sequence

```
MOV   EAX,A
ADD   EAX,B
MOV   C,EAX
```

The subtract instruction

```
SUB   dst,src
```

performs the operation

$$dst \leftarrow [dst] - [src]$$

Useful one-operand instructions INC and DEC are provided for incrementing or decrementing their operand by 1.

We need a few other instructions before we can give an example of a complete loop program for adding numbers. One of these is a conditional branch instruction. The branch instruction

```
JG   LOOPSTART
```

causes a branch to the instruction at memory location LOOPSTART if the result of the most recent arithmetic operation was greater than 0. All conditional branch instructions start with the letter J, for Jump, and the following letters specify the condition. In this example, the G stands for greater than 0. Other conditional branch instructions will be discussed later.

In order to use a general-purpose register for Register indirect addressing, it is first necessary to load the address of the memory location to be accessed into the register. The IA-32 instruction set provides two ways to do this. If the address is known explicitly as an address label, say LOCATION, it can be loaded into the register by using the Immediate addressing mode in a Move instruction such as

```
MOV   EBX, OFFSET LOCATION
```

This instruction loads the address represented by the label LOCATION into register EBX. Alternatively, an instruction called Load Effective Address, with the mnemonic LEA, may be used. The instruction

```
LEA   EBX,LOCATION
```

has exactly the same effect. The LEA instruction can be used to load any address that is computed dynamically during program execution. For example, suppose there is a need to load the address of a data operand that is referenced by the Base with displacement mode into register EBX. The instruction

```
LEA   EBX,[EBP + 12]
```

loads the address of the operand at location [EBP] + 12 into register EBX. This address depends on the contents of register EBP when the instruction is executed.

A Loop Program for Adding Numbers

Using the instructions just introduced, we can now give a program for adding numbers using a loop. Assume that memory location N contains the number of 32-bit integers in a list that starts at memory location NUM1. The assembly language program shown in Figure 3.40*a* can be used to add the numbers and place their sum in memory location SUM.

Register EBX is loaded with the address value NUM1. It is used as the base register in the Base with index addressing mode in the instruction at the location STARTADD, which is the first instruction of the loop. Register EDI is used as the index register. It is cleared by loading it with zero before the loop is entered. On the first pass through the loop, the first number at location [EBX] = NUM1 is added into the EAX register, which was initially cleared to zero. The index register is then incremented by 1. Thus, on the second pass, the scale factor of 4 in the Add instruction causes the second 32-bit number at byte address NUM1 + 4 to be added into EAX. Subsequent passes will add the numbers at NUM1 + 8, NUM1 + 12, ... Register ECX is used as a counter register. It is initially loaded with the contents of memory location N in the second instruction of the program and is decremented by 1 during each pass through the loop. The conditional

```
           LEA   EBX,NUM1              Initialize base (EBX) and
           MOV   ECX,N                   counter (ECX) registers.
           MOV   EAX,0                 Clear accumulator (EAX)
           MOV   EDI,0                   and index (EDI) registers.
STARTADD:  ADD   EAX,[EBX + EDI * 4]   Add next number into EAX.
           INC   EDI                   Increment index register.
           DEC   ECX                   Decrement counter register.
           JG    STARTADD              Branch back if [ECX] > 0.
           MOV   SUM,EAX               Store sum in memory.
```

(a) Straightforward approach

```
           LEA   EBX,NUM1              Load base register EBX and
           SUB   EBX,4                   adjust to hold NUM1−4.
           MOV   ECX,N                 Initialize counter/index (ECX).
           MOV   EAX,0                 Clear the accumulator (EAX).
STARTADD:  ADD   EAX,[EBX + ECX * 4]   Add next number into EAX.
           LOOP  STARTADD              Decrement ECX and branch
                                         back if [ECX] > 0.
           MOV   SUM,EAX               Store sum in memory.
```

(b) More compact program

Figure 3.40 IA-32 program for adding numbers.

branch instruction JG causes a branch back to STARTADD while [ECX] > 0. When the contents of ECX reach zero, all the numbers have been added. The branch is not taken, and the Move instruction stores the sum in register EAX into memory location SUM.

A more compact program for the same task can be developed by making the following two observations on the program in Figure 3.40*a*. The two-instruction sequence

```
DEC   ECX
JG    STARTADD
```

occurs often at the end of program loops. Hence, the IA-32 instruction set includes a single instruction that combines the operations of these two instructions. The instruction

```
LOOP   STARTADD
```

first decrements the ECX register and then branches to the target address STARTADD if the contents of ECX have not reached zero.

The second observation is that we have used two registers, EDI and ECX, as counters. If we scan the list of numbers to be added in the opposite direction, starting with the last number in the list, only one counter register is needed. We will use register ECX because it is the register referenced implicitly by the LOOP instruction. Assuming $[N] = n$, the first program accesses the numbers using the address sequence NUM1, NUM1 + 4, NUM1 + 8, ..., NUM1 + $4(n - 1)$, as EDI contains the sequence of values 0, 1, 2, ..., $(n - 1)$. The new program, shown in Figure 3.40*b*, uses the address sequence (NUM1 − 4) + $4n$, (NUM1 − 4) + $4(n - 1)$, ..., (NUM1 − 4) + 4(1), as ECX contains the sequence $n, n - 1, \ldots, 1$. Hence, the value in the base register EBX needs to be changed from NUM1 to NUM1 − 4 in the new program in order to account for the difference between the EDI sequence and the ECX sequence. On the last pass through the loop in the new program, before the LOOP instruction is executed, [ECX] = 1 and the last number to be added is accessed at memory location NUM1.

The reader should note that this type of detailed reasoning in properly accounting for 0-origin and 1-origin indexing and for the correct choice for branch conditions often arises when dealing with lists and arrays. It can be the source of subtle errors. The difficulty is mitigated somewhat in high-level languages where list variables are explicitly referenced as LIST(0), LIST(1), ... LIST(n − 1), and loop ranges are related to index values using expressions such as

```
FOR  i  FROM  0  UPTO  (n − 1)
```

or

```
FOR  i  FROM  (n − 1)  DOWNTO  0
```

This brief discussion of some commonly used IA-32 instructions, along with the addition loop program example, provides an introduction to the basic features of the instruction set and the assembly language used with it. We will now describe the format for machine representation of instructions.

3.17.1 MACHINE INSTRUCTION FORMAT

The general format for machine instructions is shown in Figure 3.41. The instructions are variable in length, ranging from one byte (only an OP code, which is always required) to 12 bytes, consisting of up to four fields. The OP-code field consists of one or two bytes, with most instructions requiring only one byte. The addressing mode information is contained in one or two bytes immediately following the OP code. For instructions that involve the use of only one register in generating the effective address of an operand, only one byte is needed in the addressing mode field. Two bytes are needed for encoding the last two addressing modes in Table 3.3. Those modes use two registers to generate the effective address of a memory operand.

If a displacement value is needed in computing an effective address for a memory operand, it is encoded into either one or four bytes in a field that immediately follows the addressing mode field. If one of the operands is an immediate value, then it is placed in the last field of an instruction and it occupies either one or four bytes.

For some simple instructions, such as those that we will discuss first below, the code for a register involved in the instruction is given in the OP-code byte. However, for most instructions and addressing modes, the registers used are specified in the addressing mode field.

In any instruction set, when instructions are encoded in a variable-length format, the bit pattern for an instruction, when read from left to right, must determine the total length of the instruction. This is necessary because successive instructions in a program are placed one after another in the memory, and there is no other information available to indicate the boundaries between them.

One-Byte Instructions

Registers can be incremented or decremented by instructions that occupy only one byte. Examples are

INC EDI

and

DEC ECX

in which the general-purpose registers EDI and ECX are specified by 3-bit codes in the single OP-code byte.

OP code	Addressing mode	Displacement	Immediate
1 or 2 bytes	1 or 2 bytes	1 or 4 bytes	1 or 4 bytes

Figure 3.41 IA-32 instruction format.

Immediate Mode Encoding

The OP code specifies when the Immediate addressing mode is used. For example, the instruction

MOV EAX,820

is encoded into 5 bytes. A one-byte OP code specifies the Move operation, the fact that a 32-bit immediate operand is used, and the name of the destination register. The OP-code byte is directly followed by the 4-byte immediate value of 820. When an 8-bit immediate operand is used, as in the instruction

MOV DL,5

only two bytes are needed to encode the instruction.

Addressing Mode and Displacement Fields

As a general rule, one operand of a two-operand instruction must be in a register. The other operand can also be in a register, or it can be in the memory. There are two exceptions where both operands can be in the memory. The first is the case where the source operand is an immediate operand, and the destination operand is in the memory. The second is the case of instructions for Push and Pop operations on the processor stack. The stack is located in the stack segment of memory, and it is possible to push a memory operand onto the stack or to pop an operand from the stack into the memory. We will discuss this later in Section 3.22.

