, the microprocessor addresses memory location 12000H.

The offset address is always added to the starting address of the segment to locate the data. The segment and offset address is sometimes written as 1000:2000 for a segment address of 1000H with an offset of 2000H.

In the 80286 (with special external circuitry) and the 80386 through the Pentium 4, an extra 64K minus 16 bytes of memory is addressable when the segment address is FFFFH and the HIMEM.SYS driver for DOS is installed in the system. This area of memory (0FFFF0H– 10FFEFH) is referred to as **high memory**. When an address is generated using a segment address of FFFFH, the A20 address pin is enabled (if supported in older systems) when an offset is added.

For example, if the segment address is FFFFH and the offset address is 4000H, the machine addresses memory location or 103FF0H. Notice that the A20 address line is the one in address 103FF0H. If A20 is not supported, the address is generated as 03FF0H because A20 remains a logic zero.

Some addressing modes combine more than one register and an offset value to form an offset address. When this occurs, the sum of these values may exceed FFFFH. For example, the address accessed in a segment whose segment address is 4000H and whose offset address is specified as the sum of F000H plus 3000H will access memory location 42000H instead of location 52000H. When the F000H and 3000H are added, they form a l6-bit (**modulo 16**) sum of 2000H used as the offset address; not 12000H, the true sum. Note that the carry of 1 (F000H + 3000H = 12000H) is dropped for this addition to form the offset address of 2000H. The address is generated as 4000:2000 or 42000H.

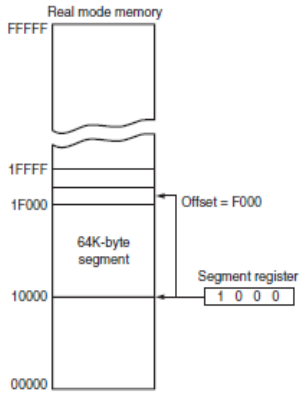


FIGURE 2–3 The real mode memory-addressing scheme, using a segment address plus an offset.

Segment Register	*Starting Address*	*Ending Address*
2000H	20000H	2FFFFH
2001H	20010H	3000FH
2100H	21000H	30FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

TABLE 2–2 Example of real mode segment addresses.

Default Segment and Offset Registers

The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules, which apply in the real and protected mode, define the segment register and offset register combination. For example, the code segment register is always used with the instruction pointer to address the next instruction in a program. This combination is **CS:IP** or **CS:EIP**, depending upon the microprocessor's mode of operation. The **code segment** register defines the start of the code segment and the **instruction pointer** locates the next instruction within the code segment. This combination (CS:IP or CS:EIP) locates the next instruction executed by the microprocessor. For example, if CS = 1400H and IP/EIP = 1200H the microprocessor fetches its next instruction from memory location 14000H+1200H or 15200H.

Another of the default combinations is the **stack**. Stack data are referenced through the stack segment at the memory location addressed by either the stack pointer (SP/ESP) or the pointer (BP/EBP). These combinations are referred to as SS:SP (SS:ESP), or SS:BP (SS:EBP).

For example, if SS=2000H and BP=3000H the microprocessor addresses memory location 23000H for the stack segment memory location. Note that in real mode, only the rightmost 16 bits of the extended register address a location within the memory segment. In the 80386–Pentium 4, never place a number larger than FFFFH into an offset register if the microprocessor is operated in the real mode. This causes the system to halt and indicate an addressing error. Other defaults are shown in Table 2–3 for addressing memory using any Intel microprocessor with 16-bit registers. Table 2–4 shows the defaults assumed in the 80386 and above using 32-bit registers. Note that the 80386 and above have a far greater selection of segment/ offset address combinations than do the 8086 through the 80286 microprocessors.

The 8086–80286 microprocessors allow four memory segments and the 80386–Core2 microprocessors allow six memory segments. Figure 2–4 shows a system that contains four memory segments. Note that a memory segment can touch or even overlap if 64K bytes of memory are not required for a segment. Think of segments as windows that can be moved over any area of memory to access data or code. Also note that a program can have more than four or six segments, but only access four or six segments at a time.

Suppose that an application program requires 1000H bytes of memory for its code, 190H bytes of memory for its data, and 200H bytes of memory for its stack. This application does not require an extra segment. When this program is placed in the memory system by DOS, it is loaded in the TPA at the first available area of memory above the drivers and other TPA program. This area is indicated by a **free-pointer** that is maintained by DOS. Program loading is handled automatically by the **program loader** located within DOS. Figure 2–5 shows how an application is stored in the memory system. The segments show an overlap because the amount of data in them does not require 64K bytes of memory. The side view of the segments clearly shows the overlap. It also shows how segments can be moved over any area of memory by changing the segment starting address. Fortunately, the DOS program loader calculates and assigns segment starting addresses.

Segment and Offset Addressing Scheme Allows Relocation

The segment and offset addressing scheme seems unduly complicated. It is complicated, but it also affords an advantage to the system. This complicated scheme of segment plus offset addressing allows DOS programs to be relocated in the memory system. It also allows programs written to function in the real mode to operate in a protected mode system. A **relocatable program** is one that can be placed into any area of memory and executed without change. **Relocatable data** are data that can be placed in any area of memory and used without any change to the program. The segment and offset addressing scheme allows both programs and data to be relocated without changing a thing in a program or data. This is ideal for use in a general-purpose computer system in which not all machines contain the same memory areas. The personal computer memory structure is different from machine to machine, requiring relocatable software and data.

Segment	*Offset*	*Special Purpose*
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

TABLE 2–3 Default 16-bit segment and offset combinations.

