```
        LDA     ZERO        INITIALIZE INDEX VALUE TO 0
        STA     INDEX
ADDLP   LDX     INDEX       LOAD INDEX VALUE INTO REGISTER X
        LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        LDA     INDEX       ADD 3 TO INDEX VALUE
        ADD     THREE
        STA     INDEX
        COMP    K300        COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX IS LESS THAN 300
        .
        .
        .
INDEX   RESW    1           ONE-WORD VARIABLE FOR INDEX VALUE
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
.                           ONE-WORD CONSTANTS
ZERO    WORD    0
K300    WORD    300
THREE   WORD    3
```

(a)

```
        LDS     #3          INITIALIZE REGISTER S TO 3
        LDT     #300        INITIALIZE REGISTER T TO 300
        LDX     #0          INITIALIZE INDEX REGISTER TO 0
ADDLP   LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        ADDR    S,X         ADD 3 TO INDEX VALUE
        COMPR   X,T         COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX VALUE IS LESS THAN 300
        .
        .
        .
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
```

(b)

Figure 1.5 Sample indexing and looping operations for (a) SIC and (b) SIC/XE.

ALPHA and BETA, storing the results in the elements of GAMMA. The general principles of looping and indexing are the same as previously discussed. However, the value in the index register must be incremented by 3 for each iteration of this loop, because each iteration processes a 3-byte (i.e., one-word) element of the arrays. The TIX instruction always adds 1 to register X, so it is not suitable for this program fragment. Instead, we use arithmetic and comparison instructions to handle the index value.

In Fig. 1.5(a), we define a variable INDEX that holds the value to be used for indexing for each iteration of the loop. Thus, INDEX should be 0 for the first iteration, 3 for the second, and so on. INDEX is initialized to 0 before the start of the loop. The first instruction in the body of the loop loads the current value of INDEX into register X so that it can be used for target address calculation. The next three instructions in the loop load a word from ALPHA, add the corresponding word from BETA, and store the result in the corresponding word of GAMMA. The value of INDEX is then loaded into register A, incremented by 3, and stored back into INDEX. After being stored, the new value of INDEX is still present in register A. This value is then compared to 300 (the length of the arrays in bytes) to determine whether or not to terminate the loop. If the value of INDEX is less than 300, then all bytes of the arrays have not yet been processed. In that case, the JLT instruction causes a jump back to the beginning of the loop, where the new value of INDEX is loaded into register X.

This particular loop is cumbersome on SIC, because register A must be used for adding the array elements together and also for incrementing the index value. The loop can be written much more efficiently for SIC/XE, as shown in Fig. 1.5(b). In this example, the index value is kept permanently in register X. The amount by which to increment the index value (3) is kept in register S, and the register-to-register ADDR instruction is used to add this increment to register X. Similarly, the value 300 is kept in register T, and the instruction COMPR is used to compare registers X and T in order to decide when to terminate the loop.

Figure 1.6 shows a simple example of input and output on SIC; the same instructions would also work on SIC/XE. (The more advanced input and output facilities available on SIC/XE, such as I/O channels and interrupts, are discussed in Chapter 6.) This program fragment reads 1 byte of data from device F1 and copies it to device 05. The actual input of data is performed using the RD (Read Data) instruction. The operand for the RD is a byte in memory that contains the hexadecimal code for the input device (in this case, F1). Executing the RD instruction transfers 1 byte of data from this device into the rightmost byte of register A. If the input device is character-oriented (for example, a keyboard), the value placed in register A is the ASCII code for the character that was read.

The READ subroutine itself consists of a loop. Each execution of this loop reads 1 byte of data from the input device, using the same techniques illustrated in Fig. 1.6. The bytes of data that are read are stored in a 100-byte buffer area labeled RECORD. The indexing and looping techniques that are used in storing characters in this buffer are essentially the same as those illustrated in Fig. 1.4(a).

Figure 1.7(b) shows the same READ subroutine as it might be written for SIC/XE. The main differences from Fig. 1.7(a) are the use of immediate addressing and the TIXR instruction, as was illustrated in Fig. 1.4(a).

1.4 TRADITIONAL (CISC) MACHINES

This section introduces the architectures of two of the machines that will be used as examples later in the text. Section 1.4.1 describes the VAX architecture, and Section 1.4.2 describes the architecture of the Intel x86 family of processors.

