CHAPTER 3

Data movement Instruction

Topics - Data movement Instruction, PUSH and POP, Load Effective Address, String Data Transfer Arithmetic Instruction : a) Addition b) Subtraction c) Comparison d) Multiplication e) Division, BCD & ASCII Arithmetic, Assembler Details.

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

MOV REVISITED

The MOV instruction, introduced in Chapter 3, explains the diversity of 8086–Core2 addressing modes. In this chapter, the MOV instruction introduces the machine language instructions available with various addressing modes and instructions. Machine code is introduced because it may occasionally be necessary to interpret machine language programs generated by an assembler or inline assembler of Visual C++. Interpretation of the machine's native language (**machine language**) allows debugging or modification at the machine language level. Occasionally, machine language patches are made by using the DEBUG program available with DOS and also in Visual for Windows, which requires some knowledge of machine language. Conversion between machine and assembly language instructions is illustrated in Appendix B.

Machine Language

Machine language is the native binary code that the microprocessor understands and uses as its instructions to control its operation. Machine language instructions for the 8086 through the Core2 vary in length from 1 to as many as 13 bytes. Although machine language appears complex, there is order to this microprocessor's machine language. There are well over 100,000 variations of machine language instructions, meaning that there is no complete list of these variations.

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

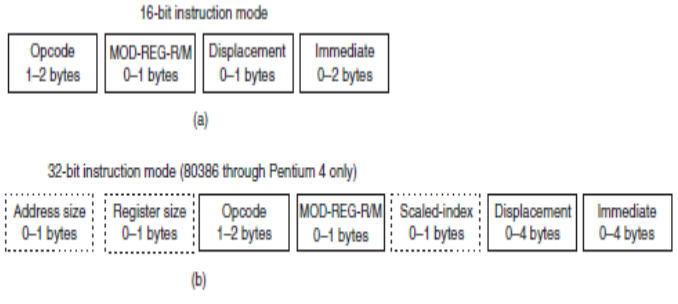
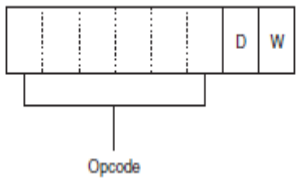


FIGURE 4–1 The formats of the 8086–Core2 instructions. (a) The 16-bit form and (b) the 32-bit form.

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



The first 2 bytes of the 32-bit instruction mode format are called override prefixes because they are not always present. The first modifies the size of the operand address used by the instruction and the second modifies the register size. If the 80386 through the Pentium 4 operate as 16-bit instruction mode machines (real or protected mode) and a 32-bit register is used, the **register-size prefix** (66H) is appended to the front of the instruction. If operated in the 32-bit instruction mode (protected mode only) and a 32-bit register is

used, the register-size prefix is absent. If a 16-bit register appears in an instruction in the 32-bit instruction mode, the register-size 16-bit instruction mode, the register-size prefix is present to select a l6-bit register. The **address size-prefix** (67H) is used in a similar fashion, as explained later in this chapter. The prefixes toggle the size of the register and operand address from l6-bit to 32-bit or from 32-bit to l6-bit for the prefixed instruction.

Note that the l6-bit instruction mode uses 8- and l6-bit registers and addressing modes, while the 32-bit instruction mode uses 8- and 32-bit registers and addressing modes by default. The prefixes override these defaults so that a 32-bit register can be used in the l6-bit mode or a l6-bit register can be used in the 32-bit mode. The **mode of operation** (16 or 32 bits) should be selected to function with the current application. If 8- and 32-bit data pervade the application, the 32-bit mode should be selected; likewise, if 8- and l6-bit data pervade, the l6-bit mode should be selected.

Normally, mode selection is a function of the operating system. (Remember that DOS can operate only in the l6-bit mode, where Windows can operate in both modes.)

The Opcode. The **opcode** selects the operation (addition, subtraction, move, and so on) that is performed by the microprocessor. The opcode is either 1 or 2 bytes long for most machine language instructions. Figure 4–2 illustrates the general form of the first opcode byte of many, but *not* all, machine language instructions. Here, the first 6 bits of the first byte are the binary opcode. The remaining 2 bits indicate the **direction** (D)—not to be confused with the instruction mode bit (16/32) or direction flag bit (used with string instructions)—of the data flow, and indicate whether the data are a byte or a word (W). In the 80386 and above, words and doublewords are both specified when W=1. The instruction mode and register-size prefix (66H) determine whether W represents a word or a doubleword.

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

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

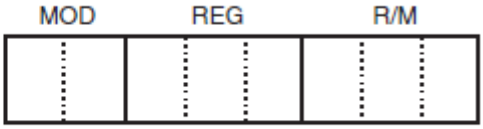


TABLE 4–1 MOD field for the 16-bit instruction mode.

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

MOD Field. The MOD field specifies the addressing mode (MOD) for the selected instruction. The MOD field selects the type of addressing and whether a displacement is present with the selected type. Table 4–1 lists the operand forms available to the MOD field for l6-bit instruction mode, unless the operand address-size override prefix (67H) appears. If the MOD field contains an 11, it selects the register-addressing mode. Register addressing uses the R/M field to specify a register instead of a memory location. If the MOD field contains a 00, 01, or 10, the R/M field selects one of the data memory-addressing modes. When MOD selects a data memory addressing mode, it indicates that the addressing mode contains no displacement (00), an 8-bit signextended displacement (01), or a l6-bit displacement (10). The MOV AL,[DI] instruction is an example that contains no displacement, a MOV AL,[DI+2] instruction uses an 8-bit displacement (+2), and a MOV AL,[DI+1000H] instruction uses a 16-bit displacement (+1000H). All 8-bit displacements are sign-extended into 16-bit displacements when the microprocessor executes the instruction. If the 8-bit

displacement is 00H–7FH (positive), it is signextended to 0000H–007FH before adding to the offset address. If the 8-bit displacement is 80H–FFH (negative), it is sign-extended to FF80H–FFFFH. To sign-extend a number, its sign-bit is copied to the next higher-order byte, which generates either a 00H or an FFH in the next higher-order byte. Some assembler programs do not use the 8-bit displacements and in place default to all 16-bit displacements.

In the 80386 through the Core2 microprocessors, the MOD field may be the same as shown in Table 4–1 for 16-bit instruction mode; if the instruction mode is 32 bits, the MOD field is as it appears in Table 4–2. The MOD field is interpreted as selected by the address-size override prefix or the operating mode of the microprocessor. This change in the interpretation of the MOD field and instruction supports many of the numerous additional addressing modes allowed in the 80386 through the Core2. The main difference is that when the MOD field is a 10, this causes the 16-bit displacement to become a 32-bit displacement, to allow any protected mode memory location (4G bytes) to be accessed. The 80386 and above only allow an 8- or 32-bit displacement when operated in the 32-bit instruction mode, unless the address-size override prefix appears. Note that if an 8-bit displacement is selected, it is sign-extended into a 32-bit displacement by the microprocessor.

Register Assignments. Table 4–3 lists the register assignments for the REG field and the R/M field (MOD=11). This table contains three lists of register assignments: one is used when the W bit=0 (bytes), and the other two are used when the W bit=1 (words or doublewords). Note that doubleword registers are only available to the 80386 through the Core2.

TABLE 4–2 MOD field for the 32-bit instruction mode (80386–Core2 only).

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

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

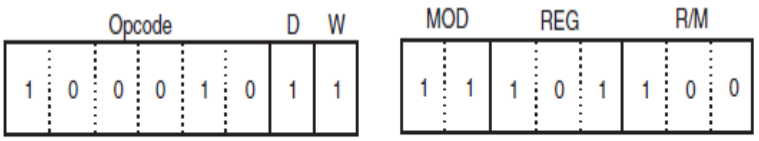
Code	W = 0 *(Byte)*	W = 1 *(Word)*	W = 1 *(Doubleword)*
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Suppose that a 2-byte instruction, 8BECH, appears in a machine language program. Because neither a 67H (operand address-size override prefix) nor a 66H (register-size override prefix) appears as the first byte, the first byte is the opcode. If the microprocessor is operated in the 16-bit instruction mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2, as illustrated in Figure 4–4. The opcode is 100010. If you refer to Appendix B, which lists the machine language instructions, you will find that this is the opcode for a MOV instruction. Notice that both the D and W bits are a logic 1, which means that a word moves into the destination register specified in the REG field. The REG field contains a 101, indicating register BP, so the MOV instruction moves data into register BP. Because the MOD field contains a 11, the R/M field also indicates a register. Here, R/M=100 (SP); therefore, this instruction moves data from SP into BP and is written in symbolic form as a MOV BP, SP instruction.

Suppose that a 668BE8H instruction appears in an 80386 or above, operated in the 16-bit instruction mode. The first byte (66H) is the register-size override prefix that selects 32-bit register operands for the 16-bit instruction mode. The remainder of the instruction indicates that the opcode is a MOV with a source operand of EAX and a destination operand of EBP. This instruction is a MOV EBP, EAX. The same instruction

becomes a MOV BP, AX instruction in the 80386 and above if it is operated in the 32-bit instruction mode, because the register-size override prefix selects a 16-bit register. Luckily, the assembler program keeps track of the register- and address-size prefixes and the mode of operation. Recall that if the .386 switch is placed before the .MODEL statement, the 32-bit mode is selected; if it is placed after the .MODEL statement, the 16-bit mode is selected. All programs written using the inline assembler in Visual C++ are always in the 32-bit mode.

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



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

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

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

*Note: See text section, Special Addressing Mode.

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

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

Figure 4–5 illustrates the machine language version of the 16-bit instruction MOV DL,[DI] or instruction (8AI5H). This instruction is 2 bytes long and has an opcode 100010, D=1 (to REG from R/M), W=0 (byte), MOD=00 (no displacement), REG=010 (DL), and R/M=101 ([DI]). If the instruction changes to MOV DL,3DI_14, the MOD field changes to 01 for an 8-bit displacement, but the first 2 bytes of the instruction otherwise remain the same. The instruction now becomes 8A5501H instead of 8A15H. Notice that the 8-bit displacement appends to the first 2 bytes of the instruction to form a 3-byte instruction instead of 2 bytes. If the instruction is again changed to a MOV DL,3DI_1000H], the machine language form becomes 8A750010H. Here the 16-bit displacement of 1000H (coded as 0010H) appends the opcode.

Special Addressing Mode. There is a special addressing mode that does not appear in Tables 4–2, 4–3, or 4–4. It occurs whenever memory data are referenced by only the displacement mode of addressing for 16-bit instructions. Examples are the MOV [1000H],DL and MOV NUMB,DL instructions. The first instruction moves the contents of register DL into data segment memory location 1000H. The second instruction moves register DL into symbolic data segment memory location NUMB.

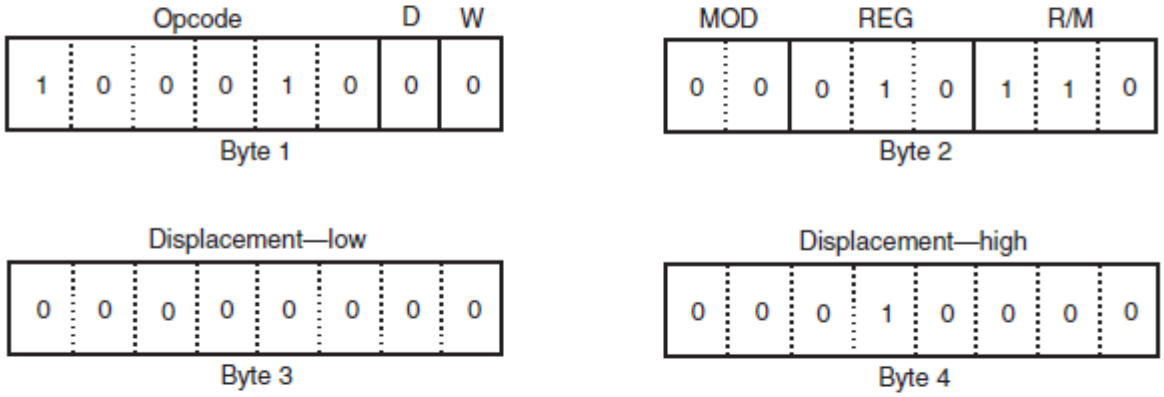
Whenever an instruction has only a displacement, the MOD field is always a 00 and the R/M field is always 110. As shown in the tables, the instruction contains no displacement and uses addressing mode [BP]. You cannot actually *use* addressing mode [BP] without a displacement in machine language. The assembler takes care of this by using an 8-bit displacement (MOD_01) of 00H whenever the [BP] addressing mode appears in an instruction. This means that the [BP] addressing mode assembles as a [BP+0], even though a [BP] is used in the instruction. The same special addressing mode is also available for the 32-bit mode.

Figure 4–6 shows the binary bit pattern required to encode the MOV [1000H],DL instruction in machine language. If the individual translating this symbolic instruction into machine language does not know about the special addressing mode, the instruction would incorrectly translate to a MOV [BP],DL instruction.

Figure 4–7 shows the actual form of the MOV [BP],DL instruction. Notice that this is a 3-byte instruction with a displacement of 00H.

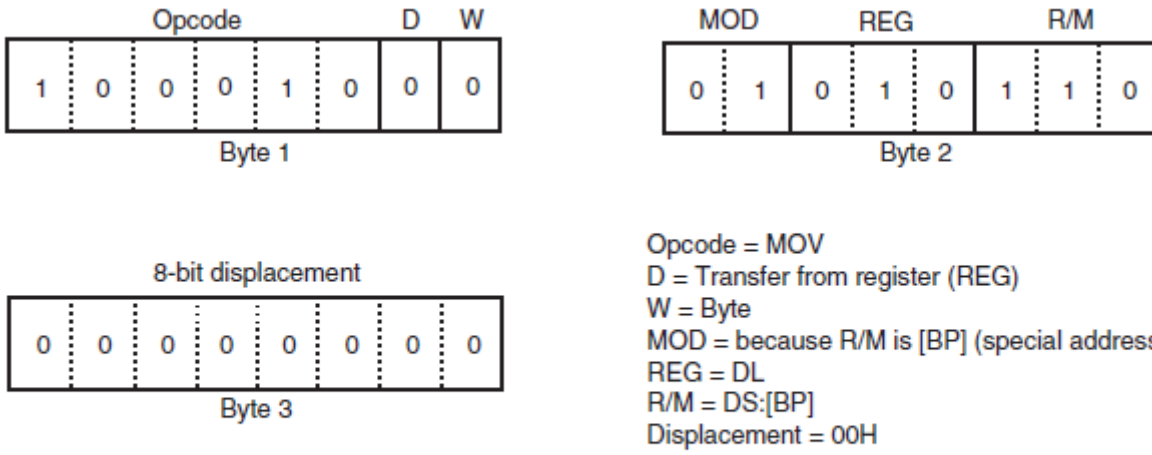
Opcode = MOV
D = Transfer to register (REG)
W = Byte
MOD = No displacement
REG = DL
R/M = DS:[DI]

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



Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 1000H

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



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

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

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

*Note: See text section, Special Addressing Mode.