When both operands are in registers, only one addressing mode byte is needed. For example, the instruction

ADD EAX,EDX

is encoded into two bytes. The first byte contains the OP code and the other byte is an addressing mode byte that specifies the two registers.

Now let us consider a few examples of instruction encoding where one operand is in a register and the other is in memory. The instruction

MOV ECX,N

in the programs in Figure 3.40 is encoded in six bytes: one for the OP code, one for the addressing mode byte that specifies both the Direct mode and the destination register ECX, and four bytes for the address of memory location N.

The instruction

ADD EAX,[EBX + EDI*4]

in the same programs requires two addressing mode bytes because two registers are used to generate the effective address of the source operand. The scale factor of 4 is also included in the second of these two bytes. Thus, the instruction requires a total of three bytes, including the OP-code byte.

As a third example, consider the instruction

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

The assembly language directive DWORD PTR is needed to specify that the operand length is 32 bits for the immediate operand with the value 10. In other assembly languages, specification of the size of the operand is often included in the OP-code mnemonic. For example, in the Motorola 68000, discussed in Part II of this chapter, MOVE.B specifies a 1-byte operand, and MOVE.L specifies a 4-byte long-word operand. If a 32-bit displacement value DISP is used, a total of 11 bytes is needed to encode this instruction: one byte for the OP code, two for the addressing mode field, and four bytes each for the displacement and immediate fields. It is noted in Table 3.3 that displacements can have a length of either 8 bits or 32 bits. The size is specified in the first of the two addressing mode bytes.

In the encoding of two-operand instructions, the specifications of the register operand and the memory operand are placed in a fixed order, with the register operand always being specified first. In order to distinguish between the instructions

```
MOV    EAX,LOCATION
```

which loads the contents of memory location LOCATION into register EAX, and the instruction

```
MOV    LOCATION,EAX
```

which stores the contents of EAX into LOCATION, the OP-code byte contains a bit called the *direction bit*. This bit indicates which operand is the source.

The encoding of OP codes and addressing modes in the IA-32 architecture is somewhat complex and has a number of nonuniformities and exceptions. While this makes it difficult for a compiler to take advantage of all of the features of the instruction set and addressing modes, there is no doubt that the IA-32 architecture has very powerful and flexible features.

Appendix D contains a summary of the IA-32 instructions and a guide to entering and running assembly language programs on a personal computer.

3.18 IA-32 ASSEMBLY LANGUAGE

Basic aspects of the IA-32 assembly language for specifying OP codes, addressing modes, and instruction address labels are illustrated by the programs in Figure 3.40. As discussed in Section 2.6.1, assembler directives are needed to define the data area of a program and to define the correspondence between symbolic names for data locations and the actual physical address values.

A complete assembly language program for the program in Figure 3.40*b* is shown in Figure 3.42. The *.data* and *.code* assembler directives define the beginning of the data and code (instruction) sections of the program. In the data section, the DD directives allocate 4-byte doubleword data locations. NUM1 is the label assigned to the address of the first of five doublewords initialized to the decimal values 17, 3, −51, 242, and −113. The next two doubleword locations, initialized to 5 and 0, are given the address labels N and SUM.

Assembler directives	.data		
	NUM1	DD	17,3,–51,242,–113
	N	DD	5
	SUM	DD	0
	.code		
Statements that generate machine instructions	MAIN :	LEA	EBX,NUM1
		SUB	EBX,4
		MOV	ECX,N
		MOV	EAX,0
	STARTADD :	ADD	EAX,[EBX+ECX * 4]
		LOOP	STARTADD
		MOV	SUM,EAX
Assembler directive		END	MAIN

Figure 3.42 Complete IA-32 assembly language representation for the program in Figure 3.40*b*.

The three symbolic names declared in the data section are used in the addressing modes of the instructions in the code section. The MAIN label is used to specify the location where instruction execution is to begin, and this label is used in the END assembler directive that terminates the text file for the program. Other assembler directives, such as EQU which was discussed in Section 2.6.1, are also available.

3.19 PROGRAM FLOW CONTROL

There are two main ways in which the flow of executing instructions varies from straight-line sequencing. Calls to subroutines and returns from them break straight-line sequencing, as will be discussed in Section 3.22. Also, branch instructions, either conditional or unconditional, can cause a break. We will now discuss branch instructions. In IA-32 terminology, branch instructions are called Jumps.

3.19.1 CONDITIONAL JUMPS AND CONDITION CODE FLAGS

The instruction

```
JG   STARTADD
```

in Figure 3.40*a* is an example of a conditional jump instruction. The condition is "Greater than 0," as signified by the G suffix in the OP code. This condition is related to the results of the most recently executed data manipulation instruction, which

is the

DEC ECX

instruction in this example. Properties of the results generated by instructions such as Decrement, Add, or others that perform arithmetic and comparison operations are recorded in four condition code flags in the processor Status Register, as shown in Figure 3.37. These flags, called SF (sign), ZF (zero), OF (overflow), and CF (carry), are set to 1 or cleared to 0 as described in Section 2.4.6, where they were called N, Z, V, and C, respectively, with one exception. On a subtract operation, the CF bit is set to 1 if no carry occurs, signifying a borrow signal. The flags can be tested in subsequent conditional jump instructions to determine whether or not the jump should be taken. In our example, execution control switches to the instruction at the jump target address STARTADD if the condition [ECX] > 0 holds.

Conditional jump instructions do not contain the jump target address as an absolute address. They contain a signed number that is added to the contents of the Instruction Pointer register to determine the target address. Thus, the target address is given relative to the address in the Instruction Pointer. The first step performed after an instruction is fetched is to advance the Instruction Pointer to point to the next instruction. Hence, the Instruction Pointer contains the address of the instruction that immediately follows the jump instruction when the relative offset to the jump target address is added to it. In our example, assume that the address STARTADD is 1000. A total of seven bytes is needed to encode the four instructions ADD, INC, DEC, and JG in Figure 3.40*a*. The updated contents of the Instruction Pointer register, EIP, will be 1007, which is the address of the last MOV instruction in the program. Therefore, the relative distance to the jump target address is −7, and that is the value stored in the conditional jump instruction. This small negative number can be represented in one byte. Hence, including the OP-code byte, only two bytes are needed to encode a conditional jump instruction when the relative address of the target is in the range −128 through +127. When the jump target is farther away, a 4-byte offset is used.

In this example, the result of decrementing the ECX register is tested to see if it is greater than zero. Other arithmetic properties of a result can be tested by different conditional jump instructions. For example, Jump if equal to 0, and Jump if sign bit is 1 (negative) are done by instructions with the OP codes JZ (or JE) and JS, respectively.

Compare Instructions

It is often necessary to make conditional jumps in a program based on the result of comparing two numbers. The Compare instruction

CMP dst,src

performs the operation

[dst] − [src]

and sets the condition code flags based on the result obtained. Neither of the operands is changed. The first operand is always compared to the second. For example, if we follow the Compare instruction by a conditional jump that is based on the "greater than" condition, then we wish to take the jump to the target address if the destination operand is greater than the source operand.

3.19.2 UNCONDITIONAL JUMP

An unconditional jump instruction, JMP, always causes a branch to the instruction at the target address. In addition to using short (one byte) or long (four bytes) relative signed offsets to determine the target address, as is done in conditional jump instructions, the JMP instruction also allows the use of other addressing modes. This flexibility in generating the target address can be very useful. Consider the Case statement that is found in many high-level languages. At some point in a program, exactly one of a number of alternative calculations is to be performed. Each of these is referred to as a case. Suppose that the 4-byte addresses of the first instruction for each of the routines corresponding to the cases are stored in a table in the memory, starting at a location labeled JUMPTABLE. If the cases are numbered with indices 0, 1, 2, ... and the index of the case to be executed is loaded into index register ESI, then a jump to the selected case can be performed by executing the instruction

JMP [JUMPTABLE + ESI*4]

which uses the Index with displacement addressing mode.

3.20 LOGIC AND SHIFT/ROTATE INSTRUCTIONS

3.20.1 LOGIC OPERATIONS

The IA-32 architecture has instructions that perform the logic operations AND, OR, and XOR. The operation is performed bitwise on two operands, and the result is placed in the destination location. For example, suppose register EAX contains the hexadecimal pattern 0000FFFF and register EBX contains the pattern 02FA62CA. Then the instruction

AND EBX,EAX

will clear the left half of EBX to all zeros, and leave the right half unchanged. The result in EBX will be 000062CA.

There is also a NOT instruction which generates the logical complement of all bits of the operand, that is, it changes all 1s to 0s and all 0s to 1s.