The machines described in this section are classified as Complex Instruction Set Computers (CISC). CISC machines generally have a relatively large and complicated instruction set, several different instruction formats and lengths, and many different addressing modes. Thus the implementation of such an architecture in hardware tends to be complex.

You may want to compare the examples in this section with the Reduced Instruction Set Computer (RISC) examples in Section 1.5. Further discussion of CISC versus RISC designs can be found in Tabak (1995).

1.4.1 VAX Architecture

The VAX family of computers was introduced by Digital Equipment Corporation (DEC) in 1978. The VAX architecture was designed for compatibility with the earlier PDP-11 machines. A compatibility mode was provided at the hardware level so that many PDP-11 programs could run unchanged on the VAX. It was even possible for PDP-11 programs and VAX programs to share the same machine in a multi-user environment.

This section summarizes some of the main characteristics of the VAX architecture. For further information, see Baase (1992).

Memory

The VAX memory consists of 8-bit bytes. All addresses used are byte addresses. Two consecutive bytes form a *word;* four bytes form a *longword;* eight bytes form a *quadword;* sixteen bytes form an *octaword.* Some operations

are more efficient when operands are aligned in a particular way—for example, a longword operand that begins at a byte address that is a multiple of 4.

All VAX programs operate in a *virtual address space* of 2^{32} bytes. This virtual memory allows programs to operate as though they had access to an extremely large memory, regardless of the amount of memory actually present on the system. Routines in the operating system take care of the details of memory management. We discuss virtual memory in connection with our study of operating systems in Chapter 6. One half of the VAX virtual address space is called *system space*, which contains the operating system, and is shared by all programs. The other half of the address space is called *process space*, and is defined separately for each program. A part of the process space contains stacks that are available to the program. Special registers and machine instructions aid in the use of these stacks.

Registers

There are 16 general-purpose registers on the VAX, denoted by R0 through R15. Some of these registers, however, have special names and uses. All general registers are 32 bits in length. Register R15 is the *program counter*, also called PC. It is updated during instruction execution to point to the next instruction byte to be fetched. R14 is the *stack pointer* SP, which points to the current top of the stack in the program's process space. Although it is possible to use other registers for this purpose, hardware instructions that implicitly use the stack always use SP. R13 is the *frame pointer* FP. VAX procedure call conventions build a data structure called a stack frame, and place its address in FP. R12 is the *argument pointer* AP. The procedure call convention uses AP to pass a list of arguments associated with the call.

Registers R6 through R11 have no special functions, and are available for general use by the program. Registers R0 through R5 are likewise available for general use; however, these registers are also used by some machine instructions.

In addition to the general registers, there is a *processor status longword* (PSL), which contains state variables and flags associated with a process. The PSL includes, among many other items of information, a condition code and a flag that specifies whether PDP-11 compatibility mode is being used by a process. There are also a number of control registers that are used to support various operating system functions.

Data Formats

Integers are stored as binary numbers in a byte, word, longword, quadword, or octaword; 2's complement representation is used for negative values. Characters are stored using their 8-bit ASCII codes.

```
INLOOP    TD      INDEV       TEST INPUT DEVICE
          JEQ     INLOOP      LOOP UNTIL DEVICE IS READY
          RD      INDEV       READ ONE BYTE INTO REGISTER A
          STCH    DATA        STORE BYTE THAT WAS READ
          .
          .
          .
OUTLP     TD      OUTDEV      TEST OUTPUT DEVICE
          JEQ     OUTLP       LOOP UNTIL DEVICE IS READY
          LDCH    DATA        LOAD DATA BYTE INTO REGISTER A
          WD      OUTDEV      WRITE ONE BYTE TO OUTPUT DEVICE
          .
          .
          .
INDEV     BYTE    X'F1'       INPUT DEVICE NUMBER
OUTDEV    BYTE    X'05'       OUTPUT DEVICE NUMBER
DATA      RESB    1           ONE-BYTE VARIABLE
```

Figure 1.6 Sample input and output operations for SIC.