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

32-Bit Addressing Modes The 32-bit addressing modes found in the 80386 and above are obtained by either running these machines in the 32-bit instruction mode or in the 16-bit instruction mode by using the address-size prefix 67H. Table 4–5 shows the coding for R/M used to specify the 32-bit addressing modes. Notice that when R/M=100, an additional byte called a scaled-index byte appears in the instruction. The scaled-index byte indicates the additional forms of scaled-index addressing that do not appear in Table 4–5. The scaled-index byte is mainly used when two registers are added to specify the memory address in an instruction. Because the scaled-index byte is added to the instruction, there are 7 bits in the opcode and 8 bits in the scaled-index byte to define. This means that a scaled-index instruction has 2^{15} (32K) possible combinations. There are over 32,000 different variations of the MOV instruction alone in the 80386 through the Core2 microprocessors. Figure 4–8 shows the format of the scaled-index byte as selected by a value of 100 in the R/M field of an instruction when the 80386 and above use a 32-bit address. The leftmost 2 bits select a scaling factor (multiplier) of 1x, 2x, 4x or 8x. Note that a scaling factor of 1x is implicit if none is used in an instruction that contains two 32-bit indirect address registers. The index and base fields both contain register numbers, as indicated in Table 4–3 for 32-bit registers.

The instruction MOV EAX,[EBX+4*ECX] is encoded as 67668B048BH. Notice that both the *address size* (67H) and *register size* (66H) override prefixes appear in the instruction. This coding (67668B048BH) is used when the 80386 and above microprocessors are operated in the 16-bit instruction mode for this instruction. If the microprocessor operates in the 32-bit instruction mode, both prefixes disappear and the instruction becomes an 8B048BH instruction.

The use of the prefixes depends on the mode of operation of the microprocessor. Scaled-index addressing can also use a single register multiplied by a scaling factor. An example is the MOV AL,[2*ECX] instruction. The contents of the data segment location addressed by two times ECX are copied into AL.

An Immediate Instruction. Suppose that the MOV WORD PTR [BX+1000H],1234H instruction is chosen as an example of a 16-bit instruction using immediate addressing. This instruction moves a 1234H into the word-sized memory location addressed by the sum of 1000H, BX, and DS X 10H This 6-byte instruction uses 2 bytes for the opcode, W, MOD, and R/M fields. Two of the 6 bytes are the data of 1234H; 2 of the 6 bytes are the displacement of 1000H. Figure 4–9 shows the binary bit pattern for each byte of this instruction.

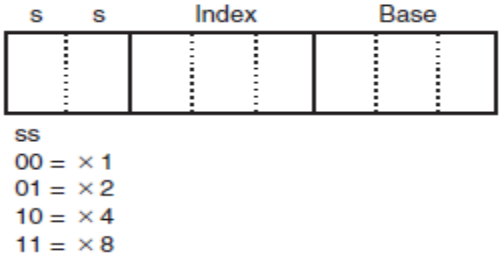
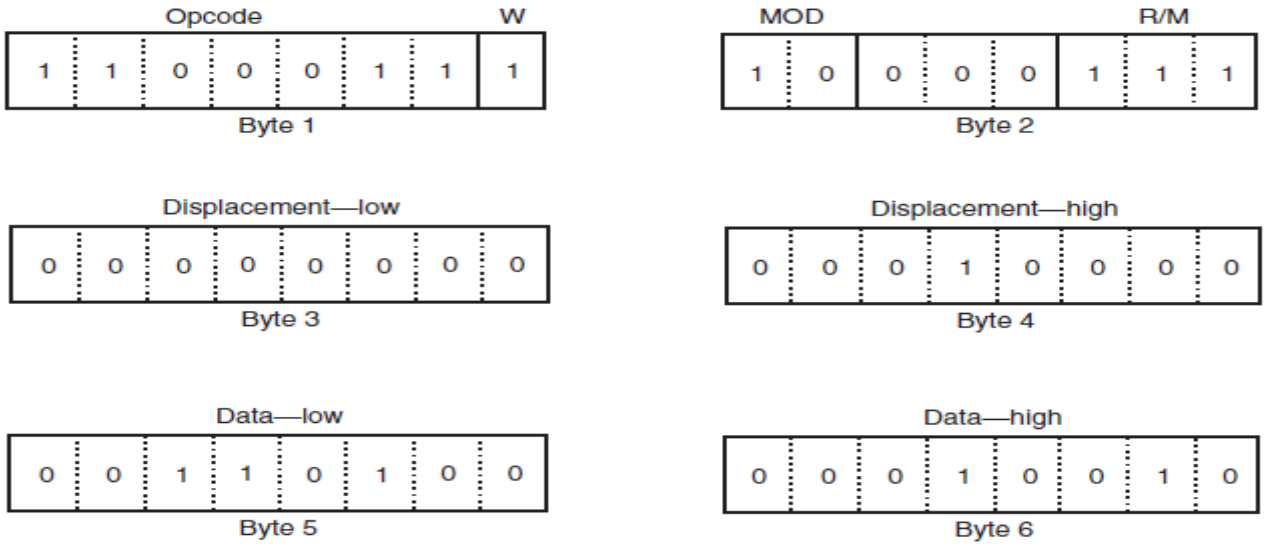


FIGURE 4–8 The scaled-index byte.



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

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

This instruction, in symbolic form, includes WORD PTR. The WORD PTR directive indicates to the assembler that the instruction uses a word-sized memory pointer. If the instruction moves a byte of immediate data, BYTE PTR replaces WORD PTR in the instruction. Likewise, if the instruction uses a doubleword of immediate data, the DWORD PTR directive replaces BYTE PTR. Most instructions that refer to memory through a pointer do not need the BYTE PTR, WORD PTR, or DWORD PTR directives. These directives are necessary only when it is not clear whether the operation is a byte, word, or doubleword. The MOV [BX],AL instruction is clearly a byte move; the MOV [BX],9 instruction is not exact, and could therefore be a byte-, word-, or doubleword-sized move. Here, the instruction must be coded as MOV BYTE PTR [BX],9, MOV WORD PTR [BX],9, or MOV DWORD PTR [BX],9. If not, the assembler flags it as an error because it cannot determine the intent of the instruction.

Segment MOV Instructions. If the contents of a segment register are moved by the MOV, PUSH, or POP instructions, a special set of register bits (REG field) selects the segment register (see Table 4–6).
Figure 4–10 shows a MOV BX,CS instruction converted to binary. The opcode for this type of MOV instruction is different for the prior MOV instructions. Segment registers can be moved between any 16-bit register or 16-bit memory location. For example, the MOV [DI],DS instruction stores the contents of DS into the memory location addressed by DI in the data segment. An immediate segment register MOV is not available in the instruction set. To load a segment register with immediate data, first load another register with the data and then move it to a segment register.
Although this discussion has not been a complete coverage of machine language coding, it provides enough information for machine language programming. Remember that a program written in symbolic assembly language *(assembly language)* is rarely assembled by hand into binary machine language. An assembler program converts symbolic assembly language into machine language. With the microprocessor and its over 100,000 instruction variations, let us hope that an assembler is available for the conversion, because the process is very time-consuming, although not impossible.

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

*Note: MOV CS,R/M and POP CS are not allowed.

TABLE 4–6 Segment register selection.

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

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

The 64-Bit Mode for the Pentium 4 and Core2

None of the information presented thus far addresses the issue of 64-bit operation of the Pentium 4 or Core2. In the 64-bit mode, an additional prefix called REX (*register extension*) is added. The REX prefix, which is encoded as a 40H–4FH, follows other prefixes and is placed immediately before the opcode to modify it for 64-bit operation. The purpose of the REX prefix is to modify the reg and r/m fields in the second byte of the instruction. REX is needed to be able to address registers R8 through R15. Figure 4–11 illustrates the structure of REX and also its application to the second byte of the opcode.

The register and memory address assignments for the rrrr and mmmm fields are shown in Table 4–7 for 64-bit operations. The reg field can only contain register assignments as in other modes of operation and the r/m field contains either a register or memory assignment. Figure 4–12 shows the scaled-index byte with the REX prefix for more complex addressing modes and also for using a scaling factor in the 64-bit mode of operation. As with 32-bit instructions, the modes allowed by the scaled-index byte are fairly all conclusive allowing pairs of registers to address memory and also an index factor of 2X, 4X or 8X . An example is the instruction MOV RAXW,[RDX+RCX-12], which requires the scaled-index byte with an index of 1, which is understood but never entered into the instruction.

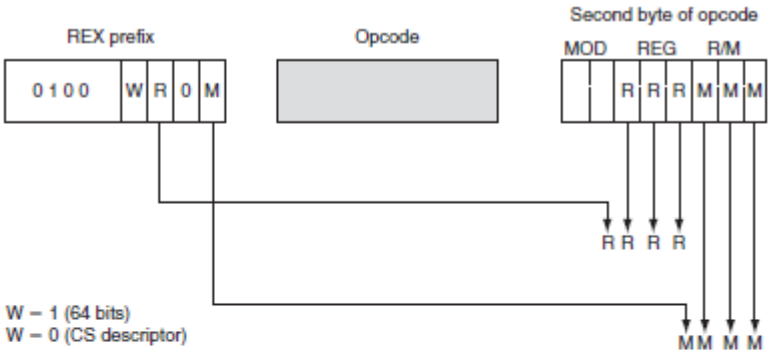


FIGURE 4–11 The application of REX without scaled index.

Code	*Register*	*Memory*
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP	See note
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

Note: This addressing mode specifies the inclusion of the scaled-index byte.

TABLE 4–7 The 64-bit register and memory designators for rrrr and mmmm.

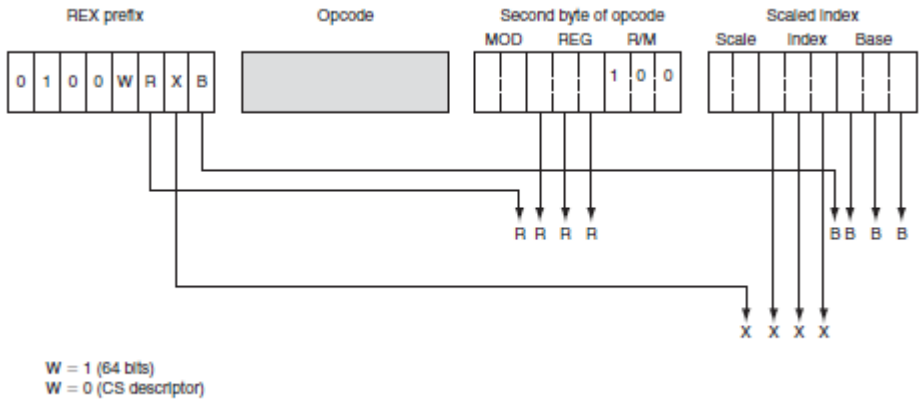


FIGURE 4–12 The scaled-index byte and REX prefix for 64-bit operations.

PUSH/POP

The PUSH and POP instructions are important instructions that *store* and *retrieve* data from the LIFO (last-in, first-out) stack memory. The microprocessor has six forms of the PUSH and POP instructions: register, memory, immediate, segment register, flags, and all registers. The PUSH and POP immediate and the PUSHA and POPA (all registers) forms are not available in the earlier 8086/8088 microprocessors, but are available to the 80286 through the Core2.

Register addressing allows the contents of any 16-bit register to be transferred to or from the stack. In the 80386 and above, the 32-bit extended registers and flags (EFLAGS) can also be pushed or popped from the stack. Memory-addressing PUSH and POP instructions store the contents of a 16-bit memory location (or 32 bits in the 80386 and above) on the stack or stack data into a memory location. Immediate addressing allows immediate data to be pushed onto the stack, but not popped off the stack. Segment register addressing allows the contents of any segment register to be pushed onto the stack or removed from the stack (ES may be pushed, but data from the stack may never be popped into ES). The flags may be pushed or popped from that stack, and the contents of all the registers may be pushed or popped.

PUSH

The 8086–80286 PUSH instruction always transfers 2 bytes of data to the stack; the 80386 and above transfer 2 or 4 bytes, depending on the register or size of the memory location. The source of the data may be any internal 16- or 32-bit register, immediate data, any segment register, or any 2 bytes of memory data. There is also a PUSHA instruction that copies the contents of the internal register set, except the segment registers, to the stack. The PUSHA (**push all**) instruction copies the registers to the stack in the following

order: AX, CX, DX, BX, SP, BP, SI, and DI. The value for SP that is pushed onto the stack is whatever it was before the PUSHA instruction executed. The PUSHF (**push flags**) instruction copies the contents of the flag register to the stack. The PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set found in the 80386 through the Pentium 4. The PUSHA and POPA instructions do not function in the 64-bit mode of operation for the Pentium 4.

Whenever data are pushed onto the stack, the first (most-significant) data byte moves to the stack segment memory location addressed by SP - 1. The second (least-significant) data byte moves into the stack segment memory location addressed by SP - 2. After the data are stored by a PUSH, the contents of the SP register decrement by 2. The same is true for a doubleword push, except that 4 bytes are moved to the stack memory (most-significant byte first), after which the stack pointer decrements by 4. Figure 4–13 shows the operation of the PUSH AX instruction. This instruction copies the contents of AX onto the stack where address SS:[SP – 1] = AH, SS:[SP – 2] = AL, and afterwards SP = SP - 2 . In 64-bit mode, 8 bytes of the stack are used to store the number pushed onto the stack. The PUSHA instruction pushes all the internal 16-bit registers onto the stack, as illustrated in Figure 4–14. This instruction requires 16 bytes of stack memory space to store all eight 16-bit registers. After all registers are pushed, the contents of the SP register are decremented by 16. The PUSHA instruction is very useful when the entire register set (microprocessor environment) of the 80286 and above must be saved during a task. The PUSHAD instruction places the 32-bit register set on the stack in the 80386 through the Core2. PUSHAD requires 32 bytes of stack storage space.

