APPENDIX

D

INTEL IA-32 INSTRUCTION SET

This appendix contains a summary of the Intel IA-32 instruction set which was introduced in Part III of Chapter 3. This instruction set is very extensive. We only describe a small part of it, about 50 instructions, including all of those used in Chapter 3. Some general aspects of other instruction types are also covered.

The IA-32 register structure is shown in Figures 3.37 and 3.38 and described in Section 3.16.1. The general format of an instruction is shown in Figure 3.41. Memory is byte addressable and addresses are 32 bits long. There are two operand sizes: doubleword (32 bits) and byte (8 bits). Word operands (16 bits) were used in earlier 16-bit Intel processors. IA-32 processors can operate in a 16-bit mode to execute machine programs prepared for the earlier 16-bit processors.

D.1 INSTRUCTION ENCODING

Figure D.1*a* shows the general format for encoding IA-32 instructions. The OP code is either one or two bytes long. For some instructions, the OP code is extended into the 3-bit Reg/OPcode field of the ModR/M byte shown in Figure D.1*b*. Since the encoding

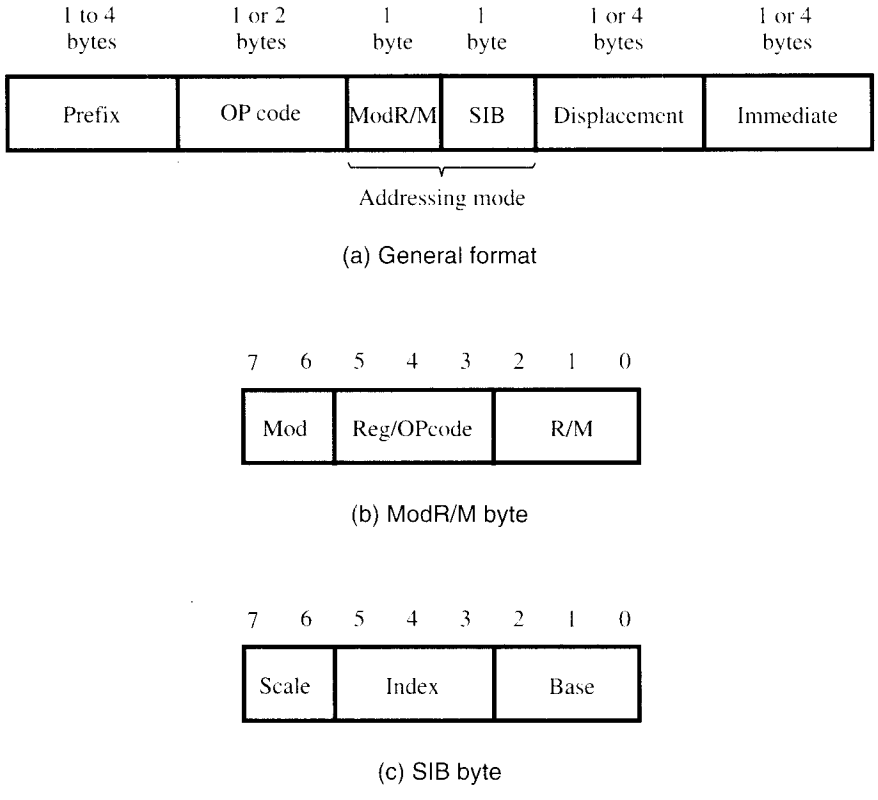

Figure D.1 IA-32 instruction format.

of the OP codes is very irregular, we will not give any details. Prefix bytes, which are not needed in encoding most of the common instructions discussed here, are discussed in Section D.3.

The length of an instruction can range from one byte (an OP code) to 11 or more bytes when both a 4-byte displacement and a 4-byte immediate operand are specified, along with two addressing mode bytes and the OP code. For example, the Increment (INC) and Decrement (DEC) instructions on a register operand require only a 1-byte OP code that includes a 3-bit field to name the register. As an example of a long instruction, 11 bytes are needed to encode the instruction

```
MOV   DWORD PTR [EBP + ESI*4 + DISP],10
```

as discussed in Section 3.17.1. Other instruction examples are also discussed in that section.

The operand data size (8 bits or 32 bits) is indicated in the OP code. The OP code also indicates whether or not one of the operands is an immediate value contained in the last field of the instruction.