3.20.2 SHIFT AND ROTATE OPERATIONS

An operand can be shifted right or left, using either logical or arithmetic shifts, by a number of bit positions determined by a specified count. The format of the shift instructions is

OPcode dst,count

where the destination operand to be shifted is specified by the general addressing modes and the count is given either as an 8-bit immediate value or is contained in the 8-bit

```
LEA   EBP,LOC         EBP points to first byte.
MOV   AL,[EBP]        Load first byte into AL.
SHL   AL,4            Shift left by 4 bit positions.
MOV   BL,[EBP+1]      Load second byte into BL.
AND   BL,0FH          Clear high-order 4 bits to zero.
OR    AL,BL           Concatenate the BCD digits.
MOV   PACKED,AL       Store the result.
```

Figure 3.43 An IA-32 routine to pack two BCD digits into a byte.

register CL. There are four shift instructions:

SHL	Shift left logical
SHR	Shift right logical
SAL	Shift left arithmetic (same as SHL)
SAR	Shift right arithmetic

Shift operations are discussed in Section 2.10 and illustrated in Figure 2.30.

There are also four rotate instructions. They are ROL and ROR for left and right rotate without the carry flag CF, and RCL and RCR for rotations that include CF. All four operations are illustrated in Figure 2.32.

Digit-Packing Program

As a simple application of these instructions, consider the BCD digit-packing program shown in Figure 2.31. The IA-32 code for this routine is shown in Figure 3.43. The two ASCII bytes are loaded into registers AL and BL. The SHL instruction shifts the byte in AL four bit positions to the left, filling the low-order four bits with zeros. The second entry in the operand field of this instruction is a count that indicates the number of bit positions to be shifted. The AND instruction clears the high-order four bit positions of the second byte to zeros. Finally, the 4-bit patterns that are the desired BCD codes are combined in AL with the OR instruction and then stored in memory byte location PACKED.

3.21 I/O OPERATIONS

3.21.1 MEMORY-MAPPED I/O

Input/Output device buffer registers are most commonly accessed in modern computer systems by the memory-mapped I/O method as described in Section 2.7. The IA-32 Move instruction can be used to transfer directives to I/O devices, and to transfer data and status information to and from devices. For example, suppose that keyboard and display devices have their synchronization flags SIN and SOUT (see Figure 2.19) stored

in bit 3 of device status registers INSTATUS and OUTSTATUS, respectively. Using program-controlled I/O, a byte can be read from the keyboard buffer register DATAIN into register AL using the wait loop

```
READWAIT:  BT    INSTATUS,3
           JNC   READWAIT
           MOV   AL,DATAIN
```

The instruction BT is a bit-test instruction. The value in the destination bit position specified by the source operand (in this case, the immediate value 3) is loaded into the carry flag CF. The conditional jump JNC (jump if no carry) causes a jump to READWAIT if CF = 0.

Similarly, an output operation to send a byte from register AL to the display buffer register DATAOUT is performed by

```
WRITEWAIT:  BT    OUTSTATUS,3
            JNC   WRITEWAIT
            MOV   DATAOUT,AL
```

An IA-32 program that reads one line of characters from a keyboard, stores them in memory starting at address LOC, and echoes them back out to the display, is shown in Figure 3.44. This program is patterned after the generic program in Figure 2.20. Register EBP points to the memory area where the line is to be stored.

3.21.2 ISOLATED I/O

The IA-32 instruction set also has two instructions, with OP codes IN and OUT, that are used only for I/O purposes. The addresses issued by these instructions are in an address space that is separate from the memory address space used by the other instructions.

```
        LEA   EBP,LOC           EBP points to memory area.
READ:   BT    INSTATUS,3        Wait for character to be
        JNC   READ                entered into DATAIN.
        MOV   AL,DATAIN         Transfer character into AL.
        MOV   [EBP],AL          Store the character in memory
        INC   EBP                 and increment pointer.
ECHO:   BT    OUTSTATUS,3       Wait for display to
        JNC   ECHO                be ready.
        MOV   DATAOUT,AL        Send character to display.
        CMP   AL,CR             If not carriage return,
        JNE   READ                read more characters.
```

Figure 3.44 An IA-32 program that reads a line of characters and displays it.

This arrangement is called *isolated I/O* to distinguish it from memory-mapped I/O in which the addressable locations in I/O devices are in the same address space as memory locations. The same address and data lines on Intel processor chips are used for both address spaces. An output control line is used to indicate which address space is referenced by the current instruction.

Sixteen-bit addresses are used in the byte-addressable I/O address space. There are 8-bit and 32-bit I/O device operand locations that hold data, status, and command information. The first 256 addresses can be specified directly using an 8-bit field in the In and Out instructions. The format for the input instruction using this mode is

IN REG,DEVADDR

where the destination REG must be register AL or EAX, denoting an 8-bit or a 32-bit operand transfer, respectively. The last field in the instruction contains the 8-bit device address DEVADDR. The corresponding output instruction is

OUT DEVADDR,REG

Since the address space is byte addressable, a keyboard device that can send an 8-bit ASCII character to the processor could have its data buffer register at byte address DEVADDR and its 8-bit status register at address DEVADDR + 1.

The full 16-bit I/O address space spans 64K locations; it can be referenced through the DX register using the input instruction

IN REG,DX

where, as before, REG must be AL or EAX. The 16-bit device address is contained in the DX register, which is the low-order 16 bits of the EDX register, and the width of the data transfer is determined by the size of the REG operand. The corresponding output instruction is

OUT DX,REG

3.21.3 BLOCK TRANSFERS

In addition to the instructions IN and OUT that transfer a single item of information between the processor and an I/O device, the IA-32 architecture also has two block transfer I/O instructions: REPINS and REPOUTS. They transfer a block of data items serially, one item at a time, between the memory and an I/O device. The S suffix in the OP codes stands for string, and the REP prefix stands for "repeat the item by item transfer until the complete block has been transferred". The instructions themselves do not specify the parameters needed to describe the transfer. These parameters are specified implicitly by processor registers DX, EDI, and ECX as follows:

DX contains a 16-bit I/O device address.

EDI contains a 32-bit address for the beginning of a block in memory.

ECX contains the number of data items to be transferred.

A suffix B or D in the OP-code mnemonic indicates that the item size is either of byte or doubleword length. Thus, REPINSB is a byte-block transfer, and REPINSD is a doubleword-block transfer. The block transfer instructions operate as follows: After each data item is transferred, the index register EDI is incremented by 1 or 4, depending on the size of the data items, and the ECX register is decremented by 1. The transfers are repeated until the contents of the counter register ECX have been decremented to 0. The effect of these single instructions is equivalent to a program loop that uses register ECX as the loop counter.

As an example, suppose that a block of 128 doublewords is to be transferred from a disk storage device into the memory. Assume that the addressable data buffer register in the disk device contains a doubleword data item, and has the I/O address DISKDATA. The data block is to be written into the memory beginning at address MEMBLOCK. The counter register ECX has to be initialized to 128. The instruction sequence

```
LEA      EDI,MEMBLOCK
MOV      DX,DISKDATA
MOV      ECX,128
REPINSD
```

can be used to accomplish the transfer. This assumes that MEMBLOCK has been defined as an address label, and DISKDATA has been defined by an EQU assembler directive to represent the 16-bit address of the device data buffer register.

This discussion illustrates how isolated I/O can be organized. It also shows how a multi-item block transfer can be performed by a single instruction with the use of an address counter (EDI) and a data item counter (ECX).

3.22 SUBROUTINES

As explained in Section 2.9, the processor stack data structure is convenient for handling entry to and return from subroutines. In the IA-32 architecture, register ESP is used as the stack pointer, which points to the current top element (TOS) in the processor stack. The stack grows toward lower numbered addresses. This arrangement is the same as discussed in Section 2.8 and illustrated in Figures 2.21 and 2.22. The width of the stack is 32 bits, that is, all stack entries are doublewords.

There are four instructions for pushing elements onto the stack and for popping them off. The instruction

```
PUSH    src
```

decrements ESP by 4, and then stores the doubleword at location src into the memory location pointed to by ESP. The instruction

```
POP    dst
```

reverses this process by retrieving the TOS doubleword from the location pointed to by ESP, storing it at location dst, and then incrementing ESP by 4, thus effectively removing

the TOS element from the stack. These instructions implicitly use ESP as the stack pointer. The source and destination operands are specified using the IA-32 addressing modes. Two instructions push or pop multiple register contents. The instruction

PUSHAD

pushes the contents of all eight general-purpose registers EAX through EDI onto the stack, and the instruction

POPAD

pops them off in the reverse order. When POPAD reaches the old stored value of ESP, it discards those four bytes without loading them into ESP and continues to pop the remaining values into their respective registers. These two instructions are used to efficiently save and restore the contents of all registers as part of implementing subroutines.

The list addition program in Figure 3.40*a* can be written as a subroutine as shown in Figure 3.45. Parameters are passed through registers. Memory address NUM1 of the first of the numbers to be added is loaded into register EBX by the calling program, and the number of values to be added, contained in memory location N, is loaded into register ECX. The calling program expects to get the sum of the numbers in the list passed back to it in register EAX. Thus, the registers EBX, ECX, and EAX are used for passing parameters. Register EDI is used by the subroutine as an index register in performing the addition, so its contents have to be saved and restored in the subroutine by PUSH and POP instructions.