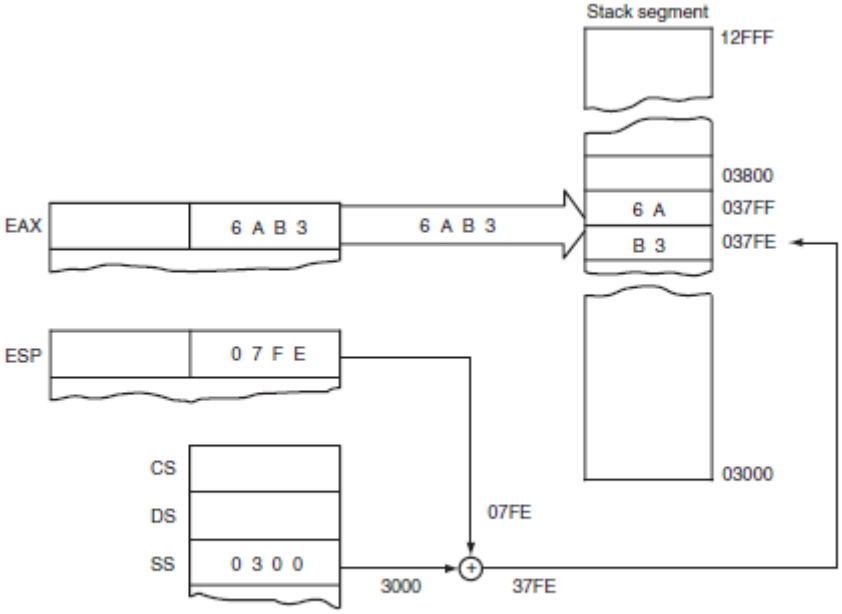


FIGURE 4–13 The effect of the PUSH AX instruction on ESP and stack memory locations 37FFH and 37FEH. This instruction is shown at the point after execution.

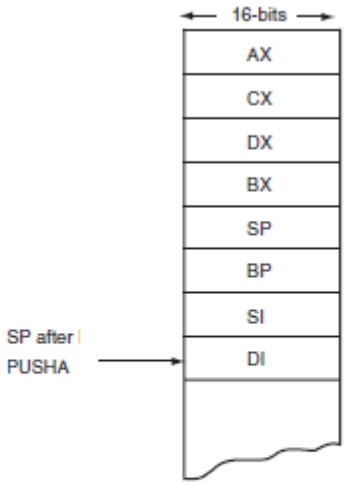


FIGURE 4–14 The operation of the PUSHA instruction, showing the location and order of stack data.

The PUSH immediate data instruction has two different opcodes, but in both cases, a 16-bit immediate number moves onto the stack; if PUSHD is used, a 32-bit immediate datum is pushed. If the values of the immediate data are 00H–FFH, the opcode is a 6AH; if the data are 0100H–FFFFH, the opcode is 68H. The PUSH 8 instruction, which pushes 0008H onto the stack, assembles as 6A08H. The PUSH 1000H instruction assembles as 680010H. Another example of PUSH immediate is the PUSH ‘A’ instruction, which pushes a 0041H onto the stack. Here, the 41H is the ASCII code for the letter A.

Table 4–8 lists the forms of the PUSH instruction that include PUSHA and PUSHF. Notice how the instruction set is used to specify different data sizes with the assembler.

Symbolic	Example	Note
PUSH reg16	PUSH BX	16-bit register
PUSH reg32	PUSH EDX	32-bit register
PUSH mem16	PUSH WORD PTR[BX]	16-bit pointer
PUSH mem32	PUSH DWORD PTR[EBX]	32-bit pointer
PUSH mem64	PUSH QWORD PTR[RBX]	64-bit pointer (64-bit mode)
PUSH seg	PUSH DS	Segment register
PUSH imm8	PUSH 'R'	8-bit immediate
PUSH imm16	PUSH 1000H	16-bit immediate
PUSHD imm32	PUSHD 20	32-bit immediate
PUSHA	PUSHA	Save all 16-bit registers
PUSHAD	PUSHAD	Save all 32-bit registers
PUSHF	PUSHF	Save flags
PUSHFD	PUSHFD	Save EFLAGS

TABLE 4–8 The PUSH instruction.

POP

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

Suppose that a POP BX instruction executes. The first byte of data removed from the stack (the memory location addressed by SP in the stack segment) moves into register BL. The second byte is removed from stack segment memory location SP + 1 and is placed into register BH.

After both bytes are removed from the stack, the SP register is incremented by 2. Figure 4–15 shows how the POP BX instruction removes data from the stack and places them into register BX. The opcodes used for the POP instruction and all of its variations appear in Table 4–9. Note that a POP CS instruction is not a valid instruction in the instruction set. If a POP CS instruction executes, only a portion of the address (CS) of the next instruction changes. This makes the POP CS instruction unpredictable and therefore not allowed.

Initializing the Stack

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

For example, if the stack segment is to reside in memory locations 10000H–1FFFFH, load SS with a 1000H. (Recall that the rightmost end of the stack segment register is appended with a 0H for real mode addressing.) To start the stack at the top of this 64K-byte stack segment, the stack pointer (SP) is loaded with a 0000H. Likewise, to address the top of the stack at location 10FFFH, use a value of 1000H in SP. Figure 4–16 shows how this value causes data to be pushed onto the top of the stack segment with a PUSH CX instruction. Remember that all segments are cyclic in nature—that is, the top location of a segment is contiguous with the bottom location of the segment.

In assembly language, a stack segment is set up as illustrated in Example 4–1. The first statement identifies the start of the stack segment and the last statement identifies the end of the stack segment. The assembler and linker programs place the correct stack segment address in SS and the length of the segment (top of the stack) into SP. There is no need to load these registers in your program unless you wish to change the initial values for some reason.

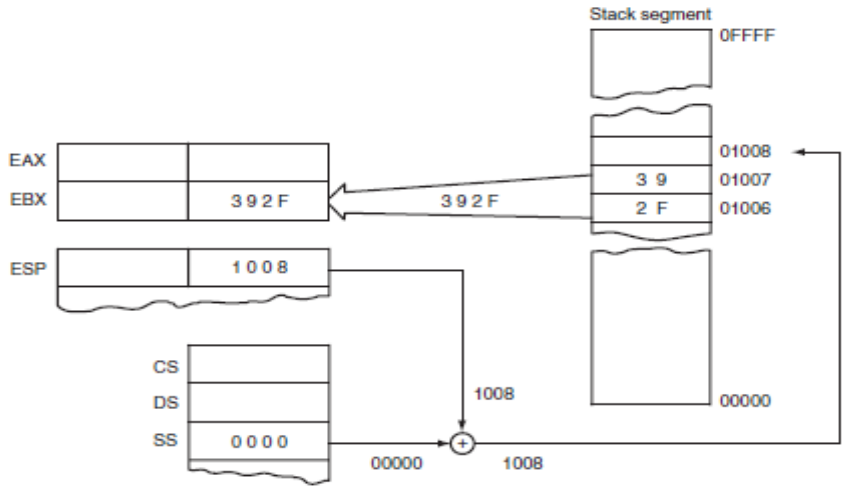


FIGURE 4–15 The POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.

TABLE 4–9 The POP instructions.

Symbolic	*Example*	*Note*
POP reg16	POP CX	16-bit register
POP reg32	POP EBP	32-bit register
POP mem16	POP WORD PTR[BX+1]	16-bit pointer
POP mem32	POP DATA3	32-bit memory address
POP mem64	POP FROG	64-bit memory address (64-bit mode)
POP seg	POP FS	Segment register
POPA	POPA	Pops all 16-bit registers
POPAD	POPAD	Pops all 32-bit registers
POPF	POPF	Pops flags
POPFD	POPFD	Pops EFLAGS

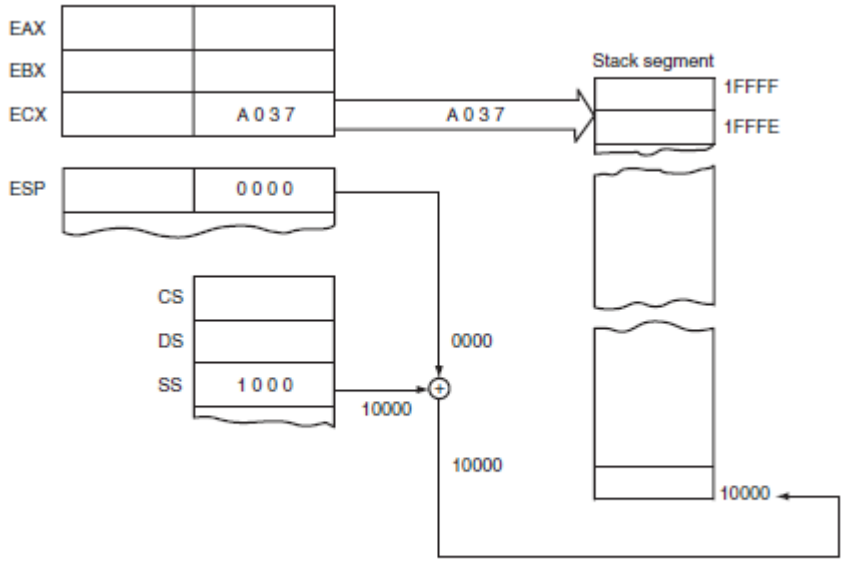


FIGURE 4–16 The PUSH CX instruction, showing the cyclical nature of the stack segment. This instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.

EXAMPLE 4–1

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

EXAMPLE 4–2

```
.MODEL SMALL
.STACK 200H  ;set stack size
```

An alternative method for defining the stack segment is used with one of the memory models for the MASM assembler only (refer to Appendix A). Other assemblers do not use models; if they do, the models are not exactly the same as with MASM. Here, the .STACK statement, followed by the number of bytes allocated to the stack, defines the stack area (see Example 4–2).

The function is identical to Example 4–1. The .STACK statement also initializes both SS and SP. Note that this text uses memory models that are designed for the Microsoft Macro Assembler program MASM.
If the stack is not specified by using either method, a warning will appear when the program is linked. The warning may be ignored if the stack size is 128 bytes or fewer. The system automatically assigns (through DOS) at least 128 bytes of memory to the stack. This memory section is located in the **program segment prefix** (PSP), which is appended to the beginning of each program file. If you use more memory for the stack, you will erase information in the PSP that is critical to the operation of your program and the computer. This error often causes the computer program to crash. If the TINY memory model is used, the stack is automatically located at the very end of the segment, which allows for a larger stack area.

LOAD-EFFECTIVE ADDRESS

There are several load-effective address instructions in the microprocessor instruction set. The LEA instruction loads any 16-bit register with the offset address, as determined by the addressing mode selected for the instruction. The LDS and LES variations load any 16-bit register with the offset address retrieved from a memory location, and then load either DS or ES with a segment address retrieved from memory. In the 80386 and above, LFS, LGS, and LSS are added to the instruction set, and a 32-bit register can be selected to receive a 32-bit offset from memory. In the 64-bit mode for the Pentium 4, the LDS and LES instructions are *invalid* and not used because the segments have no function in the flat memory model. Table 4–10 lists the load-effective address instructions.

LEA

The LEA instruction loads a 16- or 32-bit register with the offset address of the data specified by the operand. As the first example in Table 4–9 shows, the operand address NUMB is loaded into register AX, not the contents of address NUMB.
By comparing LEA with MOV, we observe that LEA BX,[DI] loads the offset address specified by [DI] (contents of DI) into the BX register; MOV BX,[DI] loads the data stored at the memory location addressed by [DI] into register BX.
Earlier in the text, several examples were presented by using the OFFSET directive. The OFFSET directive performs the same function as an LEA instruction if the operand is a displacement. For example, the MOV BX,OFFSET LIST performs the same function as LEA BX,LIST. Both instructions load the offset address of memory location LIST into the BX register. See Example 4–3 for a short program that loads SI with the address of DATA1 and DI with the address of DATA2. It then exchanges the contents of these memory locations. Note that the LEA and MOV with OFFSET instructions are both the same length (3 bytes).

EXAMPLE 4–3

```
                        .MODEL SMALL          ;select small model
0000                    .DATA                 ;start data segment
0000 2000       DATA1   DW     2000H          ;define DATA1
0002 3000       DATA2   DW     3000H          ;define DATA2
0000                    .CODE                 ;start code segment
                        .STARTUP              ;start program
0017 BE 0000 R          LEA SI,DATA1          ;address DATA1 with SI
001A BF 0002 R          MOV DI,OFFSET DATA2   ;address DATA2 with DI
001D 8B 1C              MOV BX,[SI]           ;exchange DAT1 with DATA2
001F 8B 0D              MOV CX,[DI]
0021 89 0C              MOV [SI],CX
0023 89 1D              MOV [DI],BX
                        .EXIT
                        END
```

TABLE 4–10 Load-effective address instructions.

Assembly Language	*Operation*
LEA AX,NUMB	Loads AX with the offset address of NUMB
LEA EAX,NUMB	Loads EAX with the offset address of NUMB
LDS DI,LIST	Loads DS and DI with the 32-bit contents of data segment memory location LIST
LDS EDI,LIST1	Loads the DS and EDI with the 48-bit contents of data segment memory location LIST1
LES BX,CAT	Loads ES and BX with the 32-bit contents of data segment memory location CAT
LFS DI,DATA1	Loads FS and DI with the 32-bit contents of data segment memory location DATA1
LGS SI,DATA5	Loads GS and SI with the 32-bit contents of data segment memory location DATA5
LSS SP,MEM	Loads SS and SP with the 32-bit contents of data segment memory location MEM

But why is the LEA instruction available if the OFFSET directive accomplishes the same task? First, OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI], and so on. The OFFSET directive is more efficient than the LEA instruction for simple operands. It takes the microprocessor longer to execute the LEA BX,LIST instruction than the MOV BX,OFFSET LIST. The 80486 microprocessor, for example, requires two clocks to execute the LEA BX,LIST instruction and only one clock to execute MOV BX,OFFSET LIST. The reason that the MOV BX,OFFSET LIST instruction executes faster is because the assembler calculates the offset address of LIST, whereas the microprocessor calculates the address for the LEA instruction. The MOV BX,OFFSET LIST instruction is actually assembled as a move immediate instruction and is more efficient.

Suppose that the microprocessor executes an LEA BX,[DI] instruction and DI contains a 1000H. Because DI contains the offset address, the microprocessor transfers a copy of DI into BX. A MOV BX,DI instruction performs this task in less time and is often preferred to the LEA BX,[DI] instruction.

Another example is LEA SI,[BX+DI]. This instruction adds BX to DI and stores the sum in the SI register. The sum generated by this instruction is a modulo-64K sum. (A modulo-64K sum drops any carry out of the 16-bit result.) If BX = 1000H and DI = 2000H, the offset address moved into SI is 3000H. If BX = 1000H and DI = FF00H, the offset address is 0F00H instead of 10F00H. Notice that the second result is a modulo-64K sum of 0F00H.