At least one of the operands in a two-operand instruction must be in a register. It is named in the Reg/OPcode field of the ModR/M byte. The 3-bit codes for the registers are shown in Table D.1. If the other operand is in a register, it is named in the R/M field of the same byte. If the other operand is not in a register, it may be an immediate value or it may be the contents of a memory location. The address of a memory operand is specified by the two addressing mode bytes and the displacement field as described in the next section. The specification of which operand is the source in the encoding of a two-operand instruction is determined by a bit in the OP code, called the *direction* bit, as described in Section 3.17.1.

Table D.1 Register field encoding in IA-32 instructions

Reg/Base/Index* field	Register
0 0 0	EAX
0 0 1	ECX
0 1 0	EDX
0 1 1	EBX
1 0 0	ESP
1 0 1	EBP
1 1 0	ESI
1 1 1	EDI

*ESP (100) cannot be used as an index register.

D.1.1 ADDRESSING MODES

Table 3.3 lists the IA-32 addressing modes, the assembler syntax used to specify them, and the way that the effective address EA is generated. We have already discussed the Immediate mode and the use of the Reg/OPcode field of the ModR/M byte to specify a register as the location of one operand. The other operand is specified as shown in Table D.2. The Register indirect, Base with displacement, and Register addressing

Table D.2 IA-32 addressing modes selected by the ModR/M and SIB bytes

ModR/M byte		Addressing mode
Mod field $b_7\ b_6$	**R/M field** $b_2\ b_1\ b_0$	
0 0	Reg	Register indirect EA = [Reg]
0 1	Reg	Base with 8-bit displacement EA = [Reg] + Disp8
1 0	Reg	Base with 32-bit displacement EA = [Reg] + Disp32
1 1	Reg	Register EA = Reg
Exceptions		
0 0	1 0 1	Direct EA = Disp32
0 0	1 0 0	Base with index (uses SIB byte) EA = [Base] + [Index] × Scale When Base = EBP the addressing mode is: Index with 32-bit displacement EA = [Index] × Scale + Disp32
0 1	1 0 0	Base with index and 8-bit displacement (uses SIB byte) EA = [Base] + [Index] × Scale + Disp8
1 0	1 0 0	Base with index and 32-bit displacement (uses SIB byte) EA = [Base] + [Index] × Scale + Disp32

Table D.3 Scale field encoding in IA-32 SIB byte

Scale field	Scale
0 0	1
0 1	2
1 0	4
1 1	8

modes are determined by the 2-bit Mod field of the ModR/M byte as shown in the first four rows of the table. The 3-bit R/M field normally specifies the register (Reg) involved in these modes. Exceptions to this are used to generate the addressing modes listed in the remainder of the table. All of these modes use the SIB byte, shown in Figure D.1*c*, except for the Direct mode. The SIB byte encodes the base and index registers as shown in Table D.1. The scale factors of 1, 2, 4, and 8, are encoded as shown in Table D.3.

As noted in Table 3.3, the ESP register (encoding 100) cannot be used as an index register. This is not actually a programming restriction because ESP is used as the processor stack pointer. When the bit pattern 100 is placed in the Index field of the SIB byte, no scaled index is used, but the other components of the addressing mode operate normally in generating the effective address of the operand. This has the following useful effect. From Table D.2, it would appear that ESP cannot be used in the first three modes listed because the ESP encoding (100) is used to signify exceptions that generate the last three modes in the table. But if 100 is placed in both the Base and Index fields of the SIB byte, the addressing modes generated are effectively the first three in the table with ESP as the base register because no index is used.

Note that the base register encoding of 101 (EBP) is used as an exception in the Base with index addressing mode in order to generate the Index with 32-bit displacement mode. The EBP register can still be used effectively as the base register in a Base with index mode by using it in the Base with index and displacement mode with a displacement value of zero.

D.2 BASIC INSTRUCTIONS

A set of commonly used IA-32 instructions is listed alphabetically in Table D.4. All of the instructions used in Chapter 3 are included except for the jump instructions and the specialized string instructions with the repeat option that were used in Section 3.21.3 for I/O block transfers. The conditional and unconditional jump instructions are described in the next two subsections. String instructions are described in Section D.4. In Table D.4, the OP-code mnemonic and name of the instruction are shown in the

(Continued on page 782.)