Segment	*Offset*	*Special Purpose*
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EAX, EBX, ECX, EDX, ESI, EDI, an 8- or 32-bit number	Data address
ES	EDI for string instructions	String destination address
FS	No default	General address
GS	No default	General address

TABLE 2–4 Default 32-bit segment and offset combinations.

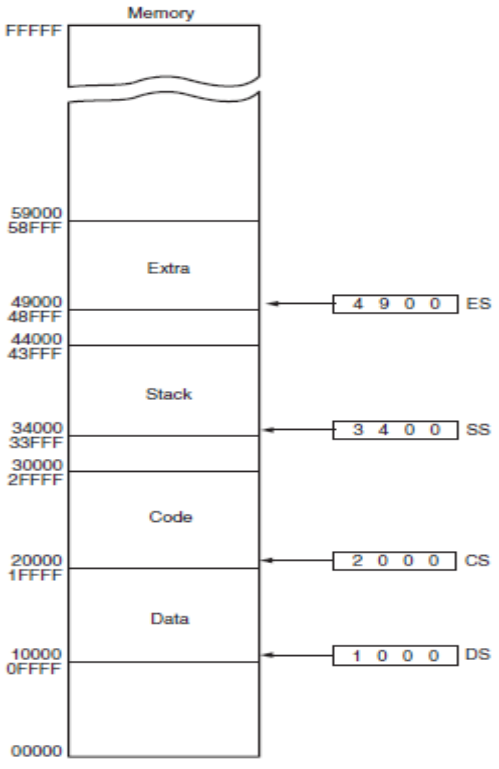


FIGURE 2–4 A memory system showing the placement of four memory segments.

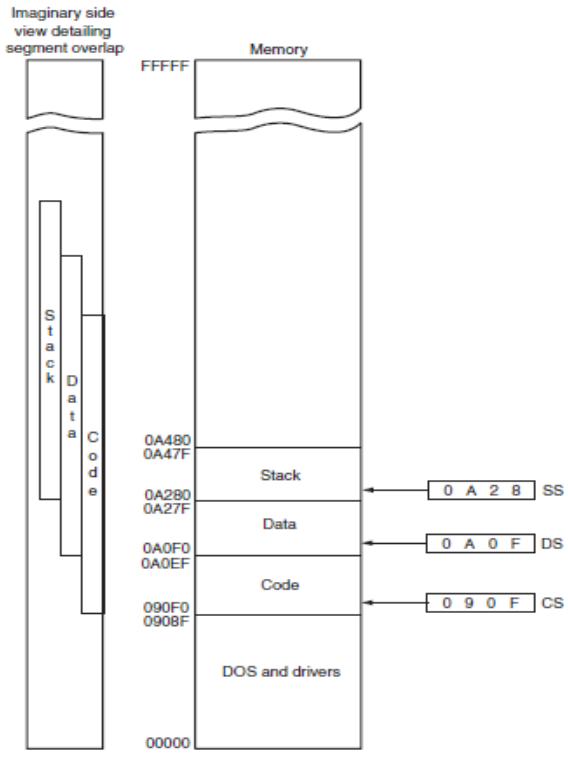


FIGURE 2–5 An application program containing a code, data, and stack segment loaded into a DOS system memory.

Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses. This is accomplished by moving the entire program, as a block, to a new area and then changing only the contents of the segment

registers. If an instruction is 4 bytes above the start of the segment, its offset address is 4. If the entire program is moved to a new area of memory, this offset address of 4 still points to 4 bytes above the start of the segment. Only the contents of the segment register must be changed to address the program in the new area of memory. Without this feature, a program would have to be extensively rewritten or altered before it is moved. This would require additional time or many versions of a program for the many different configurations of computer systems. This concept also applies to programs written to execute in the protected mode for Windows. In the Windows environment all programs are written assuming that the first 2G of memory are available for code and data. When the program is loaded, it is placed in the actual memory, which may be anywhere and a portion may be located on the disk in the form of a swap file.

Addressing Modes

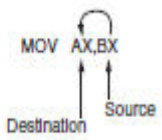
Data-Addressing Modes –

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:

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



Register Register addressing transfers a copy of a byte or word from the source **addressing** 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 doublewordsized 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 Immediate addressing transfers the source, an immediate byte, word, **addressing** 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 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 Direct addressing moves a byte or word between a memory location **addressing** 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 Register indirect addressing transfers a byte or word between a **addressing** 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 Base-plus-index addressing transfers a byte or word between a **addressing** 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 bytesized contents of register CL into the data segment memory location addressed by EAX plus EBX.)

Register relative Register relative addressing moves a byte or word between a register **addressing** 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- Base relative-plus-index addressing transfers a byte or word between a **index addressing** 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 Scaled-index addressing is available only in the 80386 through the **addressing** Pentium 4 microprocessor. The second register of a pair of registers is modified by the scale factor of 2X,4X or 8x to generate the operand memory address. (Example: A MOV EDX, [EAX+4*EBX] instruction loads EDX from the data segment memory location addressed by EAX plus four times EBX.) Scaling allows access to word (2X), doubleword (4X), or quadword (8X) memory array data. Note that a scaling factor of 1X also exists, but it is normally implied and does not appear 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 This addressing mode is only available to the 64-bit extensions on the **addressing** 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 +/-2 from the instruction is accessible by this addressing mode.