LDS, LES, LFS, LGS, and LSS

The LDS, LES, LFS, LGS, and LSS instructions load any 16-bit or 32-bit register with an offset address, and the DS, ES, FS, GS, or SS segment register with a segment address. These instructions use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. The 32-bit section of memory contains a 16-bit offset and segment address, while the 48-bit section contains a 32-bit offset and a segment address. These instructions may not use the register addressing mode (MOD=11). Note that the LFS, LGS, and LSS instructions are only available on 80386 and above, as are the 32-bit registers.

Figure 4–17 illustrates an example LDS BX,[DI] instruction. This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers. The LDS, LES, LFS, LGS, and LSS instructions obtain a new far address from memory. The offset address appears first, followed by the segment address. This format is used for storing all 32-bit memory addresses.

A far address can be stored in memory by the assembler. For example, the ADDR DD FAR PTR FROG instruction stores the offset and segment address (far address) of FROG in 32 bits of memory at location ADDR. The DD directive tells the assembler to store a doubleword (32-bit number) in memory address ADDR.

In the 80386 and above, an LDS EBX,[DI] instruction loads EBX from the 4-byte section of memory addressed by DI in the data segment. Following this 4-byte offset is a word that is loaded to the DS register. Notice that instead of addressing a 32-bit section of memory, the 80386 and above address a 48-bit section of the memory whenever a 32-bit offset address is loaded to a 32-bit register. The first 4 bytes contain the offset value loaded to the 32-bit register and the last 2 bytes contain the segment address.

The most useful of the load instructions is the LSS instruction. Example 4–4 shows a short program that creates a new stack area after saving the address of the old stack area. After executing some dummy instructions, the old stack area is reactivated by loading both SS and SP with the LSS instruction. Note that the CLI (**disable interrupt**) and STI (**enable interrupt**) instructions must be included to disable interrupts. (This topic is discussed near the end of this chapter.) Because the LSS instruction functions in the 80386 or above, the .386 statement appears after the .MODEL statement to select the 80386 microprocessor. Notice how the WORD PTR directive is used to override the doubleword (DD) definition for the old stack memory location. If an 80386 or newer microprocessor is in use, it is suggested that the .386 switch be used to develop software for the 80386 microprocessor. This is true even if the microprocessor is a Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, or Core2. The reason is that the 80486–Core2 microprocessors add only a few additional instructions to the 80386 instruction set, which are seldom used

in software development. If the need arises to use any of the CMPXCHG, CMPXCHG8 (new to the Pentium), XADD or BSWAP instructions, select either the .486 switch for the 80486 microprocessor of the .586 switch for the Pentium. You can even specify the Pentium II–Core2 using the .686 switch.

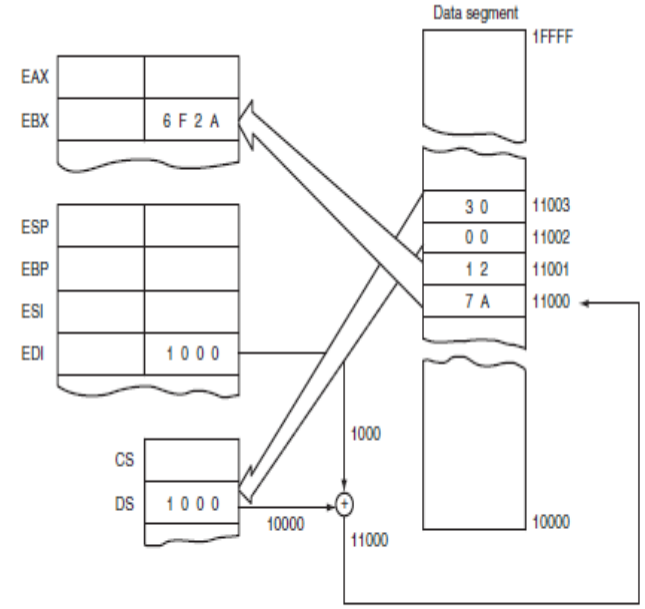


FIGURE 4–17 The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

EXAMPLE 4–4

```
                        .MODEL SMALL            ;select small model
                        .386                    ;select 80386
0000                    .DATA                   ;start data segment
0000 00000000       $ADDR  DD     ?              ;old stack address
0004 1000 [         $AREA  DW     1000H DUP(?)   ;new stack area
             ????
                   ]
2004 = 2004         STOP   EQU THIS WORD         ;define top of new stack
0000                    .CODE                   ;start code segment
                        .STARTUP                ;start program
0010 FA                 CLI                     ;disable interrupts
0011 8B C4              MOV AX,SP               ;save old SP
0013 A3 0000 R          MOV WORD PTR $ADDR,AX
0016 8C D0              MOV AX,SS               ;save old SS
0018 A3 0002 R          MOV WORD PTR $ADDR+2,AX

001B 8C D8              MOV AX,DS               ;load new SS
001D 8E D0              MOV SS,AX
001F B8 2004 R          MOV AX,OFFSET STOP      ;load new SP
0022 8B E0              MOV SP,AX
0024 FB                 STI                     ;enable interrupts

0025 8B C0              MOV AX,AX               ;do some dummy instructions
0027 8B C0              MOV AX,AX
0029 9F B2 26 0000 R    LSS SP,$ADDR            ;get old stack
                        .EXIT                   ;exit to DOS
                        END                     ;end program listing
```

STRING DATA TRANSFERS

There are five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS. Each string instruction allows data transfers that are either a single byte, word, or doubleword (or if repeated, a block of bytes, words, or doublewords). Before the string instructions are presented, the operation of the D flag-bit (direction), DI, and SI must be understood as they apply to the string instructions. In the 64-bit mode of the Pentium 4 and Core2, quadwords are also used with the string instructions such as LODSQ.

The Direction Flag

The direction flag (D, located in the flag register) selects the auto-increment (D=0) or the auto-decrement (D=1) operation for the DI and SI registers during string operations. The direction flag is used only with the string instructions. The CLD instruction clears the D Flag (D=0) and the STD instruction sets it (D=1). Therefore, the CLD instruction selects the auto-increment mode (D=0) and STD selects the auto-decrement mode (D=1). Whenever a string instruction transfers a byte, the contents of DI and/or SI are incremented or decremented by 1. If a word is transferred, the contents of DI and/or SI are incremented or decremented by 2. Doubleword transfers cause DI and/or SI to increment or decrement by 4. Only the actual registers used by the string instruction are incremented or decremented. For example, the STOSB instruction uses the DI register to address a memory location. When STOSB executes, only the DI register is incremented or decremented without affecting SI. The same is true of the LODSB instruction, which uses the SI register to address memory data. A LODSB instruction will only increment or decrement SI without affecting DI.

DI and SI

During the execution of a string instruction, memory accesses occur through either or both of the DI and SI registers. The DI offset address accesses data in the extra segment for all string instructions that use it. The SI offset address accesses data, by default, in the data segment. The segment assignment of SI may be changed with a segment override prefix, as described later in this chapter. The DI segment assignment is always in the extra segment when a string instruction executes. This assignment cannot be changed. The reason that one pointer addresses data in the extra segment and the other in the data segment is so that the MOVS instruction can move 64K bytes of data from one segment of memory to another.

When operating in the 32-bit mode in the 80386 microprocessor or above, the EDI and ESI registers are used in place of DI and SI. This allows string using any memory location in the entire 4G-byte protected mode address space of the microprocessor.

LODS

The LODS instruction loads AL, AX, or EAX with data stored at the data segment offset address indexed by the SI register. (Note that only the 80386 and above use EAX.) After loading AL with a byte, AX with a word, or EAX with a doubleword, the contents of SI increment, if D=0 or decrement, if D=1. A 1 is added to or subtracted from SI for a byte-sized LODS, a 2 is added or subtracted for a word-sized LODS, and a 4 is added or subtracted for a doubleword-sized LODS.

Table 4–11 lists the permissible forms of the LODS instruction. The LODSB (loads a byte) instruction causes a byte to be loaded into AL, the LODSW (loads a word) instruction causes a word to be loaded into AX, and the LODSD (loads a doubleword) instruction causes a doubleword to be loaded into EAX. Although rare, as an alternative to LODSB, LODSW, LODSD, and LODSQ, the LODS instruction may be followed by a byte-, word- or doubleword-sized operand to select a byte, word, or doubleword transfer. Operands are often defined as bytes with DB, as words with DW, and as doublewords with DD. The DB pseudo-operation defines byte(s), the DW pseudo-operation defines word(s), and the DD pseudo-operations define doubleword(s).

Figure 4–18 shows the effect of executing the LODSW instruction if the D flag=0, SI=1000H, and DS=1000H. Here, a 16-bit number stored at memory locations 11000H and 11001H moves into AX. Because D = 0 and this is a word transfer, the contents of SI increment by 2 after AX loads with memory data.

Assembly Language	*Operation*
LODSB	AL = DS:[SI]; SI = SI ± 1
LODSW	AX = DS:[SI]; SI = SI ± 2
LODSD	EAX = DS:[SI]; SI = SI ± 4
LODSQ	RAX = [RSI]; RSI = RSI ± 8 (64-bit mode)
LODS LIST	AL = DS:[SI]; SI = SI ± 1 (if LIST is a byte)
LODS DATA1	AX = DS:[SI]; SI = SI ± 2 (if DATA1 is a word)
LODS FROG	EAX = DS:[SI]; SI = SI ± 4 (if FROG is a doubleword)

Note: The segment register can be overridden with a segment override prefix as in LODS ES:DATA4.

TABLE 4–11 Forms of the LODS instruction.

STOS

The STOS instruction stores AL, AX, or EAX at the extra segment memory location addressed by the DI register. (Note that only the 80386–Core2 use EAX and doublewords.) Table 4–12 lists all forms of the STOS instruction. As with LODS, a STOS instruction may be appended with a B, W, or D for byte, word, or doubleword transfers. The STOSB (**stores a byte**) instruction stores the byte in AL at the extra segment memory location addressed by DI. The STOSW (**stores a word**) instruction stores AX in the extra segment memory location addressed by DI. A doubleword is stored in the extra segment location addressed by DI with the STOSD (**stores a doubleword)** instruction. After the byte (AL), word (AX), or doubleword (EAX) is stored, the contents of DI increment or decrement.

STOS with a REP. The **repeat prefix** (REP) is added to any string data transfer instruction, except the LODS instruction. It doesn't make any sense to perform a repeated LODS operation. The REP prefix causes CX to decrement by 1 each time the string instruction executes. After CX decrements, the string instruction repeats. If CX reaches a value of 0, the instruction terminates and the program continues with the next sequential instruction. Thus, if CX is loaded with 100 and a REP STOSB instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times. Because the DI register is automatically incremented or decremented after each datum is stored, this instruction stores the contents of AL in a block of memory instead of a single byte of memory. In the Pentium 4 operated in 64-bit mode, the RCX register is used with the REP prefix.

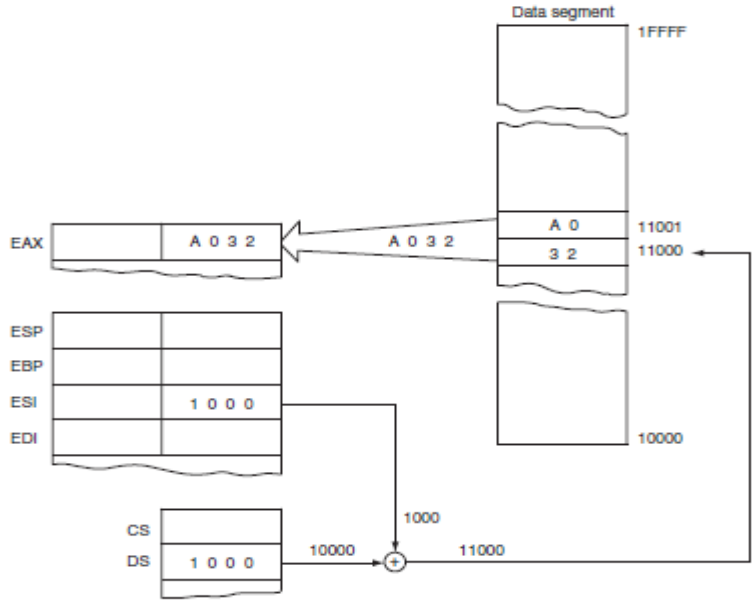


FIGURE 4–18 The operation of the LODSW instruction if DS = 1000H, D = 0, 11000H = 32, and 11001H = A0. This instruction is shown after AX is loaded from memory, but before SI increments by 2.

Suppose that the STOSW instruction is used to clear an area of memory called Buffer using a count called Count and the program is to function call ClearBuffer in the C++ environment using the inline assembler. (See Example 4–5.) Note that both the Count and Buffer address are transferred to the function. The REP STOSW instruction clears the memory buffer called Buffer. Notice that Buffer is a pointer to the actual buffer that is cleared by this function.

Assembly Language	*Operation*
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4
STOSQ	[RDI] = RAX; RDI = RDI ± 8 (64-bit mode)
STOS LIST	ES:[DI] = AL; DI = DI ± 1 (if LIST is a byte)
STOS DATA3	ES:[DI] = AX; DI = DI ± 2 (if DATA3 is a word)
STOS DATA4	ES:[DI] = EAX; DI = DI ± 4 (if DATA4 is a doubleword)

TABLE 4–12 Forms of the STOS instruction.

EXAMPLE 4–5

```
void ClearBuffer (int count, short* buffer)
{
   _asm{
         push edi             ;save registers
         push es
         push ds
         mov  ax,0
         mov  ecx, count
         mov  edi, buffer
         pop  es              ;load ES with DS
         rep  stosw           ;clear Buffer
         pop  es              ;restore registers
         pop  edi
   }
}
```

The operands in a program can be modified by using arithmetic or logic operators such as multiplication (*). Other operators appear in Table 4–13.

MOVS

One of the more useful string data transfer instructions is MOVS, because it transfers data from one memory location to another. This is the only memory-to-memory transfer allowed in the 8086–Pentium 4 microprocessors. The MOVS instruction transfers a byte, word, or doubleword from the data segment location addressed by SI to the extra segment location addressed by SI. As with the other string instructions, the pointers then are incremented or decremented, as dictated by the direction flag. Table 4–14 lists all the permissible forms of the MOVS instruction. Note that only the source operand (SI), located in the data segment, may be overridden so that another segment may be used. The destination operand (DI) must always be located in the extra segment.