Table D.4 IA-32 instructions

Mnemonic (Name)	Size	Operands		Operation performed	CC flags affected			
		dst	src		S	Z	O	C
ADC (Add with carry)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] + [src] + [CF]	x	x	x	x
ADD (Add)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] + [src]	x	x	x	x
AND (Logical AND)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ∧ [src]	x	x	0	0
BT (Bit test)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# of [dst]				x
BTC (Bit test and complement)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# of [dst]; complement bit# of [dst]				x
BTR (Bit test and reset)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# of [dst]; clear bit# of [dst] to 0				x

Mnemonic (Name)	Size	Operands dst	Operands src	Operation performed	CC flags affected S	Z	O	C
BTS (Bit test and set)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# of [dst]; set bit# of [dst] to 1				x
CALL (Subroutine call)	D	reg mem		ESP ← [ESP] − 4; [ESP] ← [EIP]; EIP ← EA of dst				
CLC (Clear carry)				CF ← 0				0
CLI (Clear int. flag)				IF ← 0				
CMC (Compl. carry)				CF ← $\overline{[CF]}$				x
CMP (Compare)	B,D	reg reg mem reg mem	reg mem reg imm imm	[dst] − [src]	x	x	x	x
DEC (Decrement)	B,D	reg mem		dst ← [dst] − 1	x	x	x	
DIV (Unsigned divide)	B,D		reg mem	for B: [AL]/[src]; AL ← quotient; AH ← remainder for D: [EAX]/[src]; EAX ← quotient; EDX ← remainder	?	?	?	?

(Continued)

Mnemonic (Name)	Size	Operands dst src	Operation performed	CC flags affected S	Z	O	C
HLT (Halt)			Halts execution until reset or external interrupt occurs				
IDIV (Signed divide)	B,D	reg mem	for B: [AL]/[src]: AL ← quotient: AH ← remainder for D: [EAX]/[src]: EAX ← quotient: EDX ← remainder	?	?	?	?
IMUL (Signed multiplication)	B,D	reg mem	(double-length product) for B: AX ← [AL] × [src] for D: EDX,EAX ← [EAX] × [src]	?	?	x	x
	D	reg reg reg mem	(single-length product) reg ← [reg] × [src]	?	?	x	x
IN (Isolated input)	B,D	dst = AL or EAX src = imm8 or [DX]	AL or EAX ← [src]				
INC (Increment)	B,D	reg mem	dst ← [dst] + 1	x	x	x	
INT (Software interrupt)	D	imm8	Push EFLAGS: Push EIP: EIP ← address (determined by imm8)				

Mnemonic (Name)	Size	Operands dst src	Operation performed	CC flags affected S	Z	O	C
IRET (Return from interrupt)	D		Pop EIP; Pop EFLAGS	x	x	x	x
LEA (Load effective address)	D	reg mem	reg ← EA of src				
LOOP (Loop)	D	target	ECX ← [ECX] − 1; If ([ECX] ≠ 0) EIP ← target				
LOOPE (Loop on equal/zero)	D	target	ECX ← [ECX] − 1; If ([ECX] ≠ 0 ∧ [Z] = 1) EIP ← target				
LOOPNE (Loop on not equal/ not zero)	D	target	ECX ← [ECX] − 1; If ([ECX] ≠ 0 ∧ [Z] ≠ 1) EIP ← target				
MOV (Move)	B,D	reg reg reg mem mem reg reg imm mem imm	dst ← [src]				
MOVSX (Sign extend byte into register)	B	reg reg reg mem	reg ← sign extend [src]				

(Continued)

Mnemonic (Name)	Size	Operands dst	Operands src	Operation performed	CC flags affected S	Z	O	C
MOVZX (Zero extend byte into register)	B	reg reg	reg mem	reg ← zero extend [src]				
MUL (Unsigned multiplication)	B,D		reg mem	(double-length product) for B: AX ← [AL] × [src] for D: EDX,EAX ← [EAX] × [src]	?	?	x	x
NEG (Negate)	B,D	reg mem		dst ← 2's-complement [dst]	x	x	x	x
NOP (No operation)				alias for: XCHG EAX,EAX				
NOT (Logical complement)	B,D	reg mem		dst ← $\overline{[dst]}$				
OR (Logical OR)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ∨ [src]	x	x	0	0
OUT (Isolated output)	B,D	dst = imm8 or [DX] src = AL or EAX		dst ← [AL] or [EAX]				