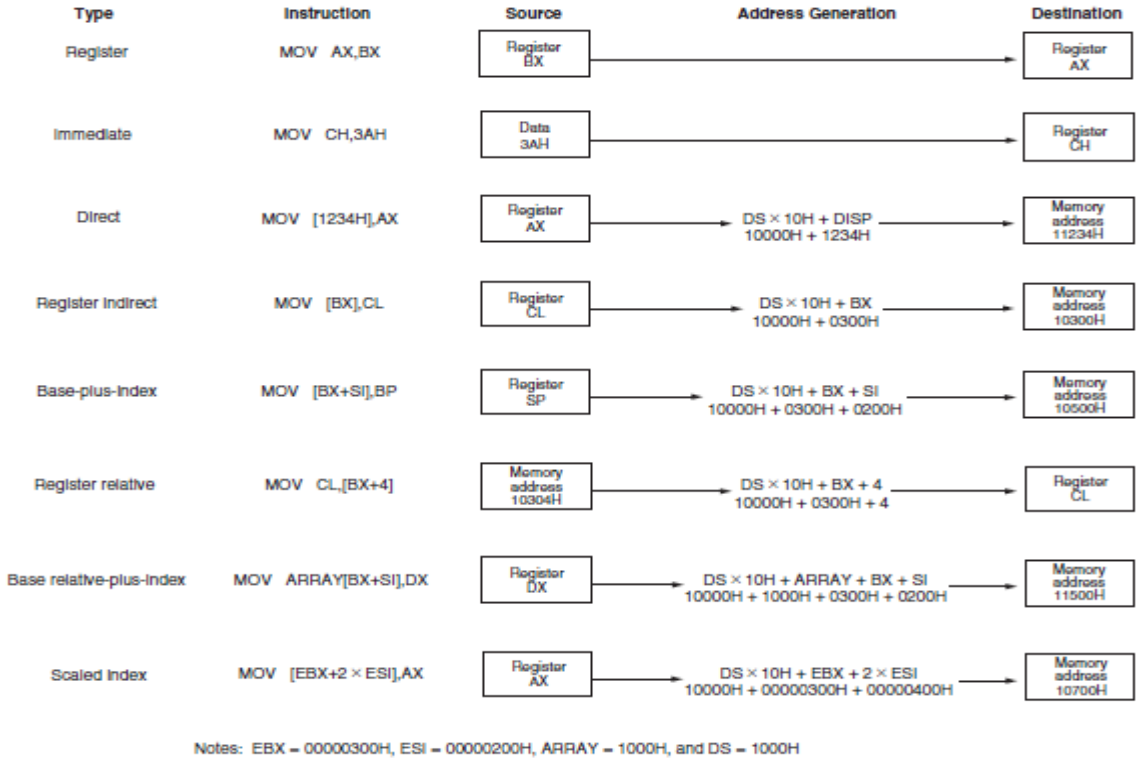


FIGURE 3–2 8086–Core2 data-addressing modes.

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 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.

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)

TABLE 3–1 Examples of register-addressed instructions.

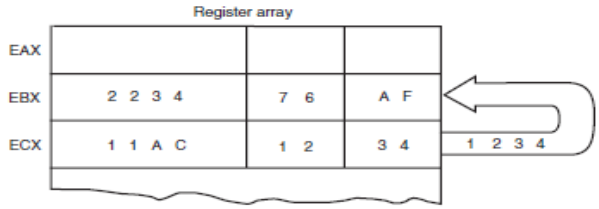


FIGURE 3–3 The effect of executing the MOV BX, CX instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.

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 TASM3—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.

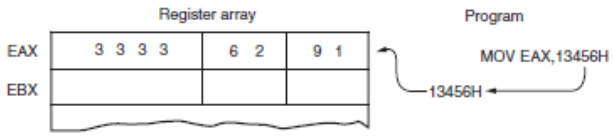


FIGURE 3–4 The operation of the MOV EAX,3456H instruction. This instruction copies the immediate data (13456H) into EAX.

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.

The symbolic assembler portrays immediate data in many ways. The letter **H** appends hexadecimal data. If hexadecimal data begin with a letter, the assembler requires that the data start with a **0**. For example, to represent 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.

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 programs. Example 3–4 shows a function in a Visual 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 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 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, 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.

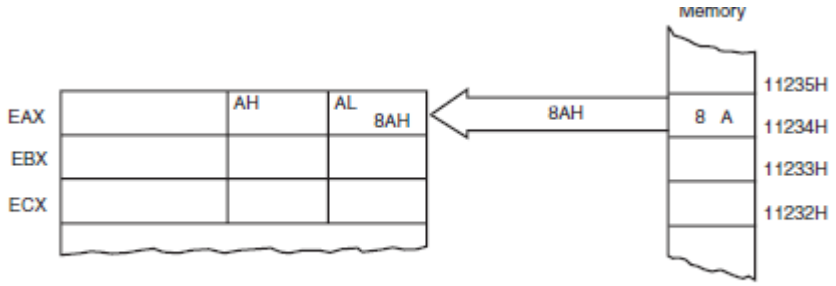


FIGURE 3–5 The operation of the MOV AL,[1234H] instruction when DS = 1000H.

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.

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 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.

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.

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.

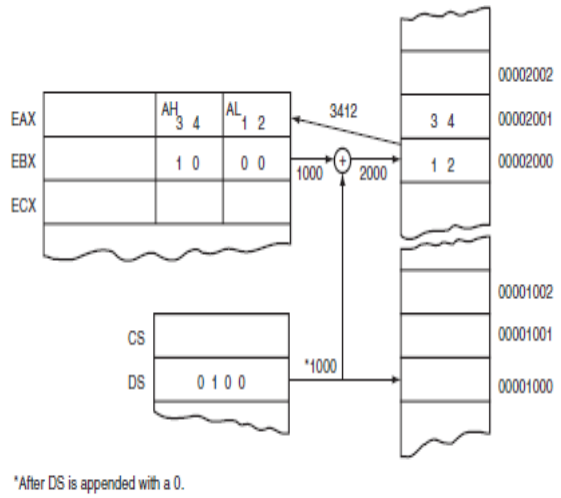


FIGURE 3–6 The operation of the MOV AX,[BX] instruction when BX = 1000H and DS = 0100H. Note that this instruction is shown after the contents of memory are transferred to AX.