The subroutine is called by the instruction

CALL LISTADD

which first pushes the return address onto the stack and then branches to LISTADD. The contents of the stack after the subroutine has saved (PUSHed) EDI are shown in Figure 3.45*b*. The return address is the address of the MOV instruction that immediately follows the CALL instruction in the calling program. The instruction RET returns execution control to the calling program by popping the TOS element into the Instruction Pointer (program counter), EIP.

Figure 3.46 shows the program of Figure 3.40*a* rewritten as a subroutine with parameters passed on the stack. The parameters NUM1 and *n* are pushed onto the stack by the two PUSH instructions in the calling program. The top of the stack is at level 2 after the Call instruction has been executed. Registers EDI, EAX, EBX, and ECX serve the same purpose in this subroutine as in the subroutine in Figure 3.45. Their values are saved and they are loaded with initial values and parameters by the first eight instructions in the subroutine. At this point, the top of the stack is at level 3. When the numbers have been added by the four-instruction loop, the sum is placed into the stack, overwriting the parameter NUM1. After the Return instruction is executed, the ADD and POP instructions in the calling program remove the parameter *n* from the stack and pop the result sum into the memory location SUM, restoring the top of the stack to level 1.

Calling program

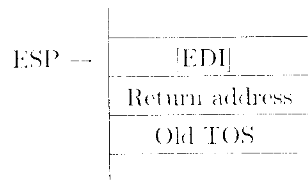
```
        ⋮
        LEA     EBX,NUM1              Load parameters
        MOV     ECX,N                   into EBX,ECX.
        CALL    LISTADD               Branch to subroutine.
        MOV     SUM,EAX               Store sum into memory.
        ⋮
```

Subroutine

```
LISTADD:    PUSH    EDI                   Save EDI.
            MOV     EDI,0                 Use EDI as index register.
            MOV     EAX,0                 Use EAX as accumulator register.

STARTADD:   ADD     EAX,[EBX + EDI * 4]   Add next number.
            INC     EDI                   Increment index.
            DEC     ECX                   Decrement counter.
            JG      STARTADD              Branch back if [ECX] > 0.
            POP     EDI                   Restore EDI.
            RET                           Branch back to Calling program.
```

(a) Calling program and subroutine



(b) Stack contents after saving EDI in subroutine

Figure 3.45 Program of Figure 3.40*a* written as an IA-32 subroutine; parameters passed through registers.

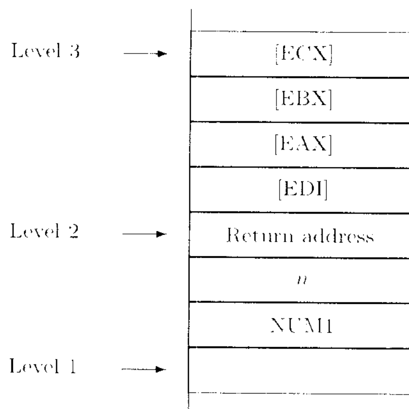
The last example of subroutines that we will discuss is the case of handling nested calls. Figure 3.47 shows the IA-32 code for the program in Figure 2.28. The stack frames corresponding to the first and second subroutines are shown in Figure 3.48. Register EBP is used as the frame pointer. The structures of the calling program and the subroutines are very close to those in Figure 2.28. The essential differences in Figure 3.47 are that multiple PUSH and POP instructions are used for saving and

```
(Assume top of stack is at level 1 below.)
Calling program

            PUSH   OFFSET NUM1        Push parameters onto the stack.
            PUSH   N
            CALL   LISTADD            Branch to the subroutine.
            ADD    ESP,4              Remove n from the stack.
            POP    SUM                Pop the sum into SUM.
            :
Subroutine

LISTADD:    PUSH   EDI                Save EDI and use
            MOV    EDI,0                as index register.
            PUSH   EAX                Save EAX and use as
            MOV    EAX,0                as accumulator register.
            PUSH   EBX                Save EBX and load
            MOV    EBX,[ESP+20]         address NUM1.
            PUSH   ECX                Save ECX and
            MOV    ECX,[ESP+20]         load count n.
STARTADD:   ADD    EAX,[EBX+EDI*4]    Add next number.
            INC    EDI                Increment index.
            DEC    ECX                Decrement counter.
            JG     STARTADD           Branch back if not done.
            MOV    [ESP+24],EAX       Overwrite NUM1 in stack with sum.
            POP    ECX                Restore registers.
            POP    EBX
            POP    EAX
            POP    EDI
            RET                       Return.
```

(a) Calling program and subroutine



(b) Stack contents at different times

Figure 3.46 Program of Figure 3.40*a* written as an IA-32 subroutine; parameters passed on the stack.

Address		Instructions		Comments
Calling program				
		⋮		
2000		PUSH	PARAM2	Place parameters
2006		PUSH	PARAM1	on stack.
2012		CALL	SUB1	
2017		POP	RESULT	Store result.
		ADD	ESP,4	Restore stack level.
		⋮		
First subroutine				
2100	SUB1:	PUSH	EBP	Save frame pointer register.
		MOV	EBP,ESP	Load frame pointer.
		PUSH	EAX	Save registers.
		PUSH	EBX	
		PUSH	ECX	
		PUSH	EDX	
		MOV	EAX,[EBP+8]	Get first parameter.
		MOV	EBX,[EBP+12]	Get second parameter.
		⋮		
		PUSH	PARAM3	Place parameter on stack.
2160		CALL	SUB2	
2165		POP	ECX	Pop SUB2 result into ECX.
		⋮		
		MOV	[EBP+8],EDX	Place answer on stack.
		POP	EDX	Restore registers.
		POP	ECX	
		POP	EBX	
		POP	EAX	
		POP	EBP	Restore frame pointer register.
		RET		Return to Main program.
Second subroutine				
3000	SUB2:	PUSH	EBP	Save frame pointer register.
		MOV	EBP,ESP	Load frame pointer.
		PUSH	EAX	Save registers.
		PUSH	EBX	
		MOV	EAX,[EBP+8]	Get parameter.
		⋮		
		MOV	[EBP+8],EBX	Place SUB2 result on stack.
		POP	EBX	Restore registers.
		POP	EAX	
		POP	EBP	Restore frame pointer register.
		RET		Return to first subroutine.

Figure 3.47 Nested subroutines in IA-32 assembly language.

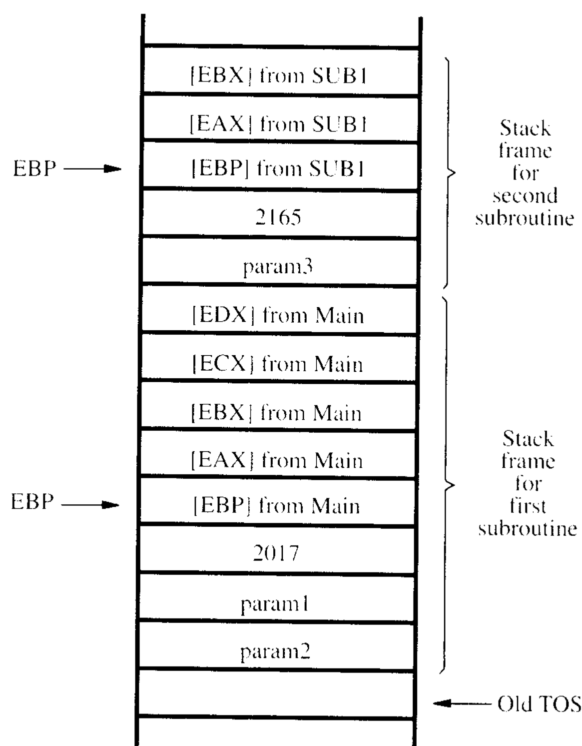


Figure 3.48 IA-32 stack frames for Figure 3.47.

restoring registers in place of the MoveMultiple instructions in Figure 2.28. The IA-32 instruction set has instructions PUSHAD and POPAD that can be used to push and pop all eight general-purpose registers, as described earlier in this section, but we have chosen to use individual PUSH and POP instructions in Figure 3.47 because only half of the register set is used by the subroutines.

3.23 OTHER INSTRUCTIONS

We have described only a small part of the full set of IA-32 instructions. Here, we will describe a few more important instructions.

3.23.1 MULTIPLY AND DIVIDE INSTRUCTIONS

In addition to the Add and Subtract instructions for signed integer numbers, discussed in Section 3.17, there are instructions for integer multiply and divide, as well as instructions for operations on floating-point numbers.