Before the RD can be executed, however, the input device must be ready to transmit the data. For example, if the input device is a keyboard, the operator must have typed a character. The program checks for this by using the TD (Test Device) instruction. When the TD is executed, the status of the addressed device is tested and the condition code is set to indicate the result of this test. If the device is ready to transmit data, the condition code is set to "less than"; if the device is not ready, the condition code is set to "equal." As Fig. 1.6 illustrates, the program must execute the TD instruction and then check the condition code by using a conditional jump. If the condition code is "equal" (device not ready), the program jumps back to the TD instruction. This two-instruction loop will continue until the device becomes ready; then the RD will be executed.

Output is performed in the same way. First the program uses TD to check whether the output device is ready to receive a byte of data. Then the byte to be written is loaded into the rightmost byte of register A, and the WD (Write Data) instruction is used to transmit it to the device.

Figure 1.7 shows how these instructions can be used to read a 100-byte record from an input device into memory. The read operation in this example is placed in a subroutine. This subroutine is called from the main program by using the JSUB (Jump to Subroutine) instruction. At the end of the subroutine there is an RSUB (Return from Subroutine) instruction, which returns control to the instruction that follows the JSUB.

```
            JSUB    READ            CALL READ SUBROUTINE
            .
            .
            .
.                                   SUBROUTINE TO READ 100-BYTE RECORD
READ        LDX     ZERO            INITIALIZE INDEX REGISTER TO 0
RLOOP       TD      INDEV           TEST INPUT DEVICE
            JEQ     RLOOP           LOOP IF DEVICE IS BUSY
            RD      INDEV           READ ONE BYTE INTO REGISTER A
            STCH    RECORD,X        STORE DATA BYTE INTO RECORD
            TIX     K100            ADD 1 TO INDEX AND COMPARE TO 100
            JLT     RLOOP           LOOP IF INDEX IS LESS THAN 100
            RSUB                    EXIT FROM SUBROUTINE
            .
            .
            .
INDEV       BYTE    X'F1'           INPUT DEVICE NUMBER
RECORD      RESB    100             100-BYTE BUFFER FOR INPUT RECORD
.                                   ONE-WORD CONSTANTS
ZERO        WORD    0
K100        WORD    100
```

(a)

```
            JSUB    READ            CALL READ SUBROUTINE
            .
            .
            .
.                                   SUBROUTINE TO READ 100-BYTE RECORD
READ        LDX     #0              INITIALIZE INDEX REGISTER TO 0
            LDT     #100            INITIALIZE REGISTER T TO 100
RLOOP       TD      INDEV           TEST INPUT DEVICE
            JEQ     RLOOP           LOOP IF DEVICE IS BUSY
            RD      INDEV           READ ONE BYTE INTO REGISTER A
            STCH    RECORD,X        STORE DATA BYTE INTO RECORD
            TIXR    T               ADD 1 TO INDEX AND COMPARE TO 100
            JLT     RLOOP           LOOP IF INDEX IS LESS THAN 100
            RSUB                    EXIT FROM SUBROUTINE
            .
            .
            .
INDEV       BYTE    X'F1'           INPUT DEVICE NUMBER
RECORD      RESB    100             100-BYTE BUFFER FOR INPUT RECORD
```

(b)

Figure 1.7 Sample subroutine call and record input operations for (a) SIC and (b) SIC/XE.

There are four different floating-point data formats on the VAX, ranging in length from 4 to 16 bytes. Two of these are compatible with those found on the PDP-11, and are standard on all VAX processors. The other two are available as options, and provide for an extended range of values by allowing more bits in the exponent field. In each case, the principles are the same as those we discussed for SIC/XE: a floating-point value is represented as a fraction that is to be multiplied by a specified power of 2.

VAX processors provide a *packed decimal* data format. In this format, each byte represents two decimal digits, with each digit encoded using 4 bits of the byte. The sign is encoded in the last 4 bits. There is also a *numeric* format that is used to represent numeric values with one digit per byte. In this format, the sign may appear either in the last byte, or as a separate byte preceding the first digit. These two variations are called *trailing numeric* and *leading separate numeric.*

VAX also supports queues and variable-length bit strings. Data structures such as these can, of course, be implemented on any machine; however, VAX provides direct hardware support for them. There are single machine instructions that insert and remove entries in queues, and perform a variety of operations on bit strings. The existence of such powerful machine instructions and complex primitive data types is one of the more unusual features of the VAX architecture.