TABLE 3–5 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.

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 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.

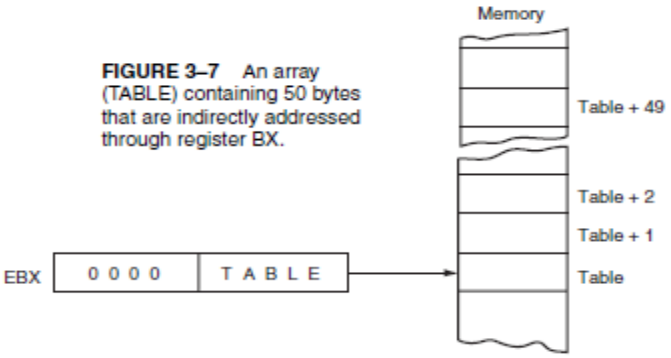


FIGURE 3–7 An array (TABLE) containing 50 bytes that are indirectly addressed through register BX.

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,[ ] 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.

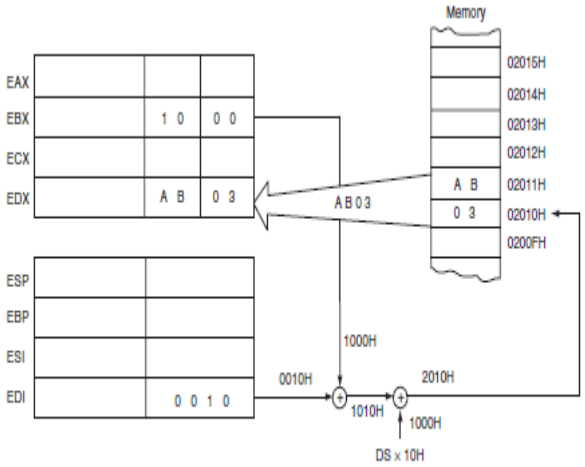


FIGURE 3–8 An example showing how the base-plus-index addressing mode functions for the MOV DX,[BX+DI] instruction. Notice that memory address 02010H is accessed because DS = 0100H, BX = 100H, and DI = 0010H.

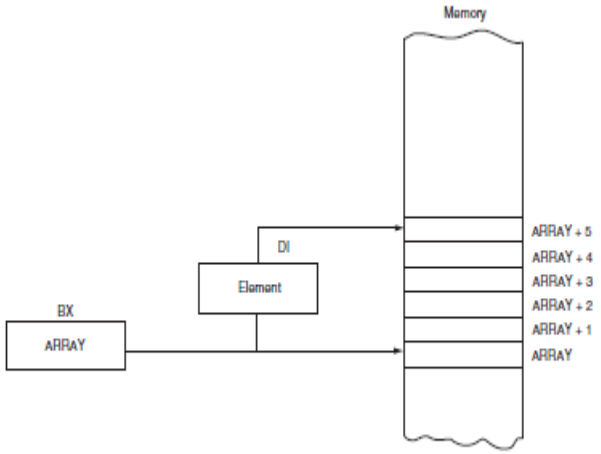


FIGURE 3–9 An example of the base-plus-index addressing mode. Here an element (DI) of an ARRAY (BX) is addressed.

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)

Locating Data with Base-Plus-Index Addressing. Figure 3–8 shows how data are addressed by the MOV DX,[ ] instruction when the microprocessor operates in the real mode. In this example, , which translate into memory address 02010H. This instruction transfers a copy of the word from location 02010H into the DX register. Table 3–6 lists some instructions used for base-plus-index addressing. Note that the Intel assembler requires that this addressing mode appear as [BX][DI] instead of [ ]. The MOV DX,[ ] 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,[ ] will assemble, but will not execute correctly.

Locating Array Data Using Base-Plus-Index Addressing. A major use of the base-plus-index addressing mode is to address elements in a memory array. Suppose that the elements in an array located in the data segment at memory location ARRAY must be accessed. To accomplish this, load the BX register (base) with the beginning address of the array and the DI register (index) with the element number to be accessed. Figure 3–9 shows the use of BX and DI to access an element in an array of data.

A short program, listed in Example 3–8, 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

```
                        .MODEL SMALL            ;select small model
0000                    .DATA                   ;start data segment
0000  0010 [  ARRAY DB 16 DUP(?)                ;setup array of 16 bytes
                00
                        ]
0010   29               DB 29H                  ;element 10H
0011  001E [            DB 20 dup(?)
              00
                        ]
0000                    .CODE                   ;start code segment
                        .STARTUP
0017 B8 0000 R          MOV BX,OFFSET ARRAY      ;address ARRAY
001A BF 0010            MOV DI,10H               ;address element 10H
001D 8A 01              MOV AL,[BX+DI]          ;get element 10H
001F BF 0020            MOV DI,20H              ;address element 20H
0022 88 01              MOV [BX+DI],AL           ;save in element 20H
                        .EXIT                   ;exit to DOS
                        END                     ;end program
```

Register Relative Addressing

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,[ ] instruction. In this example, , so the address generated is the sum of , BX, and the displacement of 1000H, which addresses location 03100H. Remember that BX, DI, or SI addresses the data segment and BP addresses the stack segment. In the 80386 and above, the displacement can be a 32-bit number and the register can 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,[ ] instruction, or it can be a displacement is subtracted from the register, as in MOV AL,[SI–l]. A displacement also can be an offset address appended to the front of the [ ], as in MOV AL,DATA[DI]. Both

forms of displacements also can appear simultaneously, as in the MOV AL,DATA[ ] instruction. 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 +32,767 (7FFFH) and –32,768 (8000H); in the 80386 and above, a 32-bit displacement is allowed with a value ranging between (7FFFFFFFH) and (80000000H).

