CHAPTER

1

BASIC STRUCTURE OF COMPUTERS

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- The basic structure of a computer
- Machine instructions and their execution
- System software that enables the preparation and execution of programs
- Performance issues in computer systems
- The history of computer development

This book is about computer organization. It describes the function and design of the various units of digital computers that store and process information. It also deals with the units of the computer that receive information from external sources and send computed results to external destinations. Most of the material in this book is devoted to *computer hardware* and *computer architecture*. Computer hardware consists of electronic circuits, displays, magnetic and optical storage media, electromechanical equipment, and communication facilities. Computer architecture encompasses the specification of an instruction set and the hardware units that implement the instructions.

Many aspects of programming and software components in computer systems are also discussed in this book. It is important to consider both hardware and software aspects of the design of various computer components in order to achieve a good understanding of computer systems.

This chapter introduces a number of hardware and software concepts, presents some common terminology, and gives a broad overview of the fundamental aspects of the subject. More detailed discussions follow in subsequent chapters.

1.1 COMPUTER TYPES

Let us first define the term *digital computer*, or simply *computer*. In the simplest terms, a contemporary computer is a fast electronic calculating machine that accepts digitized input information, processes it according to a list of internally stored instructions, and produces the resulting output information. The list of instructions is called a computer *program*, and the internal storage is called computer *memory*.

Many types of computers exist that differ widely in size, cost, computational power, and intended use. The most common computer is the *personal computer*, which has found wide use in homes, schools, and business offices. It is the most common form of *desktop computers*. Desktop computers have processing and storage units, visual display and audio output units, and a keyboard that can all be located easily on a home or office desk. The storage media include hard disks, CD-ROMs, and diskettes. Portable *notebook computers* are a compact version of the personal computer with all of these components packaged into a single unit the size of a thin briefcase. *Workstations* with high-resolution graphics input/output capability, although still retaining the dimensions of desktop computers, have significantly more computational power than personal computers. Workstations are often used in engineering applications, especially for interactive design work.

Beyond workstations, a range of large and very powerful computer systems exist that are called *enterprise systems* and *servers* at the low end of the range, and *supercomputers* at the high end. Enterprise systems, or *mainframes*, are used for business data processing in medium to large corporations that require much more computing power and storage capacity than workstations can provide. Servers contain sizable database storage units and are capable of handling large volumes of requests to access the data. In many cases, servers are widely accessible to the education, business, and personal user communities. The requests and responses are usually transported over Internet communication facilities. Indeed, the Internet and its associated servers have become a dominant worldwide source of all types of information. The Internet communication

facilities consist of a complex structure of high-speed fiber-optic backbone links interconnected with broadcast cable and telephone connections to schools, businesses, and homes.

Supercomputers are used for the large-scale numerical calculations required in applications such as weather forecasting and aircraft design and simulation. In enterprise systems, servers, and supercomputers, the functional units, including multiple processors, may consist of a number of separate and often large units.

1.2 FUNCTIONAL UNITS

A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure 1.1. The input unit accepts coded information from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines. The information received is either stored in the computer's memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations. The processing steps are determined by a program stored in the memory. Finally, the results are sent back to the outside world through the output unit. All of these actions are coordinated by the control unit. Figure 1.1 does not show the connections among the functional units. These connections, which can be made in several ways, are discussed throughout this book. We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the *processor*; and input and output equipment is often collectively referred to as the *input-output* (I/O) unit.

We now take a closer look at the information handled by a computer. It is convenient to categorize this information as either instructions or data. *Instructions*, or *machine instructions*, are explicit commands that

- Govern the transfer of information within a computer as well as between the computer and its I/O devices
- Specify the arithmetic and logic operations to be performed

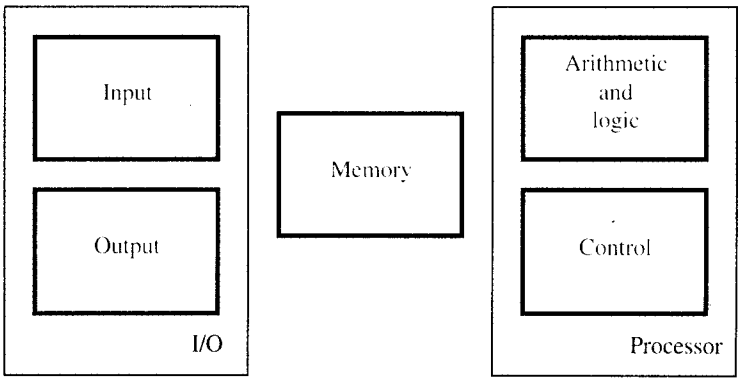

Figure 1.1 Basic functional units of a computer.

A list of instructions that performs a task is called a *program*. Usually the program is stored in the memory. The processor then fetches the instructions that make up the program from the memory, one after another, and performs the desired operations. The computer is completely controlled by the *stored program*, except for possible external interruption by an operator or by I/O devices connected to the machine.

Data are numbers and encoded characters that are used as operands by the instructions. The term data, however, is often used to mean any digital information. Within this definition of data, an entire program (that is, a list of instructions) may be considered as data if it is to be processed by another program. An example of this is the task of *compiling* a high-level language *source program* into a list of machine instructions constituting a machine language program, called the *object program*. The source program is the input data to the *compiler* program which translates the source program into a machine language program.

Information handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states, ON and OFF (see Appendix A). Each number, character, or instruction is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1. Numbers are usually represented in positional binary notation, as discussed in detail in Chapters 2 and 6. Occasionally, the *binary-coded decimal* (BCD) format is employed, in which each decimal digit is encoded by four bits.