Instruction Formats

VAX machine instructions use a variable-length instruction format. Each instruction consists of an operation code (1 or 2 bytes) followed by up to six *operand specifiers*, depending on the type of instruction. Each operand specifier designates one of the VAX addressing modes and gives any additional information necessary to locate the operand. (See the description of addressing modes in the following section for further information.)

Addressing Modes

VAX provides a large number of addressing modes. With few exceptions, any of these addressing modes may be used with any instruction. The operand itself may be in a register (*register* mode), or its address may be specified by a register (*register deferred* mode). If the operand address is in a register, the register contents may be automatically incremented or decremented by the operand length (*autoincrement* and *autodecrement* modes). There are several base relative addressing modes, with displacement fields of different lengths; when used with register PC, these become program-counter relative modes. All of these addressing modes may also include an index register, and many of them are available in a form that specifies indirect addressing (called *deferred*

modes on VAX). In addition, there are immediate operands and several special-purpose addressing modes. For further details, see Baase (1992).

Instruction Set

One of the goals of the VAX designers was to produce an instruction set that is symmetric with respect to data type. Many instruction mnemonics are formed by combining the following elements:

1. A prefix that specifies the type of operation.
2. A suffix that specifies the data type of the operands.
3. A modifier (on some instructions) that gives the number of operands involved.

For example, the instruction ADDW2 is an add operation with two operands, each a word in length. Likewise, MULL3 is a multiply operation with three longword operands, and CVTWL specifies a conversion from word to longword. (In the latter case, a two-operand instruction is assumed.) For a typical instruction, operands may be located in registers, in memory, or in the instruction itself (immediate addressing). The same machine instruction code is used, regardless of operand locations.

VAX provides all of the usual types of instructions for computation, data movement and conversion, comparison, branching, etc. In addition, there are a number of operations that are much more complex than the machine instructions found on most computers. These operations are, for the most part, hardware realizations of frequently occurring sequences of code. They are implemented as single instructions for efficiency and speed. For example, VAX provides instructions to load and store multiple registers, and to manipulate queues and variable-length bit fields. There are also powerful instructions for calling and returning from procedures. A single instruction saves a designated set of registers, passes a list of arguments to the procedure, maintains the stack, frame, and argument pointers, and sets a mask to enable error traps for arithmetic operations. For further information on all of the VAX instructions, see Baase (1992).

Input and Output

Input and output on the VAX are accomplished by I/O device controllers. Each controller has a set of control/status and data registers, which are assigned locations in the physical address space. The portion of the address space into which the device controller registers are mapped is called *I/O space.*

is stored at the lowest-numbered address. (This is commonly called *little-endian* byte ordering, because the "little end" of the value comes first in memory.)

Integers can also be stored in *binary coded decimal* (BCD). In the unpacked BCD format, each byte represents one decimal digit. The value of this digit is encoded (in binary) in the low-order 4 bits of the byte; the high-order bits are normally zero. In the packed BCD format, each byte represents two decimal digits, with each digit encoded using 4 bits of the byte.

There are three different floating-point data formats. The single-precision format is 32 bits long. It stores 24 significant bits of the floating-point value, and allows for a 7-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.) The double-precision format is 64 bits long. It stores 53 significant bits, and allows for a 10-bit exponent. The extended-precision format is 80 bits long. It stores 64 significant bits, and allows for a 15-bit exponent.

Characters are stored one per byte, using their 8-bit ASCII codes. Strings may consist of bits, bytes, words, or doublewords; special instructions are provided to handle each type of string.

Instruction Formats

All of the x86 machine instructions use variations of the same basic format. This format begins with optional prefixes containing flags that modify the operation of the instruction. For example, some prefixes specify a repetition count for an instruction. Others specify a segment register that is to be used for addressing an operand (overriding the normal default assumptions made by the hardware). Following the prefixes (if any) is an opcode (1 or 2 bytes); some operations have different opcodes, each specifying a different variant of the operation. Following the opcode are a number of bytes that specify the operands and addressing modes to be used. (See the description of addressing modes in the next section for further information.)

The opcode is the only element that is always present in every instruction. Other elements may or may not be present, and may be of different lengths, depending on the operation and the operands involved. Thus, there are a large number of different potential instruction formats, varying in length from 1 byte to 10 bytes or more.