It is often necessary to transfer the contents of one area of memory to another. Suppose that we have two blocks of doubleword memory, blockA and blockB, and we need to copy blockA into blockB. This can be accomplished using the MOVSD instruction as illustrated in Example 4–6, which is a C++ language function written using the inline assembler. The function receives three pieces of information from the caller: blockSize and the addresses of blockA and blockB. Note that all data are in the data segment in a Visual C++ program so we need to copy DS into ES, which is done using a PUSH DS followed by a POP ES. We also need to save all registers that we changed except for EAX, EBX, ECX, and EDX.

Example 4–7 shows the same function written in C++ exclusively, so the two methods can be compared and contrasted. Example 4–8 shows the assembly language version of Example 4–7 for comparison to Example

4–6. Notice how much shorter the assembly language version is compared to the C++ version generated in Example 4–8. Admittedly the C++ version is a little easier to type, but if execution speed is important, Example 4–6 will run much faster than Example 4–7.

TABLE 4–13 Common operand modifiers.

Operator	*Example*	*Comment*
+	MOV AL,6+3	Copies 9 into AL
–	MOV AL,6–3	Copies 3 into AL
*	MOV AL,4*3	Copies 12 into AL
/	MOV AX,12/5	Copies 2 into AX (remainder is lost)
MOD	MOV AX,12 MOD 7	Copies 5 into AX (quotient is lost)
AND	MOV AX,12 AND 4	Copies 4 into AX (1100 AND 0100 = 0100)
OR	MOV EAX,12 OR 1	Copies 13 into EAX (1100 OR 0001 = 1101)
NOT	MOV AL,NOT 1	Copies 254 into AL (NOT 0000 0001 = 1111 1110 or 254)

TABLE 4–14 Forms of the MOVS instruction.

Assembly Language	*Operation*
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred)
MOVSQ	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8 (64-bit mode)
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred if WORD1 and WORD2 are words)
MOVS TED,FRED	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred if TED and FRED are doublewords)

EXAMPLE 4–6

```
//Function that copies blockA into blockB using the inline assembler
//
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
      _asm{
            push es                 ;save registers
            push edi
            push esi
            push ds                 ;copy DS into ES
            pop  es
            mov  esi, blockA        ;address blockA
            mov  edi, blockB        ;address blockB
            mov  ecx, blockSize     ;load count
            rep  movsd              ;move data
            pop  esi
            pop  edi
            pop  es                 ;restore registers
      }
}
```

EXAMPLE 4–7

```
//C++ version of Example 4-6
//
void TransferBlocks (int blockSize, int* blockA, int* blockB)
{
      for (int a = 0; a < blockSize; a++)
      {
            blockA = blockB++;
            blockA++;
      }
}
```

EXAMPLE 4–8

```
void TransferBlocks(int blockSize, int* blockA, int* blockB)
{
004136A0  push        ebp
004136A1  mov         ebp,esp
004136A3  sub         esp,0D8h
004136A9  push        ebx
004136AA  push        esi
004136AB  push        edi
004136AC  push        ecx
004136AD  lea         edi,[ebp-0D8h]
004136B3  mov         ecx,36h
004136B8  mov         eax,0CCCCCCCCh
004136BD  rep stos    dword ptr [edi]
004136BF  pop         ecx
004136C0  mov         dword ptr [ebp-8],ecx
      for( int a = 0; a < blockSize; a++ )
004136C3  mov         dword ptr [a],0
004136CA  jmp         TransferBlocks+35h (4136D5h)
004136CC  mov         eax,dword ptr [a]
004136CF  add         eax, 1
004136D2  mov         dword ptr [a],eax
004136D5  mov         eax,dword ptr [a]
004136D8  cmp         eax,dword ptr [blockSize]
004136DB  jge         TransferBlocks+57h (4136F7h)
      {
            blockA = blockB++;
004136DD  mov         eax,dword ptr [blockB]
004136E0  mov         dword ptr [blockA],eax
004136E3  mov         ecx,dword ptr [blockB]
004136E6  add         ecx,4
004136E9  mov         dword ptr [blockB],ecx
            blockA++;
004136EC  mov         eax,dword ptr [blockA]
004136EF  add         eax,4
004136F2  mov         dword ptr [blockA],eax
      }
004136F5  jmp         TransferBlocks+2Ch (4136CCh)
}
004136F7  pop         edi
004136F8  pop         esi
004136F9  pop         ebx
004136FA  mov         esp,ebp
004136FC  pop         ebp
004136FD  ret         0Ch
```

INS

The INS (input string) instruction (not available on the 8086/8088 microprocessors) transfers a byte, word, or doubleword of data from an I/O device into the extra segment memory location addressed by the DI register. The I/O address is contained in the DX register. This instruction is useful for inputting a block of data from an external I/O device directly into the memory. One application transfers data from a disk drive to memory. Disk drives are often considered and interfaced as I/O devices in a computer system.

As with the prior string instructions, there are three basic forms of the INS. The INSB instruction inputs data from an 8-bit I/O device and stores it in the byte-sized memory location indexed by SI. The INSW instruction inputs 16-bit I/O data and stores it in a word-sized memory location. The INSD instruction inputs a doubleword. These instructions can be repeated using the REP prefix, which allows an entire block of input data to be stored in the memory from an I/O device. Table 4–15 lists the various forms of the INS instruction. Note that in the 64-bit mode there is no 64-bit input, but the memory address is 64 bits and located in RDI for the INS instructions.

Assembly Language	*Operation*
INSB	ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW	ES:[DI] = [DX]; DI = DI ± 2 (word transferred)
INSD	ES:[DI] = [DX]; DI = DI ± 4 (doubleword transferred)
INS LIST	ES:[DI] = [DX]; DI = DI ± 1 (if LIST is a byte)
INS DATA4	ES:[DI] = [DX]; DI = DI ± 2 (if DATA4 is a word)
INS DATA5	ES:[DI] = [DX]; DI = DI ± 4 (if DATA5 is a doubleword)

Note: [DX] indicates that DX is the I/O device address. These instructions are not available on the 8086 and 8088 microprocessors.

TABLE 4–15 Forms of the INS instruction.

EXAMPLE 4–9

```
                    ;Using the REP INSB to input data to a memory array
                    ;
0000 BF 0000 R            MOV DI,OFFSET LISTS     ;address array
0003 BA 03AC              MOV DX,3ACH             ;address I/O
0006 FC                   CLD                     ;auto-increment
0007 B9 0032              MOV CX,50               ;load counter
000A F3/6C                REP INSB                ;input data
```

Example 4–9 shows a sequence of instructions that inputs 50 bytes of data from an I/O device whose address is 03ACH and stores the data in extra segment memory array LISTS. This software assumes that data are available from the I/O device at all times. Otherwise, the software must check to see if the I/O device is ready to transfer data precluding the use of a REP prefix.

OUTS

The OUTS (output string) instruction (not available on the 8086/8088 microprocessors) transfers a byte, word, or doubleword of data from the data segment memory location address by SI to an I/O device. The I/O device is addressed by the DX register as it is with the INS instruction. Table 4–16 shows the variations available for the OUTS instruction. In the 64-bit mode for the Pentium 4 and Core2, there is no 64-bit output, but the address in RSI is 64 bits wide.

Example 4–10 shows a short sequence of instructions that transfer data from a data segment memory array (ARRAY) to an I/O device at I/O address 3ACH. This software assumes that the I/O device is always ready for data.

EXAMPLE 4–10

```
                    ;Using the REP OUTSB to output data from a memory array
                    ;
0000 BE 0064 R            MOV SI,OFFSET ARRAY     ;address array
0003 BA 03AC              MOV DX,3ACH             ;address I/O
0006 FC                   CLD                     ;auto-increment
0007 B9 0064              MOV CX,100              ;load counter
000A F3/6E                REP OUTSB               ;output data
```

Assembly Language	*Operation*
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (doubleword transferred)
OUTS DATA7	[DX] = DS:[SI]; SI = SI ± 1 (if DATA7 is a byte)
OUTS DATA8	[DX] = DS:[SI]; SI = SI ± 2 (if DATA8 is a word)
OUTS DATA9	[DX] = DS:[SI]; SI = SI ± 4 (if DATA9 is a doubleword)

Note: [DX] indicates that DX is the I/O device address. These instructions are not available on the 8086 and 8088 microprocessors.

TABLE 4–16 Forms of the OUTS instruction.

ARITHMETIC INSTRUCTIONS –

Addition

Addition (ADD) appears in many forms in the microprocessor. This section details the use of the ADD instruction for 8-, 16-, and 32-bit binary addition. A second form of addition, called **add-with-carry**, is introduced with the ADC instruction. Finally, the increment instruction (INC) is presented. Increment is a special type of addition that adds 1 to a number. In Section 5–3, other forms of addition are examined, such as BCD and ASCII. Also described is the XADD instruction, found in the 80486 through the Pentium 4.

Table 5–1 illustrates the addressing modes available to the ADD instruction. (These addressing modes include almost all those mentioned in Chapter 3.) However, because there are more than 32,000 variations of the ADD instruction in the instruction set, it is impossible to list them all in this table. The only types of addition *not* allowed are memory-to-memory and segment register. The segment registers can only be moved, pushed, or popped. Note that, as with all other instructions, the 32-bit registers are available only with the 80386 through the Core2. In the 64-bit mode of the Pentium 4 and Core2, the 64-bit registers are also used for addition.

TABLE 5–1 Example addition instructions.

Assembly Language	*Operation*
ADD AL,BL	AL = AL + BL
ADD CX,DI	CX = CX + DI
ADD EBP,EAX	EBP = EBP + EAX
ADD CL,44H	CL = CL + 44H
ADD BX,245FH	BX = BX + 245FH
ADD EDX,12345H	EDX = EDX + 12345H
ADD [BX],AL	AL adds to the byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location
ADD CL,[BP]	The byte contents of the stack segment memory location addressed by BP add to CL with the sum stored in CL
ADD AL,[EBX]	The byte contents of the data segment memory location addressed by EBX add to AL with the sum stored in AL
ADD BX,[SI+2]	The word contents of the data segment memory location addressed by SI + 2 add to BX with the sum stored in BX
ADD CL,TEMP	The byte contents of data segment memory location TEMP add to CL with the sum stored in CL
ADD BX,TEMP[DI]	The word contents of the data segment memory location addressed by TEMP + DI add to BX with the sum stored in BX
ADD [BX+D],DL	DL adds to the byte contents of the data segment memory location addressed by BX + DI with the sum stored in the same memory location
ADD BYTE PTR [DI],3	A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location
ADD BX,[EAX+2*ECX]	The word contents of the data segment memory location addressed by EAX plus 2 times ECX add to BX with the sum stored in BX
ADD RAX,RBX	RBX adds to RAX with the sum stored in RAX (64-bit mode)
ADD EDX,[RAX+RCX]	The doubleword in EDX is added to the doubleword addressed by the sum of RAX and RCX and the sum is stored in EDX (64-bit mode)

Register Addition. Example 5–1 shows a simple sequence of instructions that uses register addition to add the contents of several registers. In this example, the contents of AX, BX, CX, and DX are added to form a 16-bit result stored in the AX register.

EXAMPLE 5–1

```
0000 03 C3          ADD AX,BX
0002 03 C1          ADD AX,CX
0004 03 C2          ADD AX,DX
```

Whenever arithmetic and logic instructions execute, the contents of the flag register change. Note that the contents of the interrupt, trap, and other flags do not change due to arithmetic and logic instructions. Only the flags located in the rightmost 8 bits of the flag register and the overflow flag change. These rightmost flags denote the result of the arithmetic or a logic operation. Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags. The flag bits never change for most of the data transfer instructions presented in Chapter 4.

Immediate Addition. Immediate addition is employed whenever constant or known data are added. An 8-bit immediate addition appears in Example 5-2. In this example, DL is first loaded with 12H by using an immediate move instruction. Next, 33H is added to the 12H in DL by an immediate addition instruction. After the addition, the sum (45H) moves into register DL and the flags change, as follows:

Z = 0 (result not zero)
C = 0 (no carry)
O = 0 (no overflow)
P = 0 (odd parity)
S = 0 (result positive)
A = 0 (no half-carry)

EXAMPLE 5–2

```
0000 B2 12          MOV DL,12H
0002 80 C2 33       ADD DL,33H
```

Memory-to-Register Addition. Suppose that an application requires memory data to be added to the AL register. Example 5–3 shows an example that adds two consecutive bytes of data, stored at the data segment offset locations NUMB and NUMB+1 , to the AL register.

EXAMPLE 5–3

```
0000 BF 0000 R      MOV DI,OFFSET NUMB      ;address NUMB
0003 B0 00          MOV AL,0                ;clear sum
0005 02 05          ADD AL,[DI]             ;add NUMB
0007 02 45 01       ADD AL,[DI+1]           ;add NUMB+1
```

The first instruction loads the destination index register (DI) with offset address NUMB. The DI register, used in this example, addresses data in the data segment beginning at memory location NUMB. After clearing the sum to zero, the ADD AL,[DI] instruction adds the contents of memory location NUMB to AL. Finally, the ADD AL,[DI+1] instruction adds the contents of memory location NUMB plus 1 byte to the AL register. After both ADD instructions execute, the result appears in the AL register as the sum of the contents of NUMB plus the contents of NUMB+1 .

Array Addition. Memory arrays are sequential lists of data. Suppose that an array of data (ARRAY) contains 10 bytes, numbered from element 0 through element 9. Example 5–4 shows how to add the contents of array elements 3, 5, and 7 together.

This example first clears AL to 0, so it can be used to accumulate the sum. Next, register SI is loaded with a 3 to initially address array element 3. The ADD AL,ARRAY[SI] instruction adds the contents of array element 3 to the sum in AL. The instructions that follow add array elements 5 and 7 to the sum in AL, using a 3 in SI plus a displacement of 2 to address element 5, and a displacement of 4 to address element 7.

EXAMPLE 5–4

```
0000 B0 00              MOV AL,0            ;clear sum
0002 BE 0003            MOV SI,3            ;address element 3
0005 02 84 0000 R       ADD AL,ARRAY[SI]    ;add element 3
0009 02 84 0002 R       ADD AL,ARRAY[SI+2]  ;add element 5
000D 02 84 0004 R       ADD AL,ARRAY[SI+4]  ;add element 7
```

Suppose that an array of data contains 16-bit numbers used to form a 16-bit sum in register AX. Example 5–5 shows a sequence of instructions written for the 80386 and above, showing the scaled-index form of addressing to add elements 3, 5, and 7 of an area of memory called ARRAY. In this example, EBX is loaded with the address ARRAY, and ECX holds the array element number. Note how the scaling factor is used to multiply the contents of the ECX register by 2 to address words of data. (Recall that words are 2 bytes long.)