In general, the multiplication of two 32-bit integers produces a double-length product of 64 bits. However, in many applications, it may be known that the product is only a single-length 32-bit result. Different instructions are used for these two cases. For the latter case, which is more common, the instruction

IMUL REG,src

places the single-length product into the destination location, which must be a general-purpose register REG. The source operand can be in a register or in the memory. For the first case, the instruction

IMUL src

uses the EAX register as the destination. The source can be in a register or in the memory. The double-length product is placed in registers EDX (high-order half) and EAX (low-order half).

The instruction for integer divide has the format

IDIV src

The source operand (src) is the divisor. The dividend is assumed to be in register EAX. The quotient result is placed in EAX and the remainder is placed in EDX. The dividend in EAX must be sign-extended into EDX before the instruction is executed. This is done by the instruction CDQ (convert doubleword to quadword), which has no operands because the registers involved are assumed to be EAX and EDX.

Floating-point numbers, which are described in Chapter 6, have a much larger range than integers and are needed for scientific calculations. Instructions for the full range of arithmetic operations on these numbers are provided. The operands and results are held in the floating-point registers shown in Figure 3.37. Both single-precision (32-bit) and double-precision (64-bit) formats are supported in the instruction set.

3.23.2 MULTIMEDIA EXTENSION (MMX) INSTRUCTIONS

A two-dimensional graphic or video image can be represented by an array of a large number of sampled image points. The color and brightness of each point, called a *pixel* (picture element), can be encoded into an 8-bit data item. Processing of such data has two main characteristics. The first is that manipulations of individual pixels often involve very simple arithmetic or logic operations. The second is that very high computational rates may be needed for some real-time display applications. The same characteristics apply to sampled audio signals or speech processing, where a sequence of signed numbers represents samples of a continuous analog signal taken at periodic intervals.

In both of these applications, processing efficiency is achieved if the individual data items, which are usually bytes or 16-bit words, are packed into small groups that can be processed in parallel. The IA-32 instruction set has a number of instructions that operate in parallel on such data packed into 64-bit quadwords. (A quadword contains eight bytes or four 16-bit words.) These instructions are called *multimedia extension* (MMX) instructions. The operands can be in the memory or in the eight floating-point

registers. Thus, these registers serve a dual purpose. They can hold either floating-point numbers or MMX operands. When used by MMX instructions, the registers are referred to as MM0 through MM7.

Move instructions are provided for transferring 64-bit MMX operands between the memory and the MMX registers. The instruction

PADDB   MMi,src

adds the corresponding bytes of two 8-byte operands individually and places eight sums in the destination register. The source can be in the memory or in an MMX register, but the destination must be an MMX register. Similar instructions are provided for subtraction and for logic operations.

A common operation in signal processing applications is to multiply a short time sequence of input signal samples by constants, called *weights*, and add the products together to produce an output signal sample. An MMX instruction that combines the multiply and add operations is provided. It operates on 64-bit MMX operands that contain four 16-bit signal sample data items.

3.23.3 VECTOR (SIMD) INSTRUCTIONS

A set of instructions that are used to perform arithmetic operations on small groups of floating-point numbers is provided. SIMD (single-instruction multiple-data) instructions are useful for vector and matrix calculations in scientific applications. In Intel terminology, these instructions are called *streaming SIMD extension* (SSE) instructions. They handle compound operands that are 128 bits long, each consisting of four 32-bit floating-point numbers. Eight 128-bit registers are available for holding these operands. (These registers are not shown in Figure 3.37.) Add and Multiply are two of the basic instructions provided in this group. They operate on the four corresponding pairs of 32-bit operands in the 128-bit compound source and destination operands and place the four individual results in the 128-bit destination location.

3.24 PROGRAM EXAMPLES

This section presents the IA-32 code for the example programs described in Section 2.11.

3.24.1 VECTOR DOT PRODUCT PROGRAM

Figure 3.49 shows an IA-32 program for computing the dot product of two vectors of numbers stored in the memory starting at addresses AVEC and BVEC. This program is patterned after the generic program in Figure 2.33. In Figure 3.49, the Base with index addressing mode is used to access successive elements of each vector. Register EDI is used as the index register. A scale factor of 4 is used because the vector elements are assumed to be doubleword (4-byte) numbers. Register ECX is used as the loop counter,

```
            LEA    EBP,AVEC             EBP points to vector A.
            LEA    EBX,BVEC             EBX points to vector B.
            MOV    ECX,N                ECX is the loop counter.
            MOV    EAX,0                EAX accumulates the dot product.
            MOV    EDI,0                EDI is an index register.
LOOPSTART:  MOV    EDX,[EBP+EDI*4]      Compute the product
            IMUL   EDX,[EBX+EDI*4]        of next components.
            INC    EDI                  Increment index.
            ADD    EAX,EDX              Add to previous sum.
            LOOP   LOOPSTART            Branch back if not done.
            MOV    DOTPROD,EAX          Store dot product in memory.
```

Figure 3.49 An IA-32 dot product program.

initialized to n. This allows the use of the LOOP instruction (see Section 3.17 and Figure 3.40*b*) that first decrements ECX and then branches conditionally to the target address LOOPSTART if the contents of ECX have not reached zero. The product of two vector elements is assumed to fit into a doubleword, so the Multiply instruction IMUL explicitly specifies the desired destination register EDX, as discussed in Section 3.23.

3.24.2 BYTE-SORTING PROGRAM

Figure 3.50*b* shows the IA-32 code for the byte-sorting program of Figure 2.34*b*. Register EAX is loaded with the address LIST and serves as a base register in accessing bytes of the list by using the Base with index addressing mode. Register EDI serves as an index register for the j index in the outer loop, and register ECX serves as an index register for the k index in the inner loop. Register DL holds the current largest byte as each sublist is traversed. The IA-32 program in Figure 3.50*b* matches the generic program in Figure 2.34*b* instruction for instruction except for the interchange of LIST(k) with LIST(j). The single instruction XCHG achieves this in the IA-32 code, while three instructions and a temporary register are needed in the generic code.

3.24.3 LINKED-LIST INSERTION AND DELETION SUBROUTINES

The insertion and deletion subroutines in Figures 3.51 and 3.52 mirror the corresponding generic programs in Figures 2.37 and 2.38 very closely. Parameters are passed through registers, and register names RHEAD, RNEWREC, RIDNUM, RCURRENT, and RNEXT correspond to the same usage as in the generic routines. These names have been used instead of the IA-32 names EAX, EBX, and so on. A sixth register, RNEWID, has been used to hold the ID number of the new record to be inserted. It is loaded with the ID number of the new record by the first instruction of the subroutine

```
for   (j = n-1; j > 0; j = j - 1)
      { for ( k = j-1; k >= 0; k = k - 1 )
          { if   (LIST[k] > LIST[j])
                 { TEMP = LIST[k];
                   LIST[k] = LIST[j];
                   LIST[j] = TEMP;
                 }
          }
      }
```

(a) C-language program for sorting

```
        LEA     EAX,LIST          Load list pointer base
        MOV     EDI,N               register (EAX), and initialize
        DEC     EDI                 outer loop index register
                                    (EDI) to j = n - 1.
OUTER:  MOV     ECX,EDI           Initialize inner loop index
        DEC     ECX                 register (ECX) to k = j - 1.
        MOV     DL,[EAX + EDI]    Load LIST(j) into register DL.
INNER:  CMP     [EAX + ECX],DL    Compare LIST(k) to LIST(j).
        JLE     NEXT              If LIST(k) ≤ LIST(j), go to
                                    next lower k index entry;
        XCHG    [EAX + ECX],DL    Otherwise, interchange LIST(k)
                                    and LIST(j), leaving
        MOV     [EAX + EDI],DL      new LIST(j) in DL.
NEXT:   DEC     ECX               Decrement inner loop index k.
        JGE     INNER             Repeat or terminate inner loop.
        DEC     EDI               Decrement outer loop index j.
        JG      OUTER             Repeat or terminate outer loop.
```

(b) IA-32 program implementation

Figure 3.50 An IA-32 byte-sorting program using straight-selection sort.

in Figure 3.51. The rest of the subroutine is an instruction for instruction match with the subroutine in Figure 2.37. The subroutine for deleting a record, shown in Figure 3.52, is also an instruction for instruction match with the subroutine in Figure 2.38.

As with the generic programs in Figures 2.37 and 2.38, the IA-32 insertion subroutine assumes that the ID number of the new record does not match that of any record already in the list, and the IA-32 deletion subroutine assumes that there exists a record in the list with an ID number that matches the contents of RIDNUM. Problems 3.72

Subroutine

```
INSERTION:  MOV   RNEWID,[RNEWREC]
            CMP   RHEAD,0                   Check if list empty.
            JG    HEAD
            MOV   RHEAD,RNEWREC             If yes, new record becomes
            RET                               one-entry list.
HEAD:       CMP   RNEWID,[RHEAD]            Check if new record
                                              becomes head.
            JG    SEARCH
            MOV   [RNEWREC+4],RHEAD         If yes, make new record
            MOV   RHEAD,RNEWREC               the head.
            RET
SEARCH:     MOV   RCURRENT,RHEAD            Otherwise, use
LOOPSTART:  MOV   RNEXT,[RCURRENT+4]          RCURRENT
            CMP   RNEXT,0                     and RNEXT
            JE    TAIL                        to move through
            CMP   RNEWID,[RNEXT]              the list to find
            JL    INSERT                      the insertion point.
            MOV   RCURRENT,RNEXT
            JMP   LOOPSTART
INSERT:     MOV   [RNEWREC+4],RNEXT
TAIL:       MOV   [RCURRENT+4],RNEWREC
            RET
```

Figure 3.51 An IA-32 subroutine for inserting a new record into a linked list.