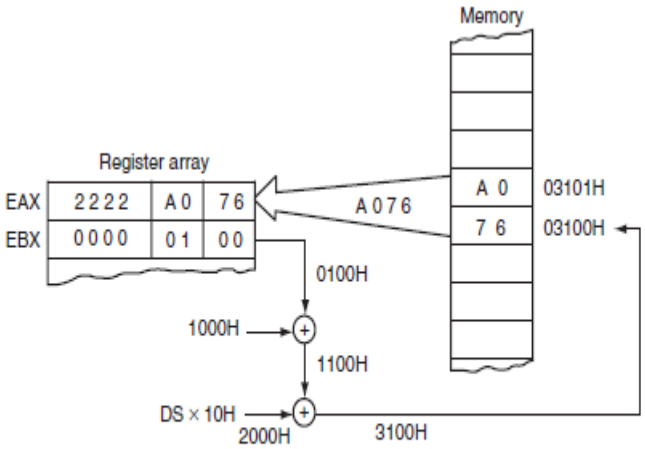


FIGURE 3–10 The operation of the MOV AX, [BX+1000H] instructon, when BX = 0100H and DS = 0200H.

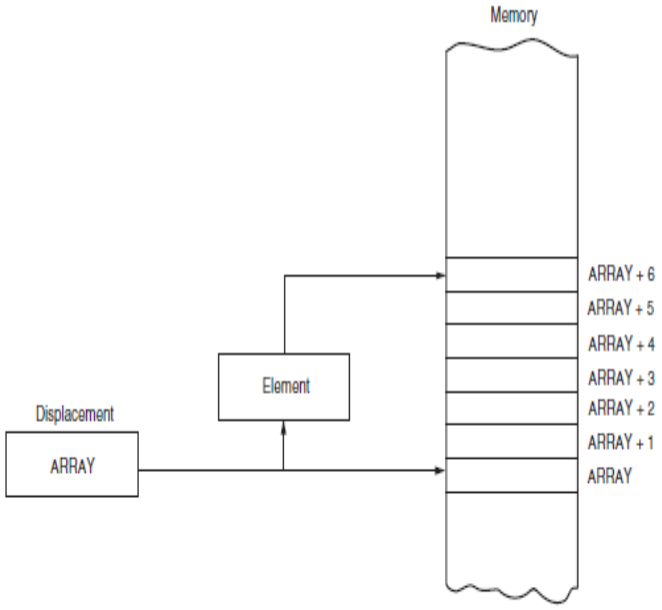


FIGURE 3–11 Register relative addressing used to address an element of ARRAY. The displacement addresses the start of ARRAY, and DI accesses an element.

TABLE 3–7 Examples of register relative addressing.

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

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

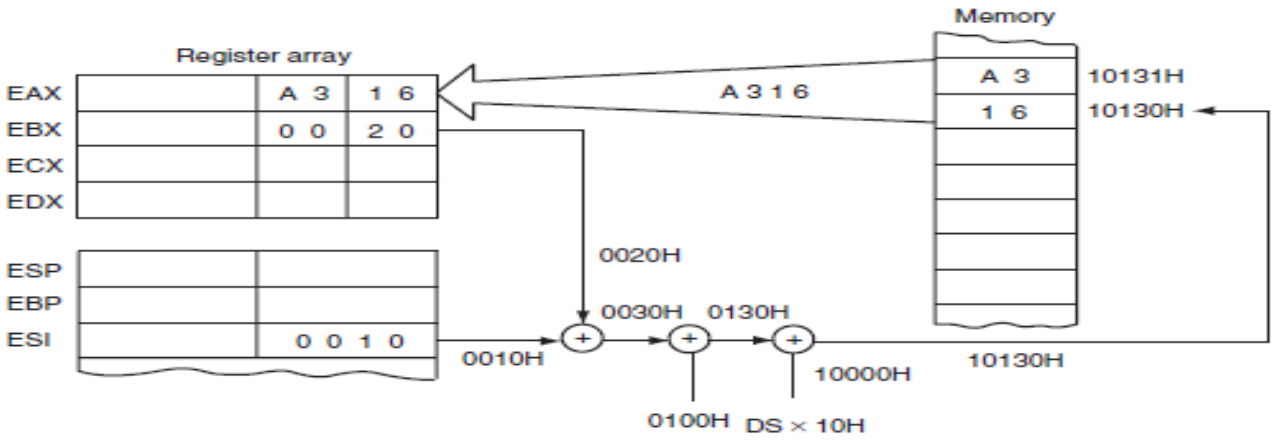


FIGURE 3–12 An example of base relative-plus-index addressing using a MOV AX,[BX+SI+100H] instruction. Note: DS = 1000H.

EXAMPLE 3–9

```
                                    .MODEL SMALL        ;select small model
0000                                .DATA               ;start data segment
0000 0010 [             ARRAY       DB 16 dup(?)        ;setup ARRAY
          00
            ]
0010 29                             DB 29               ;element 10H
0011 001E [                         DB 30 dup(?)
          00
            ]
0000                                .CODE               ;start code segment
                                    .STARTUP            ;start program
0017 BF 0010                        MOV DI,10H          ;address element 10H
001A 8A 85 0000 R                   MOV AL,ARRAY[DI]    ;get ARRAY element 10H
001E BF 0020                        MOV DI,20H          ;address element 20H
0021 88 85 0000 R                   MOV ARRAY[DI],AL    ;save it in element 20H
                                    .EXIT               ;exit to DOS
                                    END                 ;end of program
```

Base Relative-Plus-Index Addressing

The base relative-plus-index addressing mode is similar to base-plus-index addressing, but it adds a displacement, besides using a base register and an index register, to form the memory address. This type of addressing mode often addresses a two-dimensional array of memory data.