EXAMPLE 5–5

```
0000 66|BB 00000000 R     MOV EBX,OFFSET ARRAY        ;address ARRAY
0006 66|B9 00000003       MOV ECX,3                   ;address element 3
000C 67&8B 04 4B          MOV AX,[EBX+2*ECX]          ;get element 3
0010 66|B9 00000005       MOV ECX,5                   ;address element 5
0016 67&03 04 4B          ADD AX,[EBX+2*ECX]          ;add element 5
001A 66|B0 00000007       MOV ECX,7                   ;address element 7
0020 67&03 04 4B          ADD AX,[EBX+2*ECX]          ;add element 7
```

Increment Addition. Increment addition (INC) adds 1 to a register or a memory location. The INC instruction adds 1 to any register or memory location, except a segment register. Table 5–2 illustrates some of the possible forms of the increment instructions available to the 8086–Core2 processors. As with other instructions presented thus far, it is impossible to show all variations of the INC instruction because of the large number available.

With indirect memory increments, the size of the data must be described by using the BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directives. The reason is that the assembler program cannot determine if, for example, the INC [DI] instruction is a byte-, word-, or doubleword-sized increment. The INC BYTE PTR [DI] instruction clearly indicates bytesized memory data; the INC WORD PTR [DI] instruction unquestionably indicates a word-sized memory data; and the INC DWORD PTR [DI] instruction indicates doubleword-sized data. In 64-bit mode operation of the Pentium 4 and Core2, the INC QWORD PTR [RSI] instruction indicates quadword-sized data.

TABLE 5–2 Example increment instructions.

Assembly Language	*Operation*
INC BL	BL = BL + 1
INC SP	SP = SP + 1
INC EAX	EAX = EAX + 1
INC BYTE PTR[BX]	Adds 1 to the byte contents of the data segment memory location addressed by BX
INC WORD PTR[SI]	Adds 1 to the word contents of the data segment memory location addressed by SI
INC DWORD PTR[ECX]	Adds 1 to the doubleword contents of the data segment memory location addressed by ECX
INC DATA1	Adds 1 to the contents of data segment memory location DATA1
INC RCX	Adds 1 to RCX (64-bit mode)

Example 5–6 shows how to modify Example 5–3 to use the increment instruction for addressing NUMB and NUMB+1. Here, an INC DI instruction changes the contents of register DI from offset address NUMB to offset address NUMB+1 . Both program sequences shown in Examples 5–3 and 5–6 add the contents of NUMB and NUMB+1. The difference between them is the way that the address is formed through the contents of the DI register using the increment instruction.

EXAMPLE 5–6

```
0000 BF 0000 R      MOV DI,OFFSET NUMB          ;address NUMB
0003 B0 00          MOV AL,0                    ;clear sum
0005 02 05          ADD AL,[DI]                 ;add NUMB
0007 47             INC DI                      ;increment DI
0008 02 05          ADD AL,[DI]                 ;add NUMB+1
```

Increment instructions affect the flag bits, as do most other arithmetic and logic operations. The difference is that increment instructions do not affect the carry flag bit. Carry doesn't change because we often use

increments in programs that depend upon the contents of the carry flag. Note that increment is used to point to the next memory element in a byte-sized array of data only. If word-sized data are addressed, it is better to use an ADD DI,2 instruction to modify the DI pointer in place of two INC DI instructions. For doubleword arrays, use the ADD DI,4 instruction to modify the DI pointer. In some cases, the carry flag must be preserved, which may mean that two or four INC instructions might appear in a program to modify a pointer.

Addition-with-Carry. An addition-with-carry instruction (ADC) adds the bit in the carry flag (C) to the operand data. This instruction mainly appears in software that adds numbers that are wider than 16 bits in the 8086–80286 or wider than 32 bits in the 80386–Core2.

Table 5–3 lists several add-with-carry instructions, with comments that explain their operation. Like the ADD instruction, ADC affects the flags after the addition.

TABLE 5–3 Example add-with-carry instructions.

Assembly Language	*Operation*
ADC AL,AH	AL = AL + AH + carry
ADC CX,BX	CX = CX + BX + carry
ADC EBX,EDX	EBX = EBX + EDX + carry
ADC RBX,0	RBX = RBX + 0 + carry (64-bit mode)
ADC DH,[BX]	The byte contents of the data segment memory location addressed by BX add to DH with the sum stored in DH
ADC BX,[BP+2]	The word contents of the stack segment memory location addressed by BP plus 2 add to BX with the sum stored in BX
ADC ECX,[EBX]	The doubleword contents of the data segment memory location addressed by EBX add to ECX with the sum stored in ECX

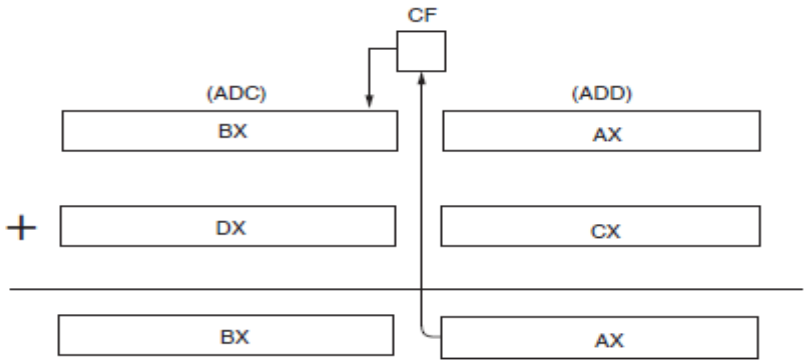


FIGURE 5–1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

Suppose that a program is written for the 8086–80286 to add the 32-bit number in BX and AX to the 32-bit number in DX and CX. Figure 5–1 illustrates this addition so that the placement and function of the carry flag can be understood. This addition cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers. Example 5–7 shows how the contents of registers AX and CX add to form the least significant 16 bits of the sum. This addition may or may not generate a carry. A carry appears in the carry flag if the sum is greater than FFFFH. Because it is impossible to predict a carry, the most significant 16 bits of this addition are added with the carry flag using the ADC instruction. The ADC instruction adds the 1 or the 0 in the carry flag to the most significant 16 bits of the result. This program adds BX–AX to DX–CX, with the sum appearing in BX–AX.

EXAMPLE 5–7

```
0000 03 C1          ADD AX,CX
0002 13 DA          ADC BX,DX
```

Suppose the same software is rewritten for the 80386 through the Core2, but modified to add two 64-bit numbers in the 32-bit mode. The changes required for this operation are the use of the extended registers to hold the data and modifications of the instructions for the 80386 and above. These changes are shown in Example 5–8, which adds two 64-bit numbers. In the 64-bit mode of the Pentium 4 and Core2, this addition

is handled with a single ADD instruction if the location of the operands is changed to RAX and RBX as in the instruction ADD RAX,RBX, which adds RBX to RAX.

EXAMPLE 5–8

```
0000 66|03 C1       ADD EAX,ECX
0003 66|13 DA       ADC EBX,EDX
```

Exchange and Add for the 80486–Core2 Processors. A new type of addition called exchange and add (XADD) appears in the 80486 instruction set and continues through the Core2. The XADD instruction adds the source to the destination and stores the sum in the destination, as with any addition. The difference is that after the addition takes place, the original value of the destination is copied into the source operand. This is one of the few instructions that change the source. For example, if BL = 12H and DL = 02H, and the XADD BL,DL instruction executes, the BL register contains the sum of 14H and DL becomes 12H. The sum of 14H is generated and the original destination of 12H replaces the source. This instruction functions with any register size and any memory operand, just as with the ADD instruction.

Subtraction

Many forms of subtraction (SUB) appear in the instruction set. These forms use any addressing mode with 8-, 16-, or 32-bit data. A special form of subtraction (decrement, or DEC) subtracts 1 from any register or memory location. Section 5–3 shows how BCD and ASCII data subtract. As with addition, numbers that are wider than 16 bits or 32 bits must occasionally be subtracted. The **subtract-with-borrow instruction** (SBB) performs this type of subtraction. In the 80486 through the Core2 processors, the instruction set also includes a compare and exchange instruction. In the 64-bit mode for the Pentium 4 and Core2, a 64-bit subtraction is also available. Table 5–4 lists some of the many addressing modes allowed with the subtract instruction (SUB). There are well over 1000 possible subtraction instructions, far too many to list here.

About the only types of subtraction not allowed are memory-to-memory and segment register subtractions. Like other arithmetic instructions, the subtract instruction affects the flag bits.

Register Subtraction. Example 5–9 shows a sequence of instructions that perform register subtraction. This example subtracts the 16-bit contents of registers CX and DX from the contents of register BX. After each subtraction, the microprocessor modifies the contents of the flag register. The flags change for most arithmetic and logic operations.

EXAMPLE 5–9

```
0000 2B D9          SUB BX,CX
0002 2B DA          SUB BX,DX
```

Immediate Subtraction. As with addition, the microprocessor also allows immediate operands for the subtraction of constant data. Example 5–10 presents a short sequence of instructions that subtract 44H from 22H. Here, we first load the 22H into CH using an immediate move instruction. Next, the SUB instruction, using immediate data 44H, subtracts 44H from the 22H.

TABLE 5–4 Example subtraction instructions.

Assembly Language	*Operation*
SUB CL,BL	CL = CL – BL
SUB AX,SP	AX = AX – SP
SUB ECX,EBP	ECX = ECX – EBP
SUB RDX,R8	RDX = RDX – R8 (64-bit mode)
SUB DH,6FH	DH = DH – 6FH
SUB AX,0CCCCH	AX = AX – 0CCCCH
SUB ESI,2000300H	ESI = ESI – 2000300H
SUB [DI],CH	Subtracts CH from the byte contents of the data segment memory addressed by DI and stores the difference in the same memory location
SUB CH,[BP]	Subtracts the byte contents of the stack segment memory location addressed by BP from CH and stores the difference in CH
SUB AH,TEMP	Subtracts the byte contents of memory location TEMP from AH and stores the difference in AH
SUB DI,TEMP[ESI]	Subtracts the word contents of the data segment memory location addressed by TEMP plus ESI from DI and stores the difference in DI
SUB ECX,DATA1	Subtracts the doubleword contents of memory location DATA1 from ECX and stores the difference in ECX
SUB RCX,16	RCX = RCX – 18 (64-bit mode)

After the subtraction, the difference (0DEH) moves into the CH register. The flags change as follows for this subtraction:

Z = 0 (result not zero)
O = 0 (no overflow)
P = 1 (even parity)
S = 1 (result negative)
A = 1 (half-borrow)
C = 1 (borrow)

EXAMPLE 5–10

```
0000 B5 22          MOV CH,22H
0002 80 ED 44       SUB CH,44H
```

Both carry flags (C and A) hold borrows after a subtraction instead of carries, as after an addition. Notice in this example that there is no overflow. This example subtracted 44H (+68) from 22H (+34), resulting in a 0DEH (-34). Because the correct 8-bit signed result is -34, there is no overflow in this example. An 8-bit overflow occurs only if the signed result is greater than +127 or less than -128.

Decrement Subtraction. Decrement subtraction (DEC) subtracts 1 from a register or the contents of a memory location. Table 5–5 lists some decrement instructions that illustrate register and memory decrements.

TABLE 5–5 Example decrement instructions.

Assembly Language	*Operation*
DEC BH	BH = BH – 1
DEC CX	CX = CX – 1
DEC EDX	EDX = EDX – 1
DEC R14	R14 = R14 – 1 (64-bit mode)
DEC BYTE PTR[DI]	Subtracts 1 from the byte contents of the data segment memory location addressed by DI
DEC WORD PTR[BP]	Subtracts 1 from the word contents of the stack segment memory location addressed by BP
DEC DWORD PTR[EBX]	Subtracts 1 from the doubleword contents of the data segment memory location addressed by EBX
DEC QWORD PTR[RSI]	Subtracts 1 from the quadword contents of the memory location addressed by RSI (64-bit mode)
DEC NUMB	Subtracts 1 from the contents of data segment memory location NUMB

TABLE 5–6 Example subtraction-with-borrow instructions.

Assembly Language	*Operation*
SBB AH,AL	AH = AH – AL – carry
SBB AX,BX	AX = AX – BX – carry
SBB EAX,ECX	EAX = EAX – ECX – carry
SBB CL,2	CL = CL – 2 – carry
SBB RBP,8	RBP = RBP– 2 – carry (64-bit mode)
SBB BYTE PTR[DI],3	Both 3 and carry subtract from the data segment memory location addressed by DI
SBB [DI],AL	Both AL and carry subtract from the data segment memory location addressed by DI
SBB DI,[BP+2]	Both carry and the word contents of the stack segment memory location addressed by BP plus 2 subtract from DI
SBB AL,[EBX+ECX]	Both carry and the byte contents of the data segment memory location addressed by EBX plus ECX subtract from AL

The decrement indirect memory data instructions require BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR because the assembler cannot distinguish a byte from a word or doubleword when an index register addresses memory. For example, DEC [SI] is vague because the assembler cannot determine whether the location addressed by SI is a byte, word, or doubleword. Using DEC BYTE PTR[SI], DEC WORD PTR[DI], or DEC DWORD PTR[SI] reveals the size of the data to the assembler. In the 64-bit mode, a DEC QWORD PTR[RSI] decrement the 64-bit number stored at the address pointed to by the RSI register.

Subtraction-with-Borrow. A subtraction-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the difference. The most common use for this instruction is for subtractions that are wider than 16 bits in the 8086–80286 microprocessors or wider than 32 bits in the 80386–Core2. Wide subtractions require that borrows propagate through the subtraction, just as wide additions propagate the carry. Table 5–6 lists several SBB instructions with comments that define their operations. Like the SUB instruction, SBB affects the flags. Notice that the immediate subtract from memory instruction in this table requires a BYTE PTR, WORD PTR, DWORD PTR, or QWORD PTR directive.

When the 32-bit number held in BX and AX is subtracted from the 32-bit number held in SI and DI, the carry flag propagates the borrow between the two 16-bit subtractions. The carry flag holds the borrow for subtraction. Figure 5–2 shows how the borrow propagates through the carry flag (C) for this task. Example 5–11 shows how this subtraction is performed by a program. With wide subtraction, the least significant 16- or 32-bit data are subtracted with the SUB instruction. All subsequent and more significant data are subtracted by using the SBB instruction.