Subroutine

```
DELETION:   CMP   RIDNUM,[RHEAD]            Check if head.
            JG    SEARCH
            MOV   RHEAD,[RHEAD+4]           If yes, remove.
            RET
SEARCH:     MOV   RCURRENT,RHEAD            Otherwise,
LOOPSTART:  MOV   RNEXT,[RCURRENT+4]          use RCURRENT
            CMP   RIDNUM,[RNEXT]              and RNEXT
            JE    DELETE                      to move through
            MOV   RCURRENT,RNEXT              the list to
            JMP   LOOPSTART                   find the record.
DELETE:     MOV   RTEMP,[RNEXT+4]
            MOV   [RCURRENT+4],RTEMP
            RET
```

Figure 3.52 An IA-32 subroutine for deleting a record from a linked list.

and 3.73 consider how the subroutines should be changed to signal an error if the assumptions are not true.

3.25 CONCLUDING REMARKS

Three different instruction set architectures — ARM, Motorola 68000, and Intel IA-32 — were discussed in this chapter. The ARM and 68000 instruction sets provide basic illustrations of the RISC and CISC design styles, respectively. The IA-32 instruction set is a very extensive CISC design.

Any instruction in the ARM instruction set is encoded into one 32-bit word. Operations on data can only be performed when the data are contained in processor registers. Load and Store instructions are used to transfer operands between the processor registers and the memory. We explained how conditional execution of all instructions and extensive options for shifting operands in most instructions lead to efficient implementation of common programming tasks.

The 68000 is a traditional example of an instruction set whose instructions are encoded into a variable number of memory words, depending on the complexity of the operation to be performed and the amount of information needed to generate the effective addresses of any required memory operands. Both register and memory operands can be referenced in individual instructions.

The Intel IA-32 instruction set was seen to have extensive facilities for a broad range of operations on different types of data.

PROBLEMS

PART I: ARM

3.1 Assume the following register and memory contents in an ARM computer:

Register R0 contains 1000.
Register R1 contains 2000.
Register R2 contains 1016.
Register R6 contains 20.
Register R7 contains 30.

The numbers 1, 2, 3, 4, 5, and 6, are stored in successive word locations starting at memory address 1000. What is the effect of executing each of the following three short instruction blocks, starting each time from the given initial values?

```
(a) LDR    R8,[R0]
    LDR    R9,[R0,#4]
    ADD    R10,R8,R9
```

```
(b) STR     R6,[R1,#-4]!
    STR     R7,[R1,#-4]!
    LDR     R8,[R1],#4
    LDR     R9,[R1],#4
    SUB     R10,R8,R9
(c) LDMIA   R2!,{R4,R5}
    ADD     R4,R4,R5
```

3.2 Which of the following ARM instructions would cause the assembler to issue a syntax error message? Why?

```
(a) ADD   R2,R2,R2
(b) SUB   R0,R1,[R2,#4]
(c) MOV   R0,#2_1010101
(d) MOV   R0,#257
(e) ADD   R0,R1,R11,LSL #8
```

3.3 When a byte is loaded from memory into an ARM processor register using the Load instruction, the high-order 24 bits are cleared to 0s. (See Section 3.1.2.) If the loaded byte represents an 8-bit signed integer in 2's-complement representation, it must be sign-extended to 32 bits in the register before it can be used in arithmetic operations. Assuming such a byte has been loaded into register R0, write a short routine to sign-extend it to the 32-bit register length. (Hint: Use MOV instructions to move the contents of R0 back into R0 after appropriate shifts from the possibilities LSL, LSR, and ASR, as described in Section 2.10.2 and shown in Figure 2.30.)

3.4 Write an ARM program to reverse the order of bits in register R2. For example, if the starting pattern in R2 is 1110 . . . 0100, the result left in R2 should be 0010 . . . 0111. (Hint: Use shift and rotate operations.)

3.5 A *program trace* is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers R0, R1, and R2 after each of the first three executions of the BGT instruction in the program in Figure 3.7. Present the results in a table that has the three registers as column headers. Use three rows to list the contents of the registers after each execution of the BGT instruction. The program data are as given in Figure 3.8.

3.6 Write an ARM program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.7 An ARM program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at

STRING to determine whether or not it contains a contiguous substring identical to the string that begins at SUBSTRING. The length parameters n and m, where $n > m$, are stored in memory locations N and M, respectively. If a matching substring is found, the address of its first byte is to be stored in register R0; otherwise, the contents of R0 are to be cleared to 0. The program does not need to determine multiple occurrences of the substring. Only the address of the first matching substring is required.

3.8 Write an ARM program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for $n = 8$, the series is

$$0, 1, 1, 2, 3, 5, 8, 13$$

Your program should store the numbers in successive memory word locations starting at MEMLOC. Assume that the value n is stored in location N.

3.9 Write an ARM program to convert a word of text from lowercase to uppercase. The word consists of ASCII characters stored in successive byte locations in the memory, starting at location WORD and ending with a space character. (See Appendix E for the ASCII code.)

3.10 The list of student marks shown in Figure 2.14 is changed to contain j test scores for each student. Assume that there are n students. Write an ARM program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM + 4, SUM + 8, ... The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that j is stored in memory location J, placed ahead of location N.

3.11 Consider an array of numbers $A(i,j)$, where $i = 0$ through $n - 1$ is the row index and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of an ARM computer one row after another, with elements of each row occupying m successive word locations. Write an ARM subroutine for adding column x to column y, element by element, leaving the sum elements in column y. The indices x and y are passed to the subroutine in registers R1 and R2. The parameters n and m are passed to the subroutine in registers R3 and R4, and the address of element A(0,0) is passed in register R0. Any of the addressing modes in Table 3.1 can be used.

3.12 Write an ARM program that reads n characters from a keyboard and echoes them back to a display after pushing them onto a user stack as they are read. Use register R6 as the stack pointer. The count value n is contained in memory word location N.

3.13 Assume that the average time taken to fetch and execute an instruction in the program in Figure 3.9 is 20 nanoseconds. If keyboard characters are entered at the rate of 10 per second, approximately how many times is the BEQ READ instruction executed per character entered? Assume that the time taken to display each character is much less than the time between the entry of successive characters at the keyboard.

3.14 In the ARM program in Figure 3.9, "in-line" code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The address INSTATUS is passed to GETCHAR in register R1, and the main program expects to get the character passed back in register R3. The address OUTSTATUS and the character to be displayed are passed to PUTCHAR in registers R2 and R3, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using a stack whose pointer is register R13. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.15 Repeat Problem 3.14 using the stack to pass parameters.

3.16 Write an ARM program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of words are stored in the memory. Each table has 10 entries. The first table, starting at word location TENS, contains the binary representations for the decimal values 0, 10, 20, ... , 90. The second table starts at word location HUNDREDS and contains the decimal values 0, 100, 200, ... , 900 in binary representation.

3.17 The decimal-to-binary conversion program of Problem 3.16 is to be implemented using two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the stack whose pointer register is R13. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location where the converted binary value is to be stored. The first subroutine reads in the three characters from the keyboard and then calls the second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the stack.

(*a*) Write the two subroutines for the ARM processor.

(*b*) Give the contents of the stack immediately after the execution of the instruction that calls the second subroutine.

3.18 Consider the queue structure described in Problem 2.18. Write ARM APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.19 Using the format for presenting results that is described in Problem 3.5, give a program trace for the byte-sorting program in Figure 3.15*b*. Show the contents of registers R0, R2, and R3, and list byte locations LIST, LIST + 1, ... , LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000 and that the initial list of byte values is 120, 13, 106, 45, and 67, where [LIST] = 120.

3.20 Rewrite the byte-sorting program in Figure 3.15*b* as a subroutine that sorts a list of 32-bit positive integers. The calling program should pass the list address to the subroutine. The first 32-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.21 Consider the byte-sorting program of Figure 3.15*b*. During each pass through a sublist, LIST(j) through LIST(0), list entries are swapped whenever LIST(k) > LIST(j). An alternative strategy is to keep track of the address of the largest value in the sublist and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.22 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an ARM program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.

3.23 The linked-list insertion subroutine in Figure 3.16 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register R10 if this occurs or to return a zero if the insertion is successful.