Addressing Data with Base Relative-Plus-Index. Base relative-plus-index addressing is the least-used addressing mode. Figure 3–12 shows how data are referenced if the instruction executed by the microprocessor is MOV AX,[BX+SI+100H]. The displacement of 100H adds to BX and SI to form the offset address within the data segment. Registers BX = 0020H, SI = 0100H, and DS = 1000H, so the effective address for this instruction is 10130H—the sum of these registers plus a displacement of 100H. This addressing mode is too complex for frequent use in programming. Some typical instructions using base relative-plus-index addressing appear in Table 3–8. Note that with the 80386 and above, the effective address is generated by the sum of two 32-bit registers plus a 32-bit displacement.

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

Assembly Language	*Size*	*Operation*
MOV DH,[BX+DI+20H]	8 bits	Copies the byte contents of the data segment memory location addressed by the sum of BX, DI and 20H into DH
MOV AX,FILE[BX+DI]	16 bits	Copies the word contents of the data segment memory location addressed by the sum of FILE, BX and DI into AX
MOV LIST[BP+DI],CL	8 bits	Copies CL into the stack segment memory location addressed by the sum of LIST, BP, and DI
MOV LIST[BP+SI+4],DH	8 bits	Copies DH into the stack segment memory location addressed by the sum of LIST, BP, SI, and 4
MOV EAX,FILE[EBX+ECX+2]	32 bits	Copies the doubleword contents of the memory location addressed by the sum of FILE, EBX, ECX, and 2 into EAX

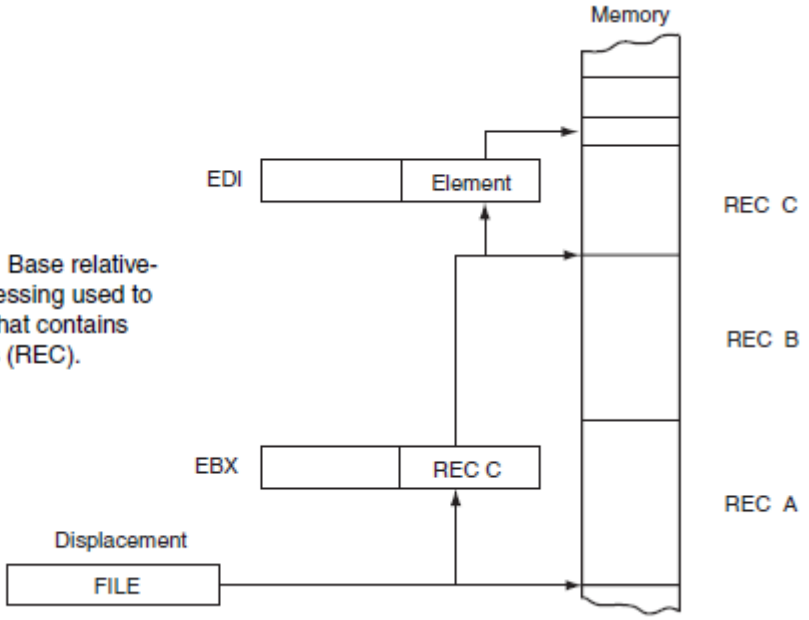


FIGURE 3–13 Base relative-plus-index addressing used to access a FILE that contains multiple records (REC).

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

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

TABLE 3–9 Examples of scaled-index addressing.

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

EXAMPLE 3–10

```
                        .MODEL SMALL            ;select small model
0000                    .DATA                   ;start data segment
0000 = 0000         FILE    EQU  THIS BYTE      ;assign FILE to this byte
0000 000A [         RECA    DB  10 dup(?)       ;10 bytes for record A
             00
               ]
000A 000A [         RECB    DB  10 dup(?)       ;10 bytes for record B
             00
               ]
0014 000A [         RECC    DB  10 dup(?)       ;10 bytes for record C
             00
               ]
001E 000A [         RECD    DB  10 dup(?)       ;10 bytes for record D
             00
               ]
0000                    .CODE                   ;start code segment
                        .STARTUP                ;start program
0017 BB 0000 R          MOV  BX,OFFSET RECA     ;address record A
001A BF 0000            MOV  DI,0               ;address element 0
001D 8A 81 0000 R       MOV  AL,FILE[BX+DI]     ;get data
0021 BB 0014 R          MOV  BX,OFFSET RECC     ;address record C
0024 BF 0002            MOV  DI,2               ;address element 2
0027 88 81 0000 R       MOV  FILE[BX+DI],AL     ;save data
                        .exit                   ;exit to DOS
                        end                     ;end of program
```

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 is used with quadword-sized memory arrays.