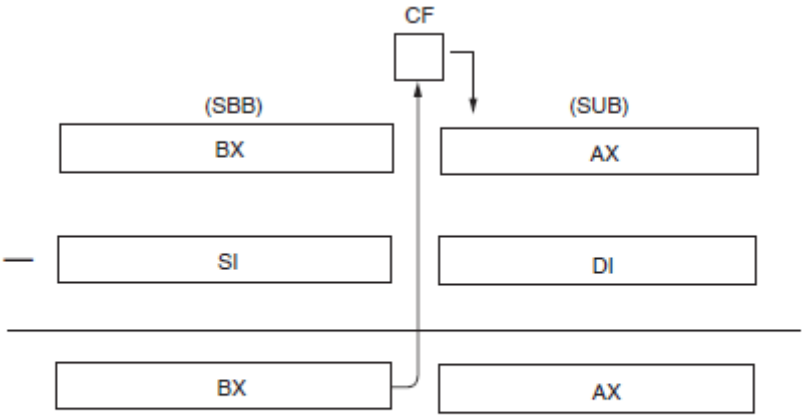


FIGURE 5–2 Subtraction- with-borrow showing how the carry flag (C) propagates the borrow.

The example uses the SUB instruction to subtract DI from AX, then uses SBB to subtract with-borrow SI from BX.

EXAMPLE 5–11

```
0000 2B C7          SUB AX,DI
0002 1B DE          SBB BX,SI
```

Comparison

The comparison instruction (CMP) is a subtraction that changes only the flag bits; the destination operand never changes. A comparison is useful for checking the entire contents of a register or a memory location against another value. A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

Table 5–7 lists a variety of comparison instructions that use the same addressing modes as the addition and subtraction instructions already presented. Similarly, the only disallowed forms of compare are memory-to-memory and segment register compares.

Example 5–12 shows a comparison followed by a conditional jump instruction. In this example, the contents of AL are compared with 10H. Conditional jump instructions that often follow the comparison are JA (jump above) or JB (jump below). If the JA follows the comparison, the jump occurs if the value in AL is above 10H. If the JB follows the comparison, the jump occurs if the value in AL is below 10H. In this example, the JAE instruction follows the comparison. This instruction causes the program to continue at memory location SUBER if the value in AL is 10H or above. There is also a JBE (jump below or equal) instruction that could follow the comparison to jump if the outcome is below or equal to 10H. Later chapters provide additional detail on the comparison and conditional jump instructions.

TABLE 5–7 Example comparison instructions.

Assembly Language	*Operation*
CMP CL,BL	CL – BL
CMP AX,SP	AX – SP
CMP EBP,ESI	EBP – ESI
CMP RDI,RSI	RDI – RSI (64-bit mode)
CMP AX,2000H	AX – 2000H
CMP R10W,12H	R10 (word portion) – 12H (64-bit mode)
CMP [DI],CH	CH subtracts from the byte contents of the data segment memory location addressed by DI
CMP CL,[BP]	The byte contents of the stack segment memory location addressed by BP subtracts from CL
CMP AH,TEMP	The byte contents of data segment memory location TEMP subtracts from AH
CMP DI,TEMP[BX]	The word contents of the data segment memory location addressed by TEMP plus BX subtracts from DI
CMP AL,[EDI+ESI]	The byte contents of the data segment memory location addressed by EDI plus ESI subtracts from AL

EXAMPLE 5–12

```
0000 3C 10          CMP AL,10H          ;compare AL against 10H
0002 73 1C          JAE SUBER           ;if AL is 10H or above
```

Compare and Exchange (80486–Core2 Processors Only). The compare and exchange instruction (CMPXCHG), found only in the 80486 through the Core2 instruction sets, compares the destination operand with the accumulator. If they are equal, the source operand is copied into the destination; if they are not equal, the destination operand is copied into the accumulator. This instruction functions with 8-, 16-, or 32-bit data.

The CMPXCHG CX,DX instruction is an example of the compare and exchange instruction. This instruction first compares the contents of CX with AX. If CX equals AX, DX is copied into AX; if CX is not equal to AX, CX is copied into AX. This instruction also compares AL with 8-bit data and EAX with 32-bit data if the operands are either 8- or 32-bit. In the Pentium–Core2 processors, a CMPXCHG8B instruction is available that compares two quadwords. This is the only new data manipulation instruction provided in the Pentium–Core2 when they are compared with prior versions of the microprocessor. The compare-and-exchange- 8-bytes instruction compares the 64-bit value located in EDX:EAX with a 64-bit number located in memory. An example is CMPXCHG8B TEMP. If TEMP equals EDX:EAX, TEMP is replaced with the value found in ECX:EBX; if TEMP does not equal EDX:EAX, the number found in TEMP is loaded into EDX:EAX. The Z (zero) flag bit indicates that the values are equal after the comparison.
This instruction has a bug that will cause the operating system to crash. More information about this flaw can be obtained at www.intel.com. There is also a CMPXCHG16B instruction available to the Pentium 4 when operated in 64-bit mode.

MULTIPLICATION AND DIVISION

Only modern microprocessors contain multiplication and division instructions. Earlier 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided by using a series of shifts and additions or subtractions. Because microprocessor manufacturers were aware of this inadequacy, they incorporated multiplication and division instructions into the instruction sets of the newer microprocessors. The Pentium–Core2 processors contain special circuitry that performs a multiplication in as little as one clocking period, whereas it took over 40 clocking periods to perform the same multiplication in earlier Intel microprocessors.

Multiplication

Multiplication is performed on bytes, words, or doublewords, and can be signed integer (IMUL) or unsigned integer (MUL). Note that only the 80386 through the Core2 processors multiply 32-bit doublewords. The product after a multiplication is always a double-width product. If two 8-bit numbers are multiplied, they generate a 16-bit product; if two 16-bit numbers are multiplied, they generate a 32-bit product; and if two 32-bit numbers are multiplied, a 64-bit product is generated. In the 64-bit mode of the Pentium 4, two 64-bit numbers are multiplied to generate a 128-bit product.
Some flag bits (overflow and carry) cha nge when the multiply instruction executes and produce predictable outcomes. The other flags also change, but their results are unpredictable and therefore are unused. In an 8-bit multiplication, if the most significant 8 bits of the result are zero, both C and O flag bits equal zero. These flag bits show that the result is 8 bits wide (C = 0) or 16 bits wide (C = 1). In a 16-bit multiplication, if the most significant 16-bits part of the product is 0, both C and O clear to zero. In a 32-bit multiplication, both C and O indicate that the most significant 32 bits of the product are zero.

TABLE 5–8 Example 8-bit multiplication instructions.

Assembly Language	*Operation*
MUL CL	AL is multiplied by CL; the unsigned product is in AX
IMUL DH	AL is multiplied by DH; the signed product is in AX
IMUL BYTE PTR[BX]	AL is multiplied by the byte contents of the data segment memory location addressed by BX; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

8-Bit Multiplication. With 8-bit multiplication, the multiplicand is always in the AL register, whether signed or unsigned. The multiplier can be any 8-bit register or any memory location. Immediate multiplication is not allowed unless the special signed immediate multiplication instruction, discussed later in this section, appears in a program. The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL. An example is the MUL BL instruction, which multiplies the unsigned contents of AL by the unsigned contents of BL. After the multiplication, the unsigned product is placed in AX—a double-width product. Table 5–8 illustrates some 8-bit multiplication instructions.

Suppose that BL and CL each contain two 8-bit unsigned numbers, and these numbers must be multiplied to form a 16-bit product stored in DX. This procedure cannot be accomplished by a single instruction because we can only multiply a number times the AL register for an 8-bit multiplication. Example 5–13 shows a short program that generates DX = BL * CL.

This example loads register BL and CL with example data 5 and 10. The product, a 50, moves into DX from AX after the multiplication by using the MOV DX,AX instruction.

EXAMPLE 5–13

```
0000 B3 05        MOV  BL,5          ;load data
0002 B1 0A        MOV  CL,10
0004 8A C1        MOV  AL,CL         ;position data
0006 F6 E3        MUL  BL            ;multiply
0008 8B D0        MOV  DX,AX         ;position product
```

For signed multiplication, the product is in binary form, if positive, and in two's complement form, if negative. These are the same forms used to store all positive and negative signed numbers used by the microprocessor. If the program of Example 5–13 multiplies two signed numbers, only the MUL instruction is changed to IMUL.

16-Bit Multiplication. Word multiplication is very similar to byte multiplication. The difference is that AX contains the multiplicand instead of AL, and the 32-bit product appears in DX–AX instead of AX. The DX register always contains the most significant 16 bits of the product, and AX contains the least significant 16 bits. As with 8-bit multiplication, the choice of the multiplier is up to the programmer. Table 5–9 shows several different 16-bit multiplication instructions.

A Special Immediate 16-Bit Multiplication. The 8086/8088 microprocessors could not perform immediate multiplication; the 80186 through the Core2 processors can do so by using a special version of the multiply instruction. Immediate multiplication must be signed multiplication, and the instruction format is different because it contains three operands. The first operand is the 16-bit destination register; the second operand is a register or memory location that contains the 16-bit multiplicand; and the third operand is either 8-bit or 16-bit immediate data used as the multiplier.

TABLE 5–9 Example 16-bit multiplication instructions.

Assembly Language	*Operation*
MUL CX	AX is multiplied by CX; the unsigned product is in DX–AX
IMUL DI	AX is multiplied by DI; the signed product is in DX–AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX–AX

The IMUL CX,DX,12H instruction multiplies 12H times DX and leaves a 16-bit signed product in CX. If the immediate data are 8 bits, they sign-extend into a 16-bit number before the multiplication occurs. Another example is IMUL BX,NUMBER,1000H, which multiplies NUMBER times 1000H and leaves the product in BX. Both the destination and multiplicand must be 16-bit numbers. Although this is immediate multiplication, the restrictions placed upon it limit its utility, especially the fact that it is a signed multiplication and the product is 16 bits wide.

32-Bit Multiplication. In the 80386 and above, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers. As with 8- and 16-bit multiplication, 32-bit multiplication can be signed or unsigned by using the IMUL and MUL instructions. With 32-bit multiplication, the contents of EAX are multiplied by the operand specified with the instruction. The product (64 bits wide) is found in EDX–EAX, where EAX contains the least significant 32 bits of the product. Table 5–10 lists some of the 32-bit multiplication instructions found in the 80386 and above instruction set.

64-Bit Multiplication. The result of a 64-bit multiplication in the Pentium 4 appears in the RDX:RAX register pair as a 128-bit product. Although multiplication of this size is relatively rare, the Pentium 4 and Core2 can perform it on both signed and unsigned numbers. Table 5–11 shows a few examples of this high precision multiplication.

TABLE 5–10 Example 32-bit multiplication instructions.

Assembly Language	*Operation*
MUL ECX	EAX is multiplied by ECX; the unsigned product is in EDX–EAX
IMUL EDI	EAX is multiplied by EDI; the signed product is in EDX–EAX
MUL DWORD PTR[ESI]	EAX is multiplied by the doubleword contents of the data segment memory location address by ESI; the unsigned product is in EDX–EAX

TABLE 5–11 Example 64-bit multiplication instructions.

Assembly Language	*Operation*
MUL RCX	RAX is multiplied by RCX; the unsigned product is in RDX–RAX
IMUL RDI	RAX is multiplied by RDI; the signed product is in RDX–RAX
MUL QWORD PTR[RSI]	RAX is multiplied by the quadword contents of the memory location address by RSI; the unsigned product is in RDX–RAX

Division

As with multiplication, division occurs on 8- or 16-bit numbers in the 8086–80286 microprocessors, and on 32-bit numbers in the 80386 and above microprocessor. These numbers are signed (IDIV) or unsigned (DIV) integers. The dividend is always a double-width dividend that is divided by the operand. This means that an 8-bit division divides a 16-bit number by an 8-bit number; a 16-bit division divides a 32-bit number by a 16-bit number; and a 32-bit division divides a 64-bit number by a 32-bit number. There is no immediate division instruction available to any microprocessor. In the 64-bit mode of the Pentium 4 and Core2, a 64-bit division divides a 128-bit number by a 64-bit number.

None of the flag bits change predictably for a division. A division can result in two different types of errors; one is an attempt to divide by zero and the other is a divide overflow. A divide overflow occurs when a small number divides into a large number. For example, suppose that AX = 3000 and that it is divided by 2. Because the quotient for an 8-bit division appears in AL, the result of 1500 causes a divide overflow because the 1500 does not fit into AL. In either case, the microprocessor generates an interrupt if a divide error occurs. In most systems, a divide error interrupt displays an error message on the video screen. The divide error interrupt and all other interrupts for the microprocessor are explained in Chapter 6.

8-Bit Division. An 8-bit division uses the AX register to store the dividend that is divided by the contents of any 8-bit register or memory location. The quotient moves into AL after the division with AH containing a whole number remainder. For a signed division, the quotient is positive or negative; the remainder always assumes the sign of the dividend and is always an integer. For example, if AX = 0010H (+16) and BL = 0FDH (-3) and the IDIV BL instruction executes, AX = 01FBH. This represents a quotient of -5 (AL) with a remainder of 1 (AH). If, on the other hand, a -16 is divided by +3, the result will be a quotient of -5 (AL) with a remainder of -1 (AH). Table 5–12 lists some of the 8-bit division instructions.

With 8-bit division, the numbers are usually 8 bits wide. This means that one of them, the dividend, must be converted to a 16-bit wide number in AX. This is accomplished differently for signed and unsigned numbers. For the unsigned number, the most significant 8 bits must be cleared to zero (zero-extended). The MOVZX instruction described in Chapter 4 can be used to zero-extend a number in the 80386 through the Core2 processors. For signed numbers, the least significant 8 bits are sign-extended into the most significant 8 bits. In the microprocessor, a special instruction sign-extends AL into AH, or converts an 8-bit signed number in AL into a 16-bit signed number in AX. The CBW (**convert byte to word**) instruction performs this conversion. In the 80386 through the Core2, a MOVSX instruction (see Chapter 4) sign-extends a number.

TABLE 5–12 Example 8-bit division instructions.

Assembly Language	*Operation*
DIV CL	AX is divided by CL; the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL; the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP; the unsigned quotient is in AL and the unsigned remainder is in AH

Example 5–14 illustrates a short program that divides the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1. Here, the quotient is stored in location ANSQ and the remainder is stored in location ANSR. Notice how the contents of location NUMB are retrieved from memory and then zero-extended to form a 16-bit unsigned number for the dividend.