Addressing Modes

The x86 architecture provides a large number of addressing modes. An operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register* mode).

Operands stored in memory are often specified using variations of the general target address calculation

$$TA = (\text{base register}) + (\text{index register}) * (\text{scale factor}) + \text{displacement}$$

Any general-purpose register may be used as a base register; any general-purpose register except ESP can be used as an index register. The scale factor may have the value 1, 2, 4, or 8, and the displacement may be an 8-, 16-, or 32-bit value. The base and index register numbers, scale, and displacement are encoded as parts of the operand specifiers in the instruction. Various combinations of these items may be omitted, resulting in eight different addressing modes. The address of an operand in memory may also be specified as an absolute location (*direct* mode), or as a location relative to the EIP register (*relative* mode).

Instruction Set

The x86 architecture has a large and complex instruction set, containing more than 400 different machine instructions. An instruction may have zero, one, two, or three operands. There are register-to-register instructions, register-to-memory instructions, and a few memory-to-memory instructions. In some cases, operands may also be specified in the instruction as immediate values.

Most data movement and integer arithmetic instructions can use operands that are 1, 2, or 4 bytes long. String manipulation instructions, which use repetition prefixes, can deal directly with variable-length strings of bytes, words, or doublewords. There are many instructions that perform logical and bit manipulations, and support control of the processor and memory-management systems.

The x86 architecture also includes special-purpose instructions to perform operations frequently required in high-level programming languages—for example, entering and leaving procedures and checking subscript values against the bounds of an array.

Input and Output

Input is performed by instructions that transfer one byte, word, or doubleword at a time from an I/O port into register EAX. Output instructions transfer one byte, word, or doubleword from EAX to an I/O port. Repetition prefixes allow these instructions to transfer an entire string in a single operation.

No special instructions are required to access registers in I/O space. An I/O device driver issues commands to the device controller by storing values into the appropriate registers, exactly as if they were physical memory locations. Likewise, software routines may read these registers to obtain status information. The association of an address in I/O space with a physical register in a device controller is handled by the memory management routines.

1.4.2 Pentium Pro Architecture

The Pentium Pro microprocessor, introduced near the end of 1995, is the latest in the Intel x86 family. Other recent microprocessors in this family are the 80486 and Pentium. Processors of the x86 family are presently used in a majority of personal computers, and there is a vast amount of software for these processors. It is expected that additional generations of the x86 family will be developed in the future.

The various x86 processors differ in implementation details and operating speed. However, they share the same basic architecture. Each succeeding generation has been designed to be compatible with the earlier versions. This section contains an overview of the x86 architecture, which will serve as background for the examples to be discussed later in the book. Further information about the x86 family can be found in Intel (1995), Anderson and Shanley (1995), and Tabak (1995).

Memory

Memory in the x86 architecture can be described in at least two different ways. At the physical level, memory consists of 8-bit bytes. All addresses used are byte addresses. Two consecutive bytes form a *word;* four bytes form a *doubleword* (also called a dword). Some operations are more efficient when operands are aligned in a particular way—for example, a doubleword operand that begins at a byte address that is a multiple of 4.

However, programmers usually view the x86 memory as a collection of *segments*. From this point of view, an address consists of two parts—a segment number and an offset that points to a byte within the segment. Segments can be of different sizes, and are often used for different purposes. For example, some segments may contain executable instructions, and other segments may be used to store data. Some data segments may be treated as stacks that can be used to save register contents, pass parameters to subroutines, and for other purposes.

It is not necessary for all of the segments used by a program to be in physical memory. In some cases, a segment can also be divided into *pages*. Some of the pages of a segment may be in physical memory, while others may be stored on disk. When an x86 instruction is executed, the hardware and the

operating system make sure that the needed byte of the segment is loaded into physical memory. The segment/offset address specified by the programmer is automatically translated into a physical byte address by the x86 Memory Management Unit (MMU). Chapter 6 contains a brief discussion of methods that can be used in this kind of address translation.

Registers

There are eight general-purpose registers, which are named EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. Each general-purpose register is 32 bits long (i.e., one doubleword). Registers EAX, EBX, ECX, and EDX are generally used for data manipulation; it is possible to access individual words or bytes from these registers. The other four registers can also be used for data, but are more commonly used to hold addresses. The general-purpose register set is identical for all members of the x86 family beginning with the 80386. This set is also compatible with the more limited register sets found in earlier members of the family.