3.24 The linked-list deletion subroutine in Figure 3.17 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.

PART II: 68000

3.25 Consider the following state of the 68000 processor:

Register D0 contains \$1000.
Register A0 contains \$2000.
Register A1 contains \$1000.
Memory location \$1000 contains the long word \$2000.
Memory location \$2000 contains the long word \$3000.

What is the effect of executing each of the following three instructions, starting each time from this initial state? How many bytes does each instruction occupy? How many memory accesses does the fetching and execution of each instruction require?

```
(a) ADD.L   D0,(A0)
(b) ADD.L   (A1,D0),D0
(c) ADD.L   #$2000,(A0)
```

3.26 Find the syntax errors in the following 68000 instructions:

(*a*)	ADDX	−(A2),D3
(*b*)	LSR.L	#9,D2
(*c*)	MOVE.B	520(A2,D2)
(*d*)	SUBA.L	12(A2,PC),A0
(*e*)	CMP.B	#254,$12(A2,D1.B)

3.27 A *program trace* is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers D0, D1, and A2 and memory locations N, NUM1, and SUM after each of five executions of the ADD.W instruction and after execution of the last MOVE.L instruction in the program in Figure 3.25. Present the results in a table that has the registers and memory locations as column headers. Use six rows to list the contents of the registers and memory locations after execution of each of the instructions. Assume the following initial values: [SUM] = 0, [N] = 5, NUM1 = 2400. and the five numbers are 83, 45, 156, −250, and 100.

3.28 Consider the following 68000 program:

```
       MOVEA.L  MEM1,A0
       MOVEA.L  MEM2,A2
       ADDA.L   A0,A1
       MOVEA.L  A0,A2
       MOVE.B   (A0) +,D0
LOOP   CMP.B    (A0) +,D0
       BLE      NXT
       LEA      −1(A0),A2
       MOVE.B   (A2),D0
NXT    CMPA.L   A0,A1
       BGT      LOOP
       MOVE.L   A2,DESIRED
```

(*a*) What does this program do?

(*b*) How many 16-bit words are needed to store this program in the memory?

(*c*) Give an expression for the number of memory accesses required. The expression should be of the form $T = a + bn + cm$, where n is the number of times the loop is executed, m is the number of times the branch to NXT is not taken, and a, b, and c are constants.

3.29 Consider the two 68000 programs given in Figure P3.1.

(*a*) Do these programs leave the same value in location RSLT?

(*b*) What task(s) do they accomplish?

Program 1			Program 2		
	CLR.L	D0		MOVE.W	#$FFFF,D0
	MOVEA.L	#LIST,A0		MOVEA.L	#LIST,A0
LOOP	MOVE.W	(A0)+,D1	LOOP	LSL.W	(A0)+
	BGE	LOOP		BCC	LOOP
	ADDQ.L	#1,D0		LSL.W	#1,D0
	CMPI	#17,D0		BCS	LOOP
	BLT	LOOP		MOVE.W	−2(A0),RSLT
	MOVE.W	−2(A0),RSLT			

Figure P3.1 Two 68000 programs for Problem 3.29.

(*c*) How many bytes of memory are needed to store each program?

(*d*) Which program requires the larger number of memory accesses?

(*e*) What are the advantages and disadvantages of these programs?

3.30 Write a 68000 program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.31 A 68000 program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another, shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at STRING to determine whether or not it contains a contiguous substring identical to the string that begins at SUBSTRING. The length parameters n and m, where $n > m$, are stored in memory locations N and M, respectively. If a matching substring is found, the address of its first byte is to be stored in register D0; otherwise, the contents of D0 are to be cleared to 0. The program does not need to determine multiple occurrences of the substring. Only the address of the first matching substring is required.

3.32 Write a 68000 program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for $n = 8$, the series is

$$0, 1, 1, 2, 3, 5, 8, 13$$

Your program should store the numbers in memory byte locations starting at MEMLOC. Assume that the value n is stored in location N. What is the largest n that your program can handle?

3.33 Write a 68000 program to convert a word of text from lowercase to uppercase. The word consists of ASCII characters stored in successive byte locations in the memory, starting at location WORD and ending with a space character. (See Appendix E for the ASCII code.)

3.34 The list of student marks shown in Figure 2.14 is changed to contain j test scores for each student. Each entry in the list is a 16-bit word, so the increments on LIST are by 2. Assume that there are n students. Write a 68000 program for computing the sums of the scores on each test and store these sums in the memory word locations at addresses SUM, SUM + 2, SUM + 4, ... The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that j is stored in memory location J, placed ahead of N.

3.35 Write a 68000 program that reads n characters from a keyboard and echoes them back to a display after pushing them onto a user stack as they are read. Use register A0 as the stack pointer. The count value n is stored in memory word location N.

3.36 Assume that the average time taken to fetch and execute an instruction in the program in Figure 3.27 is 20 nanoseconds. If keyboard characters are entered at the rate of 10 per second, approximately how many times is the BEQ READ instruction executed per character entered? Assume that the time taken to display each character is much less than the time between the entry of successive characters at the keyboard.

3.37 In the 68000 program in Figure 3.27, "in-line" code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The addresses INSTATUS and DATAIN are passed to GETCHAR in registers A0 and A1; and the main program expects to get the character passed back in register D0. The addresses OUTSTATUS and DATAOUT and the character to be displayed are passed to PUTCHAR in registers A2, A3, and D0, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using the processor stack whose pointer is register A7. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.38 Repeat problem 3.37 using the stack to pass parameters.

3.39 Consider the queue structure described in Problem 2.18. Write 68000 APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.40 Write a 68000 program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of words are stored in the memory. Each table has 10 entries. The first table, starting at word location TENS, contains the binary representations for the decimal values 0, 10, 20, ..., 90. The second table starts at word location HUNDREDS and contains the decimal values 0, 100, 200, ..., 900 in binary representation.

3.41 The decimal-to-binary conversion program of Problem 3.40 is to be implemented as two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the processor stack. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location for the converted binary value. The first subroutine reads in the three characters from the keyboard and then calls the second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the processor stack.

(*a*) Write the two subroutines for the 68000 processor.

(*b*) Give the contents of the processor stack immediately after the execution of the instruction that calls the second subroutine.

3.42 Consider an array of 16-bit numbers A(i,j), where $i = 0$ through $n - 1$ is the row index and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of a 68000 computer one row after another, with elements of each row occupying m successive word locations. Write a 68000 subroutine for adding column x to column y, element by element, leaving the sum elements in column y. The indices x and y are passed to the subroutine in registers D1 and D2. The parameters n and m are passed to the subroutine in registers D3 and D4, and the address of element A(0,0) is passed in register A0. Any of the addressing modes in Table 3.2 can be used.

3.43 Write a 68000 program to reverse the order of bits in register D2. For example, if the starting pattern in D2 is 1110 ... 0100, the result left in D2 should be 0010 ... 0111. (Hint: Use shift and rotate operations.)

3.44 How many bytes of memory are needed to store the program in Figure 3.32? How many memory accesses take place during execution of this program?

3.45 Using the format for presenting results that is described in Problem 3.27, give a program trace for the byte-sorting program in Figure 3.34*b*. Show the contents of registers D1, D2, and D3, and the list byte locations LIST, LIST + 1, ..., LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000, and that the initial list of byte values is 120, 13, 106, 45, and 67, where [LIST] = 120.

3.46 Rewrite the byte-sorting program in Figure 3.34*b* as a subroutine that sorts a list of 16-bit positive integers. The calling program should pass the list address to the subroutine. The first 16-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.47 Consider the byte-sorting program of Figure 3.34*b*. During each pass through a sublist, LIST(j) through LIST(0), list entries are swapped whenever LIST(k) > LIST(j). An alternative strategy is to keep track of the address of the largest value in the sublist and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.48 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write a 68000 program that accomplishes the

same thing as the program in Figure 2.15. The head record is stored at memory location 1000. Assume that all list entries are long words.

3.49 The linked-list insertion subroutine in Figure 3.35 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register A6 if this occurs, or return a zero if the insertion is successful.

3.50 The linked-list deletion subroutine in Figure 3.36 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.

PART III: Intel IA-32

3.51 Assume the following register and memory contents in an IA-32 computer:

Register EBX contains 1000.

Register ESI contains 2.

The numbers 1, 2, 3, 4, 5, and 6, are stored in successive doubleword locations starting at memory address 1000.

The address label LOC represents address 1008.