EXAMPLE 5–14

```
0000 A0 0000 R        MOV  AL,NUMB      ;get NUMB
0003 B4 00            MOV  AH,0         ;zero-extend
0005 F6 36 0002 R     DIV  NUMB1        ;divide by NUMB1
0009 A2 0003 R        MOV  ANSQ,AL      ;save quotient
000C 88 26 0004 R     MOV  ANSR,AH      ;save remainder
```

EXAMPLE 5–15

```
0000 B8 FF9C          MOV  AX,-100      ;load a -100
0003 B9 0009          MOV  CX,9         ;load +9
0006 99               CWD               ;sign-extend
0007 F7 F9            IDIV  CX
```

16-Bit Division. Sixteen-bit division is similar to 8-bit division, except that instead of dividing into AX, the 16-bit number is divided into DX–AX, a 32-bit dividend. The quotient appears in AX and the remainder appears in DX after a 16-bit division. Table 5–13 lists some of the 16-bit division instructions.

As with 8-bit division, numbers must often be converted to the proper form for the dividend. If a 16-bit unsigned number is placed in AX, DX must be cleared to zero. In the 80386 and above, the number is zero-extended by using the MOVZX instruction. If AX is a 16-bit signed number, the CWD (**convert word to doubleword**) instruction sign-extends it into a signed 32-bit number. If the 80386 and above is available, the MOVSX instruction can also be used to sign-extend a number. Example 5–15 shows the division of two 16-bit signed numbers. Here, -100 in AX is divided +9 by in CX. The CWD instruction converts the -100 in AX to -100 in DX–AX before the division. After the division, the results appear in DX–AX as a quotient of -11 in AX and aremainder of -1 in DX.

32-Bit Division. The 80386 through the Pentium 4 processors perform 32-bit division on signed or unsigned numbers. The 64-bit contents of EDX–EAX are divided by the operand specified by the instruction, leaving a 32-bit quotient in EAX and a 32-bit remainder in EDX. Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions. Table 5–14 shows some 32-bit division instructions. The CDQ (**convert doubleword to quadword**) instruction is used before a signed division to convert the 32-bit contents of EAX into a 64-bit signed number in EDX–EAX.

TABLE 5–13 Example 16-bit division instructions.

Assembly Language	*Operation*
DIV CX	DX–AX is divided by CX; the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX–AX is divided by SI; the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX–AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX

TABLE 5–14 Example 32-bit division instructions.

Assembly Language	*Operation*
DIV ECX	EDX–EAX is divided by ECX; the unsigned quotient is in EAX and the unsigned remainder is in EDX
IDIV DATA4	EDX–EAX is divided by the doubleword contents in data segment memory location DATA4; the signed quotient is in EAX and the signed remainder is in EDX
DIV DWORD PTR[EDI]	EDX–EAX is divided by the doubleword contents of the data segment memory location addressed by EDI; the unsigned quotient is in EAX and the unsigned remainder is in EDX

The Remainder. What is done with the remainder after a division? There are a few possible choices. The remainder could be used to round the quotient or just dropped to truncate the quotient. If the division is unsigned, rounding requires that the remainder be compared with half the divisor to decide whether to round up the quotient. The remainder could also be converted to a fractional remainder.

EXAMPLE 5–16

```
0000 F6 F3          DIV  BL       ;divide
0002 02 E4          ADD  AH,AH    ;double remainder
0004 3A E3          CMP  AH,BL    ;test for rounding
0006 72 02          JB   NEXT     ;if OK
0008 FE C0          INC  AL       ;round
000A          NEXT:
```

EXAMPLE 5–17

```
0000 B8 000D        MOV  AX,13    ;load 13
0003 B3 02          MOV  BL,2     ;load 2
0005 F6 F3          DIV  BL       ;13/2
0007 A2 0003 R      MOV  ANSQ,AL  ;save quotient
000A B0 00          MOV  AL,0     ;clear AL
000C F6 F3          DIV  BL       ;generate remainder
000E A2 0004 R      MOV  ANSR,AL  ;save remainder
```

Example 5–16 shows a sequence of instructions that divide AX by BL and round the unsigned result. This program doubles the remainder before comparing it with BL to decide whether to round the quotient. Here, an INC instruction rounds the contents of AL after the comparison. Suppose that a fractional remainder is required instead of an integer remainder. A fractional remainder is obtained by saving the quotient. Next, the AL register is cleared to zero. The number remaining in AX is now divided by the original operand to generate a fractional remainder. Example 5–17 shows how 13 is divided by 2. The 8-bit quotient is saved in memory location ANSQ, and then AL is cleared. Next, the contents of AX are again divided by 2 to generate a fractional remainder. After the division, the AL register equals 80H. This is 100000002. If the binary point (radix) is placed before the leftmost bit of AL, the fractional remainder in AL is 0.100000002 or 0.5 decimal. The remainder is saved in memory location ANSR in this example.

64-Bit Division. The Pentium 4 processor operated in 64-bit mode performs 64-bit division on signed or unsigned numbers. The 64-bit division uses the RDX:RAX register pair to hold the dividend and the quotient is found in RAX and the remainder is in RDX after the division. Table 5–15 illustrates a few 64-bit division instructions.

TABLE 5–15 Example 64-bit division instructions.

Assembly Language	*Operation*
DIV RCX	RDX–RAX is divided by RCX; the unsigned quotient is in RAX and the unsigned remainder is in RDX
IDIV DATA4	RDX–RAX is divided by the quadword contents in memory location DATA4; the signed quotient is in RAX and the signed remainder is in RDX
DIV QWORD PTR[RDI]	RDX–RAX is divided by the quadword contents of the memory location addressed by RDI; the unsigned quotient is in RAX and the unsigned remainder is in RDX

BCD AND ASCII ARITHMETIC

The microprocessor allows arithmetic manipulation of both BCD (**binary-coded decimal**) and ASCII (**American Standard Code for Information Interchange**) data. This is accomplished by instructions that adjust the numbers for BCD and ASCII arithmetic.

The BCD operations occur in systems such as point-of-sales terminals (e.g., cash registers) and others that seldom require complex arithmetic. The ASCII operations are performed on ASCII data used by many programs. In many cases, BCD or ASCII arithmetic is rarely used today, but some of the operations can be used for other purposes.

None of the instructions detailed in this section of the chapter function in the 64-bit mode of the Pentium 4 or Core2. In the future it appears that the BCD and ASCII instruction will become obsolete.

BCD Arithmetic

Two arithmetic techniques operate with BCD data: addition and subtraction. The instruction set provides two instructions that correct the result of a BCD addition and a BCD subtraction. The DAA (**decimal adjust after addition**) instruction follows BCD addition, and the DAS (**decimal adjust after subtraction**) follows BCD subtraction. Both instructions correct the result of the addition or subtraction so that it is a BCD number. For BCD data, the numbers always appear in the packed BCD form and are stored as two BCD digits per byte. The adjustment instructions function only with the AL register after BCD addition and subtraction.

DAA Instruction. The DAA instruction follows the ADD or ADC instruction to adjust the result into a BCD result. Suppose that DX and BX each contain 4-digit packed BCD numbers.

Example 5–18 provides a short sample program that adds the BCD numbers in DX and BX, and stores the result in CX. Because the DAA instruction functions only with the AL register, this addition must occur 8 bits at a time. After adding the BL and DL registers, the result is adjusted with a DAA instruction before being stored in CL. Next, add BH and DH registers with carry; the result is then adjusted with DAA before being stored in CH. In this example, a 1234 is added to 3099 to generate a sum of 4333, which moves into CX after the addition. Note that 1234 BCD is the same as 1234H.

EXAMPLE 5–18

```
0000 BA 1234    MOV  DX,1234H   ;load 1234 BCD
0003 BB 3099    MOV  BX,3099H   ;load 3099 BCD
0006 8A C3      MOV  AL,BL      ;sum BL and DL
0008 02 C2      ADD  AL,DL
000A 27         DAA
000B 8A C8      MOV  CL,AL      ;answer to CL
000D 9A C7      MOV  AL,BH      ;sum BH, DH and carry
000F 12 C6      ADC  AL,DH
0011 27         DAA
0012 8A E8      MOV  CH,AL      ;answer to CH
```

EXAMPLE 5–19

```
0000 BA 1234    MOV  DX,1234H   ;load 1234 BCD
0003 BB 3099    MOV  BX,3099H   ;load 3099 BCD
0006 8A C3      MOV  AL,BL      ;subtract DL from BL
0008 2A C2      SUB  AL,DL
000A 2F         DAS
000B 8A C8      MOV  CL,AL      ;answer to CL
000D 9A C7      MOV  AL,BH      ;subtract DH
000F 1A C6      SBB  AL,DH
0011 2F         DAS
0012 8A E8      MOV  CH,AL      ;answer to CH
```

DAS Instruction. The DAS instruction functions as does the DAA instruction, except that it follows a subtraction instead of an addition. Example 5-19 is the same as Example 5–18, except that it subtracts instead of adds DX and BX. The main difference in these programs is that the DAA instructions change to DAS, and the ADD and ADC instructions change to SUB and SBB instructions.

ASCII Arithmetic

The ASCII arithmetic instructions function with ASCII-coded numbers. These numbers range in value from 30H to 39H for the numbers 0–9. There are four instructions used with ASCII arithmetic operations: AAA (**ASCII adjust after addition**), AAD (**ASCII adjust before division**), AAM (**ASCII adjust after multiplication**), and AAS (**ASCII adjust after subtraction**). These instructions use register AX as the source and as the destination.

AAA Instruction. The addition of two one-digit ASCII-coded numbers will not result in any useful data. For example, if 31H and 39H are added, the result is 6AH. This ASCII addition (1+9) should produce a two-digit ASCII result equivalent to a 10 decimal, which is a 31H and a 30H in ASCII code. If the AAA instruction is executed after this addition, the AX register will contain a 0100H. Although this is not ASCII code, it can be converted to ASCII code by adding 3030H to AX which generates 3130H. The AAA instruction clears AH if the result is less than 10, and adds 1 to AH if the result is greater than 10.

Example 5–20 shows the way ASCII addition functions in the microprocessor. Please note that AH is cleared to zero before the addition by using the MOV AX,31H instruction. The operand of 0031H places 00H in AH and 31H into AL.

EXAMPLE 5–20

```
0000 B8 0031    MOV  AX,31H     ;load ASCII 1
0003 04 39      ADD  AL,39H     ;add ASCII 9
0005 37         AAA             ;adjust sum
0006 05 3030    ADD  AX,3030H   ;answer to ASCII
```

EXAMPLE 5–21

```
0000 B3 09      MOV  BL,9       ;load divisor
0002 B8 0702    MOV  AX,702H    ;load dividend
0005 D5 0A      AAD             ;adjust
0007 F6 F3      DIV  BL         ;divide
```

AAD Instruction. Unlike all other adjustment instructions, the AAD instruction appears before a division. The AAD instruction requires that the AX register contain a two-digit unpacked BCD number (not ASCII)

before executing. After adjusting the AX register with AAD, it is divided by an unpacked BCD number to generate a single-digit result in AL with any remainder in AH. Example 5–21 illustrates how 72 in unpacked BCD is divided by 9 to produce a quotient of 8. The 0702H loaded into the AX register is adjusted by the AAD instruction to 0048H. Notice that this converts a two-digit unpacked BCD number into a binary number so it can be divided with the binary division instruction (DIV). The AAD instruction converts the unpacked BCD numbers between 00 and 99 into binary.

AAM Instruction. The AAM instruction follows the multiplication instruction after multiplying two one-digit unpacked BCD numbers. Example 5–22 shows a short program that multiplies 5 times 5. The result after the multiplication is 0019H in the AX register. After adjusting the result with the AAM instruction, AX contains 0205H. This is an unpacked BCD result of 25. If 3030H is added to 0205H, it has an ASCII result of 3235H.

The AAM instruction accomplishes this conversion by dividing AX by 10. The remainder is found in AL, and the quotient is in AH. Note that the second byte of the instruction contains 0AH. If the 0AH is changed to another value, AAM divides by the new value. For example, if the second byte is changed to 0BH, the AAM instruction divides by 11. This is accomplished with DB 0D4H, 0BH in place of AAM, which forces the AMM instruction to multiply by 11. One side benefit of the AAM instruction is that AAM converts from binary to unpacked BCD. If a binary number between 0000H and 0063H appears in the AX register, the AAM instruction converts it to BCD. For example, if AX contains a 0060H before AAM, it will contain 0906H after AAM executes. This is the unpacked BCD equivalent of 96 decimal. If 3030H is added to 0906H, the result changes to ASCII code.

Example 5–23 shows how the l6-bit binary content of AX is converted to a four-digit ASCII character string by using division and the AAM instruction. Note that this works for numbers between 0 and 9999. First DX is cleared and then DX–AX is divided by 100. For example, if AX = 210_{10}, AX = 2 and DX = 45 after the division. These separate halves are converted to BCD using AAM, and then 3030H is added to convert to ASCII code.

EXAMPLE 5–23

```
0000 33 D2      XOR  DX,DX      ;clear DX
0002 B9 0064    MOV  CX,100     ;divide DX-AX by 100
0005 F7 F1      DIV  CX
0007 D4 0A      AAM             ;convert to BCD
0009 05 3030    ADD  AX,3030H   ;convert to ASCII
000C 92         XCHG AX,DX      ;repeat for remainder
000D D4 0A      AAM
000F 05 3030    ADD AX,3030H
```

EXAMPLE 5–24

```
                ;A program that displays the number in AL, loaded
                ;with the first instruction (48H).
                ;
                .MODEL TINY          ;select tiny model
0000            .CODE                ;start code segment
                .STARTUP             ;start program
0100 B0 48          MOV  AL,48H      ;load test data
0102 B4 00          MOV  AH,0        ;clear AH
0104 D4 0A          AAM              ;convert to BCD
0106 05 3030        ADD  AX,3030H    ;convert to ASCII
0109 8A D4          MOV  DL,AH       ;display most-significant digit
010B B4 02          MOV  AH,2
010D 50             PUSH AX
010E CD 21          INT  21H
0110 58             POP  AX
0111 8A D0          MOV  DL,AL       ;display least-significant digit
0113 CD 21          INT  21H
                .EXIT                ;exit to DOS
                END
```

Example 5–24 uses the DOS 21H function AH=02H to display a sample number in decimal on the video display using the AAM instruction. Notice how AAM is used to convert AL into BCD. Next, ADD AX,3030H converts the BCD code in AX into ASCII for display with DOS INT 21H. Once the data are converted to ASCII code, they are displayed by loading DL with the most significant digit from AH. Next, the least significant digit is displayed from AL. Note that the DOS INT 2lH function calls change AL.

AAS Instruction. Like other ASCII adjust instructions, AAS adjusts the AX register after an ASCII subtraction. For example, suppose that 35H subtracts from 39H. The result will be 04H, which requires no correction. Here, AAS will modify neither AH nor AL. On the other hand, if 38H is subtracted from 37H, then AL will equal 09H and the number in AH will decrement by 1. This decrement allows multiple-digit ASCII numbers to be subtracted from each other.