An example instruction is MOV AX,[EDI+2*ECX]. This instruction uses a scaling factor Of 2x , which multiplies the contents of ECX by 2 before adding it to the EDI register to form the memory address. If ECX contains a 00000000H, word-sized memory element 0 is addressed; if ECX contains a 00000001H, word-sized memory element 1 is accessed, and so forth. This scales the index (ECX) by a factor of 2 for a word-sized memory array. Refer to Table 3–9 for some examples of scaled-index addressing. As you can imagine, there are an extremely large number of the scaled-index addressed register combinations. Scaling is also applied to instructions that use a single indirect register to access memory. The MOV EAX,[4*EDI] is a scaled-index instruction that uses one register to indirectly address memory. In the 64-bit mode, an instruction such as MOV RAX,[8*RDI] 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 statement for the assembler to process 80386 instructions for DOS. If the 80486 is in use, the .486 directive appears after the .MODEL statement; if the Pentium 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

```
                                .MODEL SMALL            ;select small model
                                .386                    ;select 80386 microprocessor
0000                            .DATA                   ;start data segment
0000 0000 0001 0002 LIST        DW  0,1,2,3,4           ;define array LIST
     0003 0004
000A 0005 0006 0007             DW  5,6,7,8,9
     0008 0009
0000                            .CODE                   ;start code segment
0010 66|BB 00000000 R           MOV EBX,OFFSET LIST     ;address array LIST
0016 66|B9 00000002             MOV ECX,2               ;address element 2
001C 67&8B 04 4B                MOV AX,EBX+2*ECX]       ;get element 2
0020 66|B9 00000004             MOV ECX,4               ;address element 4
0026 67&89 04 4B                MOV [EBX+2*ECX],AX      ;store in element 4
002A 66|B9 00000007             MOV ECX,7               ;address element 7
0030 67&89 04 4B                MOV [EBX+2*ECX],AX      ;store in element 7
                                .exit                   ;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/cwin/279582.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 assembly 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

```
;define the INFO data structure
;
INFO    STRUC

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

INFO    ENDS

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

EXAMPLE 3–13

```
                        ;clear NAMES in array NAME1
                        ;
0000 B9 0020                    MOV  CX,32
0003 B0 00                      MOV  AL,0
0005 BE 0000 R                  MOV  DI,OFFSET NAME1.NAMES
0008 F3/AA                      REP  STOSB
                        ;
                        ;clear STREET in array NAME2
                        ;
000A B9 0020                    MOV  CX,32
000D B0 00                      MOV  AL,0
000F BE 0077 R                  MOV  DI,OFFSET NAME2.STREET
0012 F3/AA                      REP  STOSB
                        ;
                        ;clear ZIP in NAME3
                        ;
0014 B9 0005                    MOV  CX,5
0017 B0 00                      MOV  AL,0
0019 BE 0100 R                  MOV  DI,OFFSET NAME3.ZIP
001C F3/AA                      REP STOSB
```

The data structure in Example 3–12 defines five fields of information. The first is 32 bytes long and holds a name; the second is 32 bytes long and holds a street address; the third is 16 bytes long for the city; the fourth is 2 bytes long for the state; the fifth is 5 bytes long for the ZIP code. Once the structure is defined (INFO), it can be filled, as illustrated, with names and addresses. Three 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. When data are addressed in a structure, use the structure name and the field name to select a field from the structure. For example, to address the STREET

in NAME2, use the operand NAME2.STREET, where the name of the structure is first followed by a period and then by the name of the field. Likewise, use NAME3.CITY to refer to the city in structure NAME3.
A short sequence of instructions appears in Example 3-13 that clears the name field in structure NAME1, the address field in structure NAME2, and the ZIP code field in structure NAME3. The function and operation of the instructions in this program are defined in later chapters in the text. You may wish to refer to this example once you learn these instructions.

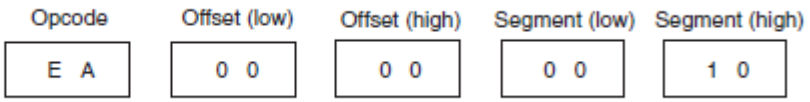
PROGRAM MEMORY-ADDRESSING MODES

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

Direct Program Memory Addressing

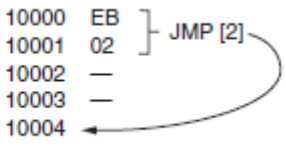
Direct program memory addressing is what many early microprocessors used for all jumps and calls. Direct program memory addressing is also used in high-level languages, such as the BASIC language GOTO and GOSUB instructions. The microprocessor uses this form of addressing, but not as often as relative and indirect program memory addressing are used. The instructions for direct program memory addressing store the address with the opcode.
For example, if a program jumps to memory location 10000H for the next instruction, the address (10000H) is stored following the opcode in the memory. Figure 3–14 shows the direct intersegment JMP instruction and the 4 bytes required to store the address 10000H. This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction.
(An **intersegment jump** is a jump to any memory location within the entire memory system.) The direct jump is often called a *far jump* because it can jump to any memory location for the next instruction. In the real mode, a far jump accesses any location within the first 1M byte of memory by changing both CS and IP. In protected mode operation, the far jump accesses a new code segment descriptor from the descriptor table, allowing it to jump to any memory location in the entire 4G-byte address range in the 80386 through Core2 microprocessors.

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



In the 64-bit mode for the Pentium 4 and Core2, a jump or a call can be to any memory location in the system. The CS segment is still used, but not for the address of the jump or the call. The CS register contains a pointer to a descriptor that describes the access rights and privilege level of the code segment, but not the address of the jump or call.
The only other instruction that uses direct program addressing is the intersegment or far CALL instruction. Usually, the name of a memory address, called a *label*, refers to the location that is called or jumped to instead of the actual numeric address. When using a label with the CALL or JMP instruction, most assemblers select the best form of program addressing.