There are also several different types of special-purpose registers in the x86 architecture. EIP is a 32-bit register that contains a pointer to the next instruction to be executed. FLAGS is a 32-bit register that contains many different bit flags. Some of these flags indicate the status of the processor; others are used to record the results of comparisons and arithmetic operations. There are also six 16-bit *segment registers* that are used to locate segments in memory. Segment register CS contains the address of the currently executing code segment, and SS contains the address of the current stack segment. The other segment registers (DS, ES, FS, and GS) are used to indicate the addresses of data segments.

Floating-point computations are performed using a special *floating-point unit* (FPU). This unit contains eight 80-bit data registers and several other control and status registers.

All of the registers discussed so far are available to application programs. There are also a number of registers that are used only by system programs such as the operating system. Some of these registers are used by the MMU to translate segment addresses into physical addresses. Others are used to control the operation of the processor, or to support debugging operations.

Data Formats

The x86 architecture provides for the storage of integers, floating-point values, characters, and strings. Integers are normally stored as 8-, 16-, or 32-bit binary numbers. Both signed and unsigned integers (also called ordinals) are supported; 2's complement is used for negative values. The FPU can also handle 64-bit signed integers. In memory, the least significant part of a numeric value

1.5 RISC MACHINES

This section introduces the architectures of three RISC machines that will be used as examples later in the text. Section 1.5.1 describes the architecture of the SPARC family of processors. Section 1.5.2 describes the PowerPC family of microprocessors for personal computers. Section 1.5.3 describes the architecture of the Cray T3E supercomputing system.

All of these machines are examples of RISC (Reduced Instruction Set Computers), in contrast to traditional CISC (Complex Instruction Set Computer) implementations such as Pentium and VAX. The RISC concept, developed in the early 1980s, was intended to simplify the design of processors. This simplified design can result in faster and less expensive processor development, greater reliability, and faster instruction execution times.

In general, a RISC system is characterized by a standard, fixed instruction length (usually equal to one machine word), and single-cycle execution of most instructions. Memory access is usually done by load and store instructions only. All instructions except for load and store are register-to-register operations. There are typically a relatively large number of general-purpose registers. The number of machine instructions, instruction formats, and addressing modes is relatively small.

The discussions in the following sections will illustrate some of these RISC characteristics. Further information about the RISC approach, including its advantages and disadvantages, can be found in Tabak (1995).

1.5.1 UltraSPARC Architecture

The UltraSPARC processor, announced by Sun Microsystems in 1995, is the latest member of the SPARC family. Other members of this family include a variety of SPARC and SuperSPARC processors. The original SPARC architecture was developed in the mid-1980s, and has been implemented by a number of manufacturers. The name SPARC stands for scalable processor architecture. This architecture is intended to be suitable for a wide range of implementations, from microcomputers to supercomputers.

Although SPARC, SuperSPARC, and UltraSPARC architectures differ slightly, they are upward compatible and share the same basic structure. This section contains an overview of the UltraSPARC architecture, which will serve as background for the examples to be discussed later in the book. Further information about the SPARC family can be found in Tabak (1995) and Sun Microsystems (1995a).

Memory

Memory consists of 8-bit bytes; all addresses used are byte addresses. Two consecutive bytes form a *halfword;* four bytes form a *word;* eight bytes form a *doubleword.* Halfwords are stored in memory beginning at byte addresses that are multiples of 2. Similarly, words begin at addresses that are multiples of 4, and doublewords at addresses that are multiples of 8.

UltraSPARC programs can be written using a virtual address space of 2^{64} bytes. This address space is divided into *pages;* multiple page sizes are supported. Some of the pages used by a program may be in physical memory, while others may be stored on disk. When an instruction is executed, the hardware and the operating system make sure that the needed page is loaded into physical memory. The virtual address specified by the instruction is automatically translated into a physical address by the UltraSPARC Memory Management Unit (MMU). Chapter 6 contains a brief discussion of methods that can be used in this kind of address translation.

Registers

The SPARC architecture includes a large *register file* that usually contains more than 100 general-purpose registers. (The exact number varies from one implementation to another.) However, any procedure can access only 32 registers, designated r0 through r31. The first eight of these registers (r0 through r7) are global—that is, they can be accessed by all procedures on the system. (Register r0 always contains the value zero.)