Mnemonic (Name)	Size	Operands dst	Operands src	Operation performed	CC flags affected S	Z	O	C
POP (Pop off stack)	D	reg mem		dst ← [[ESP]]; ESP ← [ESP] + 4				
POPAD (Pop off stack into all registers except ESP)	D			Pop eight doublewords off stack into EDI, ESI, EBP, discard, EBX, EDX, ECX, EAX; ESP ← [ESP] + 32				
PUSH (Push onto stack)	D		reg mem imm	ESP ← [ESP] − 4; [ESP] ← [src]				
PUSHAD (Push all registers onto stack)	D			Push contents of EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI onto stack; ESP ← [ESP] − 32				
RCL (Rotate left with C flag)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.32*b*; src operand is rotation count			?	x
RCR (Rotate right with C flag)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.32*d*; src operand is rotation count			?	x
RET (Return from subroutine)				EIP ← [[ESP]]; ESP ← [ESP] + 4				

(*Continued*)

Mnemonic (Name)	Size	Operands		Operation performed	CC flags affected			
		dst	src		S	Z	O	C
ROL (Rotate left)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.32*a*; src operand is rotation count			?	x
ROR (Rotate right)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.32*c*; src operand is rotation count			?	x
SAL (Shift arithmetic left) same as SHL	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.30*a*; src operand is shift count	x	x	?	x
SAR (Shift arithmetic right)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.30*c*; src operand is shift count	x	x	?	x
SBB (Subtract with borrow)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] − [src] − [CF]	x	x	x	x
SHL (Shift left) same as SAL	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.30*a*; src operand is shift count	x	x	?	x

Mnemonic (Name)	Size	Operands		Operation performed	CC flags affected			
		dst	src		S	Z	O	C
SHR (Shift right)	B,D	reg reg mem mem	imm8 CL imm8 CL	See Figure 2.30*b*; src operand is shift count	x	x	?	x
STC (Set carry flag)				CF ← 1				1
STI (Set interrupt flag)				IF ← 1				
SUB (Subtract)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] − [src]	x	x	x	x
TEST (Test)	B,D	reg mem reg mem	reg reg imm imm	[dst] ∧ [src]; set flags based on result	x	x	0	0
XCHG (Exchange)	B,D	reg reg	reg mem	[reg] ↔ [src]				
XOR (Exclusive OR)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst ← [dst] ⊕ [src]	x	x	0	0

(Concluded)

first column. The second column indicates the operand size that can be used: B (byte) or D (32-bit doubleword). The third column lists possible locations for the source and destination operands, abbreviated as:

reg	–	one of the eight processor registers
mem	–	a memory location
imm	–	an 8- or 32-bit immediate operand
imm8	–	an 8-bit immediate operand

The operation performed by the instruction is described in the fourth column. The last column indicates how the condition code flags are affected by executing the instruction using the following symbols:

x	–	affected
0	–	set to 0
1	–	set to 1
"blank"	–	not affected
?	–	unpredictable

D.2.1 CONDITIONAL JUMP INSTRUCTIONS

The conditional jump instructions are listed in Table D.5. As discussed in Section 3.19.1, the target address is used directly in an assembly language program. The machine instruction actually contains a signed number (an offset) that specifies the distance in bytes to the target address relative to the updated contents of the Instruction Pointer register. Two sizes of offset are used: 1 byte and 4 bytes. The assembler computes the offset when converting an assembly language program to machine language.

D.2.2 UNCONDITIONAL JUMP INSTRUCTIONS

Section 3.19.2 describes the unconditional jump instruction JMP. As well as the relative addressing mode used in conditional jumps, the general addressing modes can be used to specify the target address. This provides more flexibility in implementing multiple-way branching that arises in CASE statements used in high-level languages.

D.3 PREFIX BYTES

Instruction prefix bytes, shown in Figure D.1*a*, are divided into four groups. More than one prefix byte can be used with an instruction. But, only one byte from each group can be used. The first group includes *repeat* byte codes for indicating that the instruction operation is to be repeated some number of times. Instructions that allow this option are called string instructions. They will be described in Section D.4. We saw an example of

Table D.5 IA-32 conditional jump instructions

Mnemonic	Condition name	Condition code test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF ∨ ZF = 0
JBE	Unsigned below or equal	CF ∨ ZF = 1
JGE	Signed greater than or equal	SF ⊕ OF = 0
JL	Signed less than	SF ⊕ OF = 1
JG	Signed greater than	ZF ∨ (SF ⊕ OF) = 0
JLE	Signed less than or equal	ZF ∨ (SF ⊕ OF) = 1

repetition of string instruction operations in the block transfer of doublewords between an I/O device and memory in Section 3.21.3. The streaming SIMD extension (SSE) instructions, described in Sections 3.23.3, 11.3.6, and 11.3.7, are also indicated by a byte code in this group.