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



Relative Program Memory Addressing

Relative program memory addressing is not available in all early microprocessors, but it is available to this family of microprocessors. The term *relative* means “relative to the instruction pointer (IP).” For example, if a JMP instruction skips the next 2 bytes of memory, the address in relation to the instruction pointer is a 2 that adds to the instruction pointer. This develops the address of the next program instruction. An example of the relative JMP instruction is shown in Figure 3–15. Notice that the JMP instruction is a 1-byte instruction, with a 1-byte or a 2-byte displacement that adds to the instruction pointer. A 1-byte

displacement is used in **short** jumps, and a 2-byte displacement is used with **near** jumps and calls. Both types are considered to be intrasegment jumps. (An **intrasegment jump** is a jump anywhere within the current code segment.)

In the 80386 and above, the displacement can also be a 32-bit value, allowing them to use relative addressing to any location within their 4G-byte code segments. Relative JMP and CALL instructions contain either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference. (The 80386 and above can have an 8-bit or 32-bit displacement.) All assemblers automatically calculate the distance for the displacement and select the proper 1-, 2- or 4-byte form. If the distance is too far for a 2-byte displacement in an 8086 through an 80286 microprocessor, some assemblers use the direct jump. An 8-bit displacement *(short)* has a jump range of between +127 and -128 bytes from the next instruction; a 16-bit displacement *(near)* has a range of +/- 32Kbytes. In the 80386 and above, a 32-bit displacement allows a range of +/-2 bytes. The 32-bit displacement can only be used in the protected mode.

Indirect Program Memory Addressing

The microprocessor allows several forms of program indirect memory addressing for the JMP and CALL instructions. Table 3–10 lists some acceptable program indirect jump instructions, which can use any 16-bit register (AX, BX, CX, DX, SP, BP, DI, or SI); any relative register ([BP], [BX], [DI], or [SI]); and any relative register with a displacement. In the 80386 and above, an extended register can also be used to hold the address or indirect address of a relative JMP or CALL. For example, the JMP EAX jumps to the location address by register EAX.

If a 16-bit register holds the address of a JMP instruction, the jump is near. For example, if the BX register contains 1000H and a JMP BX instruction executes, the microprocessor jumps to offset address 1000H in the current code segment.

TABLE 3–10 Examples of indirect program memory addressing.

Assembly Language	*Operation*
JMP AX	Jumps to the current code segment location addressed by the contents of AX
JMP CX	Jumps to the current code segment location addressed by the contents of CX
JMP NEAR PTR[BX]	Jumps to the current code segment location addressed by the contents of the data segment 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)

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 l6-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.

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
```

EXAMPLE 3–14

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

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 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 x 10H to form the stack memory address in the real mode. In protected mode operation, the SS register holds a selector that accesses a descriptor for the base address of the stack segment.

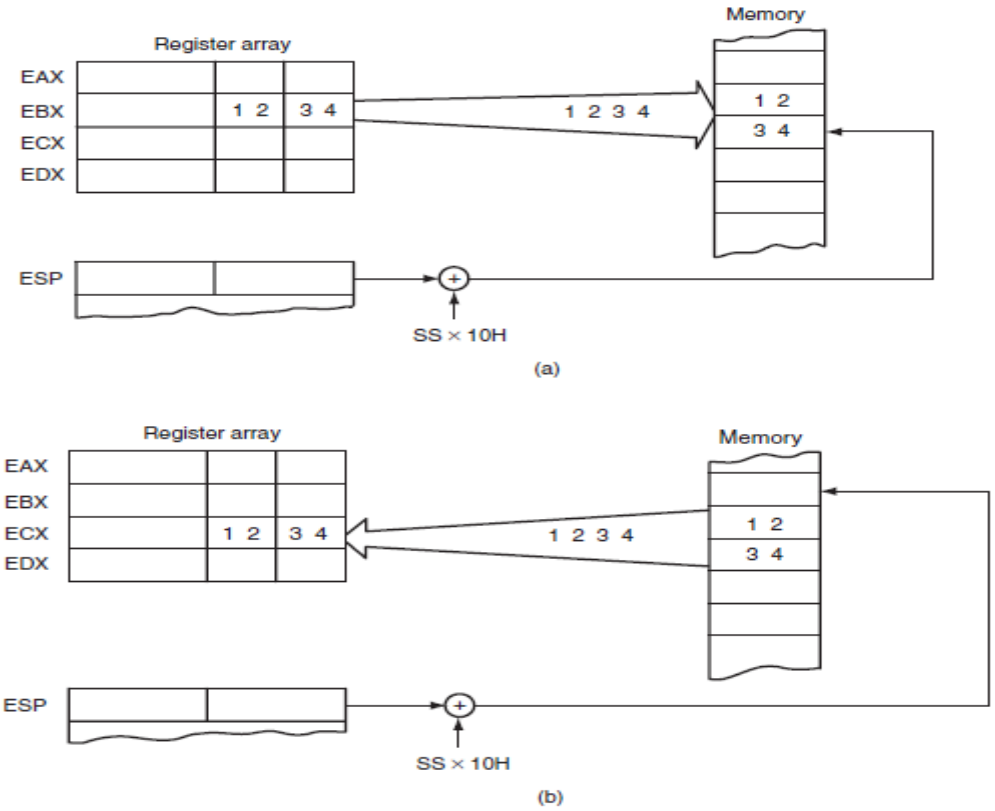


FIGURE 3–17 The PUSH and POP instructions: (a) PUSH BX places the contents of BX onto the stack; (b) POP CX removes data from the stack and places them into CX. Both instructions are shown after execution.

Whenever data are popped from the stack [see Figure 3–17(b)], the low-order 8 bits are removed from the location addressed by SP. The high-order 8 bits are removed from the location addressed by SP+1. The SP register is then incremented by 2. Table 3–11 lists some of the PUSH and POP instructions available to the microprocessor. Note that PUSH and POP 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 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.

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 RAC 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

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
```