The other 24 registers available to a procedure can be visualized as a *window* through which part of the register file can be seen. These windows overlap, so some registers in the register file are shared between procedures. For example, registers r8 through r15 of a calling procedure are physically the same registers as r24 through r31 of the called procedure. This facilitates the passing of parameters.

The SPARC hardware manages the windows into the register file. If a set of concurrently running procedures needs more windows than are physically available, a "window overflow" interrupt occurs. The operating system must then save the contents of some registers in the file (and restore them later) to provide the additional windows that are needed.

In the original SPARC architecture, the general-purpose registers were 32 bits long. Later implementations (including UltraSPARC) expanded these registers to 64 bits. Some SPARC implementations provide several physically different sets of global registers, for use by application procedures and by various hardware and operating system functions.

Floating-point computations are performed using a special *floating-point unit* (FPU). On UltraSPARC, this unit contains a file of 64 double-precision floating-point registers, and several other control and status registers.

Besides these register files, there are a program counter PC (which contains the address of the next instruction to be executed), condition code registers, and a number of other control registers.

Data Formats

The UltraSPARC architecture provides for the storage of integers, floating-point values, and characters. Integers are stored as 8-, 16-, 32-, or 64-bit binary numbers. Both signed and unsigned integers are supported; 2's complement is used for negative values. In the original SPARC architecture, the most significant part of a numeric value is stored at the lowest-numbered address. (This is commonly called *big-endian* byte ordering, because the "big end" of the value comes first in memory.) UltraSPARC supports both big-endian and little-endian byte orderings.

There are three different floating-point data formats. The single-precision format is 32 bits long. It stores 23 significant bits of the floating-point value, and allows for an 8-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.) The double-precision format is 64 bits long. It stores 52 significant bits, and allows for a 11-bit exponent. The quad-precision format stores 63 significant bits, and allows for a 15-bit exponent.

Characters are stored one per byte, using their 8-bit ASCII codes.

Instruction Formats

There are three basic instruction formats in the SPARC architecture. All of these formats are 32 bits long; the first 2 bits of the instruction word identify which format is being used. Format 1 is used for the Call instruction. Format 2 is used for branch instructions (and one special instruction that enters a value into a register). The remaining instructions use Format 3, which provides for register loads and stores, and three-operand arithmetic operations.

The fixed instruction length in the SPARC architecture is typical of RISC systems, and is intended to speed the process of instruction fetching and decoding. Compare this approach with the complex variable-length instructions found on CISC systems such as VAX and x86.

Addressing Modes

As in most architectures, an operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register direct*

mode). Operands in memory are addressed using one of the following three modes:

Mode	Target address calculation
PC-relative	TA = (PC) + displacement {30 bits, signed}
Register indirect with displacement	TA = (register) + displacement {13 bits, signed}
Register indirect indexed	TA = (register-1) + (register-2)

PC-relative mode is used only for branch instructions.

The relatively few addressing modes of SPARC allow for more efficient implementations than the 10 or more modes found on CISC systems such as x86.

Instruction Set

The basic SPARC architecture has fewer than 100 machine instructions, reflecting its RISC philosophy. (Compare this with the 300 to 400 instructions often found in CISC systems.) The only instructions that access memory are loads and stores. All other instructions are register-to-register operations.

Instruction execution on a SPARC system is *pipelined*—while one instruction is being executed, the next one is being fetched from memory and decoded. In most cases, this technique speeds instruction execution. However, an ordinary branch instruction might cause the process to "stall." The instruction following the branch (which had already been fetched and decoded) would have to be discarded without being executed.

To make the pipeline work more efficiently, SPARC branch instructions (including subroutine calls) are *delayed branches.* This means that the instruction immediately following the branch instruction is actually executed *before* the branch is taken. For example, in the instruction sequence

```
SUB     %L0, 11, %L1
BA      NEXT
MOV     %L1, %O3
```

the MOV instruction is executed before the branch BA. This MOV instruction is said to be in the *delay slot* of the branch. The programmer must take this characteristic into account when writing an assembler language program. Further discussions and examples of the use of delayed branches can be found in Section 2.5.2.