Two of the groups consist of only one byte code each. These codes are used to override the default operand size or the default address size, as described in Section D.5.

The fourth group of prefix bytes is used to override the default selection of the segment registers used in generating memory addresses. A general description of the use of segment registers was given in Section 11.3.1.

D.4 OTHER INSTRUCTIONS

The full IA-32 instruction set contains many more instructions than those listed in Table D.4. Four of the instruction types not included in the table are briefly described here.

D.4.1 STRING INSTRUCTIONS

Special instructions are provided to perform common repetitive operations efficiently on data items contained in consecutive memory locations. These data structures are called *strings* and the instructions are called string instructions. The individual items of a string can be bytes or 32-bit doublewords. String instructions can be used, for

example, to move a string from one area of memory to another area or to compare two strings to determine if they are equal.

We will use the string move instruction to illustrate how string instructions are executed. The OP codes MOVSB and MOVSD are used for byte and doubleword moves. These instructions differ from the regular Move instruction in that they consist of only the OP code and do not have explicit operands. The address of the source operand is assumed to be in register ESI, and the destination operand address is assumed to be in EDI. The execution of MOVSB consists of moving a byte from the source location to the destination location and then incrementing the ESI and EDI pointer registers. This instruction can be placed inside a loop to move all bytes of the string. Alternatively, a repeat prefix can be used with the instruction to move the complete string. In this case, in addition to initializing the ESI and EDI registers, the ECX register must be initialized to the length of the string. It is decremented after each byte is transferred. Execution of the instruction

```
REP MOVSB
```

moves a complete string of bytes. The instruction is fetched once and its operation is repeated until the count contents of the ECX register have been decremented to zero.

String instructions with the repeat option are provided for performance reasons. The same task could be programmed by using the instruction

```
MOV  BYTE PTR [EDI],[ESI]
```

inside a loop in which the pointer registers are explicitly incremented and the count register ECX is decremented until it contains zero. But that method would take much longer to execute.

D.4.2 FLOATING-POINT, MMX, AND SSE INSTRUCTIONS

There is a full range of operations on floating-point data in the IEEE format (see Chapter 6) provided by IA-32 instructions. The eight floating-point registers shown in Figure 3.37 are used to hold these data. In addition to add, subtract, multiply, and divide operations, trigonometric functions are also provided.

The MMX (multimedia extension) instructions, described in Section 3.23.2, are used to perform simple arithmetic and logic operations in parallel on short integers packed into 64-bit quadwords located either in the floating-point registers or in memory. These operations are required in graphics and signal-processing applications.

The SSE (streaming SIMD extension) instructions, first introduced in the Pentium III processor and enhanced in the Pentium 4 (see Sections 11.3.6 and 11.3.7) perform parallel arithmetic operations on floating-point numbers packed into a set of eight 128-bit processor registers. These registers are separate from the general-purpose and floating-point registers. The individual data items can be 32-bit or 64-bit floating-point numbers. The SSE instructions are useful for vector and matrix calculations in scientific applications. In the Pentium 4 enhancements, the operands can also be 64-bit integers. These data types are used in encryption and decryption operations in secure data applications.

D.5 SIXTEEN-BIT OPERATION

In Sections 3.16.1 and 11.3.2, it was noted that an IA-32 processor can execute programs in a mode that uses 16-bit addresses and data operands, as used in earlier Intel processors, as well as in the mode that uses 32-bit addresses and data operands, which is the mode that we have described in this book. In either of these two modes, byte operands can also be manipulated. When operating in the 32-bit mode, a bit in the OP code determines whether an operand is a byte or a 32-bit doubleword; in the 16-bit mode, the same bit determines whether an operand is a byte or a 16-bit word. The default mode of operation is set by a bit in the segment descriptors. These descriptors were briefly described in Section 11.3.2.

In our discussions, we have tacitly assumed that the processor is operating in the 32-bit default mode. However, on an instruction-by-instruction basis, the default mode can be overridden for the duration of one instruction by using a prefix byte as the first byte of an instruction as shown in Figure D.1. The default operand size or the default address size, or both, can be overridden by different prefix bytes.