What is the effect of executing each of the following three short instruction blocks, starting each time from the given initial values?

```
(a) MOV   EAX,10
    ADD   EAX,[EBX + ESI*4 + 8]
(b) PUSH  20
    PUSH  30
    POP   EAX
    POP   EBX
    SUB   EAX,EBX
(c) LEA   EAX,LOC
    MOV   EBX,[EAX]
```

3.52 Which of the following IA-32 instructions would cause the assembler to issue a syntax error message? Why?

```
(a) ADD   EAX,EAX
(b) ADD   [EAX],[EBX + 4]
(c) SUB   EAX,[EBX + ESI*4 + 20]
(d) SUB   EAX,[EBX + ESI*10]
(e) ADD   EAX,-31728542
(f) MOV   20,EAX
(g) MOV   EAX,[EBP + ESP*4]
```

3.53 A *program trace* is a listing of the contents of certain registers and memory locations at different times during the execution of a program. List the contents of registers EAX, EBX, and ECX after each of the first three executions of the LOOP instruction in the program in Figure 3.40*b*. Present the results in a table that has the three registers as column headers. Use three rows to list the contents of the registers after each execution of the LOOP instruction. The program data is as given in Figure 3.42.

3.54 Write an IA-32 program that compares the corresponding bytes of two lists of bytes and places the larger byte in a third list. The two lists start at byte locations X and Y, and the larger-byte list starts at LARGER. The length of the lists is stored in memory location N.

3.55 An IA-32 program is required for the following character manipulation task: A string of n characters is stored in the memory in consecutive byte locations, beginning at location STRING. Another, shorter string of m characters is stored in consecutive byte locations, beginning at location SUBSTRING. The program must search the string that begins at STRING to determine whether or not it contains a contiguous substring identical to the string that begins at SUBSTRING. The length parameters n and m, where $n > m$, are stored in memory locations N and M, respectively. If a matching substring is found, the address of its first byte is to be stored in register EAX; otherwise, the contents of EAX are to be cleared to 0. The program does not need to determine multiple occurrences of the substring. Only the address of the first matching substring is required.

3.56 Write an IA-32 program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for $n = 8$, the series is

$$0, 1, 1, 2, 3, 5, 8, 13$$

Your program should store the numbers in successive memory doubleword locations starting at MEMLOC. Assume that the value n is stored in location N.

3.57 Write an IA-32 program to convert a word of text from lowercase to uppercase. The word consists of ASCII characters stored in successive byte locations in the memory, starting at location WORD and ending with a space character. (See Appendix E for the ASCII code.)

3.58 The list of student marks shown in Figure 2.14 is changed to contain j test scores for each student. Assume that there are n students. Write an IA-32 program for computing the sums of the scores on each test and store these sums in the memory doubleword locations at addresses SUM, SUM + 4, SUM + 8, ... The number of tests, j, is larger than the number of registers in the processor, so the type of program shown in Figure 2.15 for the 3-test case cannot be used. Use two nested loops, as suggested in Section 2.5.3. The inner loop should accumulate the sum for a particular test, and the outer loop should run over the number of tests, j. Assume that j is stored in memory location J, placed ahead of location N.

3.59 Write an IA-32 program to reverse the order of bits in register EAX. For example, if the starting pattern in EAX is 1110 ... 0100, the result left in EAX should be 0010 ... 0111. (Hint: Use shift and rotate operations.)

3.60 Consider the queue structure described in Problem 2.18. Write IA-32 APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

3.61 Write an IA-32 program that reads n characters from a keyboard and echoes them back to a display after pushing them onto a user stack as they are read. Use register EBX as the stack pointer. The count value n is stored in memory doubleword location N.

3.62 Assume that the average time taken to fetch and execute an instruction in the program in Figure 3.44 is 10 nanoseconds. If keyboard characters are entered at the rate of 10 per second, approximately how many times is the JNC READ instruction executed per character entered? Assume that the time taken to display each character is much less than the time between the entry of successive characters at the keyboard.

3.63 In the IA-32 program in Figure 3.44, "in-line" code is used to read a line of characters and display them. Rewrite this program in the form of a main program that calls a subroutine named GETCHAR to read a single character and calls another subroutine named PUTCHAR to display a single character. The addresses INSTATUS and DATAIN are passed to GETCHAR in registers EBX and EDX; and the main program expects to get the character passed back in register AL. The addresses OUTSTATUS and DATAOUT and the character to be displayed are passed to PUTCHAR in registers ESI, EDI, and AL, respectively. Any other registers used by either subroutine must be saved and restored by the subroutine using the processor stack, whose pointer is register ESP. Storing the characters in memory and checking for the end-of-line character CR is to be done in the main program.

3.64 Repeat problem 3.63, passing parameters on the processor stack.

3.65 Write an IA-32 program to accept three decimal digits from a keyboard. Each digit is represented in the ASCII code (see Appendix E). Assume that these three digits represent a decimal integer in the range 0 to 999 and convert the integer into a binary number representation. The high-order digit is received first. To aid in this conversion, two tables of doublewords are stored in the memory. Each table has 10 entries. The first table, starting at doubleword location TENS, contains the binary representations for the decimal values 0, 10, 20, . . . , 90. The second table starts at doubleword location HUNDREDS and contains the decimal values 0, 100, 200, . . . , 900 in binary representation.

3.66 The decimal-to-binary conversion program of Problem 3.65 is to be implemented using two nested subroutines. The main program that calls the first subroutine passes two parameters by pushing them onto the processor stack. The first parameter is the address of a 3-byte memory buffer area for storing the input decimal-digit characters. The second parameter is the address of the location for the converted binary value. The first subroutine reads in the three characters from the keyboard and then calls the

second subroutine to perform the conversion. The necessary parameters are passed to this subroutine via the processor registers. Both subroutines must save the contents of any registers that they use on the processor stack.

(*a*) Write the two subroutines for the IA-32 processor.

(*b*) Give the contents of the processor stack immediately after the execution of the instruction that calls the second subroutine.

3.67 Consider an array of numbers A(i,j), where $i = 0$ through $n - 1$ is the row index and $j = 0$ through $m - 1$ is the column index. The array is stored in the memory of a IA-32 computer one row after another, with elements of each row occupying m successive doubleword locations. Write an IA-32 subroutine for adding column x to column y, element by element, leaving the sum elements in column y. The indices x and y are passed to the subroutine in registers ESI and EDI. The parameters n and m are passed to the subroutine in registers EAX and EBX, and the address of element A(0,0) is passed in register EDX. Any of the addressing modes in Table 3.3 can be used.

3.68 Using the format for presenting results that is described in Problem 3.53, give a program trace for the byte-sorting program in Figure 3.50*b*. Show the contents of registers EDI, ECX, and DL, and list byte locations LIST, LIST + 1, . . . , LIST + 4 for a 5-byte list after each execution of the last instruction in the program. Assume that LIST = 1000 and that the initial list of byte values is 120, 13, 106, 45, and 67, where [LIST] = 120.

3.69 Rewrite the byte-sorting program in Figure 3.50*b* as a subroutine that sorts a list of 32-bit positive integers. The calling program should pass the list address to the subroutine. The first 32-bit quantity at that location is the number of entries in the list, followed by the numbers to be sorted.

3.70 Consider the byte-sorting program of Figure 3.50*b*. During each pass through a sublist, LIST(j) through LIST(0), list entries are swapped whenever LIST(k) > LIST(j). An alternative strategy is to keep track of the address of the largest value in the sublist, and to perform, at most, one swap at the end of the sublist search. Rewrite the program using this approach. What is the advantage of this approach?

3.71 Assume that the list of student test scores shown in Figure 2.14 is stored in the memory as a linked list as shown in Figure 2.36. Write an IA-32 program that accomplishes the same thing as the program in Figure 2.15. The head record is stored at memory location 1000.

3.72 The linked-list insertion subroutine in Figure 3.51 does not check if the ID of the new record matches that of a record already in the list. What happens in the execution of the subroutine if this is the case? Modify the subroutine to return the address of the matching record in register EDX if this occurs or to return a zero if the insertion is successful.

3.73 The linked-list deletion subroutine in Figure 3.52 assumes that a record with the ID contained in register RIDNUM is in the list. What happens in the execution of the subroutine if there is no record with this ID? Modify the subroutine to return a zero in RIDNUM if deletion is successful, or leave RIDNUM unchanged if the record is not in the list.

REFERENCES

1. D. Jaggar, "ARM Architecture and Systems," *IEEE Micro*, 17(4): 9–11, July/August 1997.
2. S. Furber, *ARM System-on-Chip Architecture*, Addison-Wesley, Harlow, England, 2000.
3. A. Clements, *The Principles of Computer Hardware*, 3rd ed., Oxford University Press, New York, 2000.
4. A. van Someren and C. Atack, *The ARM RISC Chip — A Programmer's Guide*, Addison-Wesley, Wokingham, England, 1994.
5. http://www.arm.com
6. http://www.motorola.com
7. D. Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, 1991.
8. http://www.intel.com
9. B.B. Brey, *The Intel Microprocessors*, 5th ed., Prentice-Hall, Upper Saddle River, New Jersey, 2000.
10. S.P. Dandamudi, *Introduction to Assembly Language Programming — From 8086 to Pentium Processors*, Springer-Verlag, New York, 1998.