D.6 PROGRAMMING EXPERIMENTS

A convenient way to experiment with assembly language programming is to use the in-line assembly language facility provided with a high-level language. An example of this is given in Chapter 9 where I/O routines are programmed in assembly language inside a C program. Here, we outline how to use the in-line facility in C/C++. The Microsoft Corporation provides a compiler for this language that runs under their Windows operating systems on personal computers built with Intel IA-32 processors.

Figure D.2 shows how the addition loop program in Figure 3.40*a* can be incorporated into a C/C++ program. The assembly language instruction code is encapsulated in the construct

```
_asm { ... }
```

The data declarations and initialization operations are done in C/C++ at the beginning of the program, and the result of executing the assembly language program, which is the value in memory location SUM, is printed by the printf statement at the end of the program.

The operations of naming and opening a file for the source program and entering, compiling, and executing it, are not given because they vary depending on the particular software environment used.

A hexadecimal listing of the machine instructions generated for an assembly language program, such as the one in Figure D.2, can be produced by the compiler. It is instructive to study the listing to see examples of the binary encoding of IA-32 instructions in the format shown in Figure D.1. The listing for the four-instruction loop is shown in Figure D.3. Hexadecimal representations of the bytes used to encode each instruction are shown to the left of the assembly instructions in Figure D.3*a*. Parts *b* and *c* of the figure show the binary encoding details for the ADD and JG instructions.

```
# include <stdio.h>

void main(void)

{
      long NUM1[5];
      long SUM;
      long N;

      NUM1[0] = 17;
      NUM1[1] = 3;
      NUM1[2] = -51;
      NUM1[3] = 242;
      NUM1[4] = 113;
      SUM = 0;
      N = 5;

      _asm {
                        LEA     EBX,NUM1
                        MOV     ECX,N
                        MOV     EAX,0
                        MOV     EDI,0
      STARTADD:         ADD     EAX,[EBX + EDI*4]
                        INC     EDI
                        DEC     ECX
                        JG      STARTADD
                        MOV     SUM,EAX
      }

      printf ("The sum of the list values is %ld \n", SUM );
}
```

Figure D.2 IA-32 Program in Figure 3.40*a* encapsulated in a C/C++ program.

First, consider the ADD instruction. The two bits set to 1 at the right end of the OP code have the following meaning: The last 1 signifies that the operand size is 32 bits. The second last 1 signifies that the source operand is the operand located in the memory. Using Table D.2, we observe that the Mod field (00) and the R/M field (100) of the ModR/M byte specify the Base with index addressing mode for the memory operand, and the Reg/OPcode field (000) specifies the EAX register as the destination. The Base field (011) of the SIB byte specifies EBX as the base register and the Index field (111) specifies EDI as the index register. The Scale field (10) selects 4 as the scale factor.

Machine instructions (hexadecimal)			Assembly language instructions		
03	04	BB	STARTADD:	ADD	EAX,[EBX + EDI*4]
47				INC	EDI
49				DEC	ECX
7F	F9			JG	STARTADD

(a) Loop body encoding

OP code	ModR/M byte	SIB byte
03	04	BB
00000011	00 000 100	10 111 011
ADD (doubleword)	(see Table D.2)	(see Figure D.1*c*)

(b) ADD instruction

OP code	Offset
7F	F9
01111111	111111001
JG (short offset)	−7

(c) JG instruction

Figure D.3 Encoding of the loop body in Figure D.2.

The JG instruction encoding in Figure D.3*c* is interpreted as follows. The first four bits of the OP code (0111) specify a conditional jump with a 1-byte offset, and the last four bits (1111) specify the "greater than" condition. The offset byte contains the 2's-complement representation for −7. This is the distance in bytes from the address of the instruction following the JG instruction back to the address of the ADD instruction at the beginning of the loop.

Use of the Processor Stack

The compiler uses registers ESP and EBP as the processor stack pointer and the frame pointer, respectively. Therefore, in-line assembly language programs cannot use these registers for other purposes. Also, the compiler allocates memory variables, such

as NUM1, SUM, and N, declared inside the "main" procedure, as local variables on the stack. When they are referenced using the Direct addressing mode in assembly language, as in the first two instructions in the program in Figure D.2, the compiler generates the Base plus displacement mode to access them. The frame pointer EBP is used as the base register and the displacements are negative offsets into the stack, which grows toward lower addresses. If these variables are declared outside the "main" procedure, they are allocated as global variables and they are accessed by the Direct addressing mode.