Alphanumeric characters are also expressed in terms of binary codes. Several coding schemes have been developed. Two of the most widely used schemes are ASCII (American Standard Code for Information Interchange), in which each character is represented as a 7-bit code, and EBCDIC (Extended Binary-Coded Decimal Interchange Code), in which eight bits are used to denote a character. A more detailed description of binary notation and coding schemes is given in Appendix E.

1.2.1 INPUT UNIT

Computers accept coded information through input units, which read the data. The most well-known input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

Many other kinds of input devices are available, including joysticks, trackballs, and mouses. These are often used as graphic input devices in conjunction with displays. Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Detailed discussion of input devices and their operation is found in Chapter 10.

1.2.2 MEMORY UNIT

The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary storage is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed. The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written as individual cells but instead are processed in groups of fixed size called *words*. The memory is organized so that the contents of one word, containing n bits, can be stored or retrieved in one basic operation.

To provide easy access to any word in the memory, a distinct *address* is associated with each word location. Addresses are numbers that identify successive locations. A given word is accessed by specifying its address and issuing a control command that starts the storage or retrieval process.

The number of bits in each word is often referred to as the *word length* of the computer. Typical word lengths range from 16 to 64 bits. The capacity of the memory is one factor that characterizes the size of a computer. Small machines typically have only a few tens of millions of words, whereas medium and large machines normally have many tens or hundreds of millions of words. Data are usually processed within a machine in units of words, multiples of words, or parts of words. When the memory is accessed, usually only one word of data is read or written.

Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible. Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called *random-access memory* (RAM). The time required to access one word is called the *memory access time*. This time is fixed, independent of the location of the word being accessed. It typically ranges from a few nanoseconds (ns) to about 100 ns for modern RAM units. The memory of a computer is normally implemented as a *memory hierarchy* of three or four levels of semiconductor RAM units with different speeds and sizes. The small, fast, RAM units are called *caches*. They are tightly coupled with the processor and are often contained on the same integrated circuit chip to achieve high performance. The largest and slowest unit is referred to as the *main memory*. We will give a brief description of how information is accessed in the memory hierarchy later in the chapter. Chapter 5 discusses the operational and performance aspects of the computer memory in detail.

Although primary storage is essential, it tends to be expensive. Thus additional, cheaper, *secondary storage* is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. A wide selection of secondary storage devices is available, including *magnetic disks* and *tapes* and *optical disks* (CD-ROMs). These devices are also described in Chapter 5.

1.2.3 ARITHMETIC AND LOGIC UNIT

Most computer operations are executed in the *arithmetic and logic unit* (ALU) of the processor. Consider a typical example: Suppose two numbers located in the memory are to be added. They are brought into the processor, and the actual addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.

Any other arithmetic or logic operation, for example, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU. When operands are brought into the processor, they are stored in high-speed storage elements called *registers*. Each register can store one word of data. Access times to registers are somewhat faster than access times to the fastest cache unit in the memory hierarchy.

The control and the arithmetic and logic units are many times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as keyboards, displays, magnetic and optical disks, sensors, and mechanical controllers.

1.2.4 OUTPUT UNIT

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. The most familiar example of such a device is a *printer*. Printers employ mechanical impact heads, ink jet streams, or photocopying techniques, as in laser printers, to perform the printing. It is possible to produce printers capable of printing as many as 10,000 lines per minute. This is a tremendous speed for a mechanical device but is still very slow compared to the electronic speed of a processor unit.

Some units, such as graphic displays, provide both an output function and an input function. The dual role of such units is the reason for using the single name I/O unit in many cases.

1.2.5 CONTROL UNIT

The memory, arithmetic and logic, and input and output units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the task of the control unit. The control unit is effectively the nerve center that sends control signals to other units and senses their states.

I/O transfers, consisting of input and output operations, are controlled by the instructions of I/O programs that identify the devices involved and the information to be transferred. However, the actual *timing signals* that govern the transfers are generated by the control circuits. Timing signals are signals that determine when a given action is to take place. Data transfers between the processor and the memory are also controlled by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the machine. In practice, however, this is seldom the case. Much of the control circuitry is physically distributed throughout the machine. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.

- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities inside the machine are directed by the control unit.

1.3 BASIC OPERATIONAL CONCEPTS

In Section 1.2, we stated that the activity in a computer is governed by instructions. To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as operands are also stored in the memory. A typical instruction may be

```
Add   LOCA,R0
```

This instruction adds the operand at memory location LOCA to the operand in a register in the processor, R0, and places the sum into register R0. The original contents of location LOCA are preserved, whereas those of R0 are overwritten. This instruction requires the performance of several steps. First, the instruction is fetched from the memory into the processor. Next, the operand at LOCA is fetched and added to the contents of R0. Finally, the resulting sum is stored in register R0.

The preceding Add instruction combines a memory access operation with an ALU operation. In many modern computers, these two types of operations are performed by separate instructions for performance reasons that are explained in Chapter 8. The effect of the above instruction can be realized by the two-instruction sequence

```
Load  LOCA,R1
Add   R1,R0
```

The first of these instructions transfers the contents of memory location LOCA into processor register R1, and the second instruction adds the contents of registers R1 and R0 and places the sum into R0. Note that this destroys the former contents of register R1 as well as those of R0, whereas the original contents of memory location LOCA are preserved.

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

Figure 1.2 shows how the memory and the processor can be connected. It also shows a few essential operational details of the processor that have not been discussed yet. The interconnection pattern for these components is not shown explicitly since here we discuss only their functional characteristics. Chapter 7 describes the details of the interconnection as part of processor design.

In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes. The *instruction register* (IR) holds the instruction that is currently being executed. Its output is available to the control circuits,

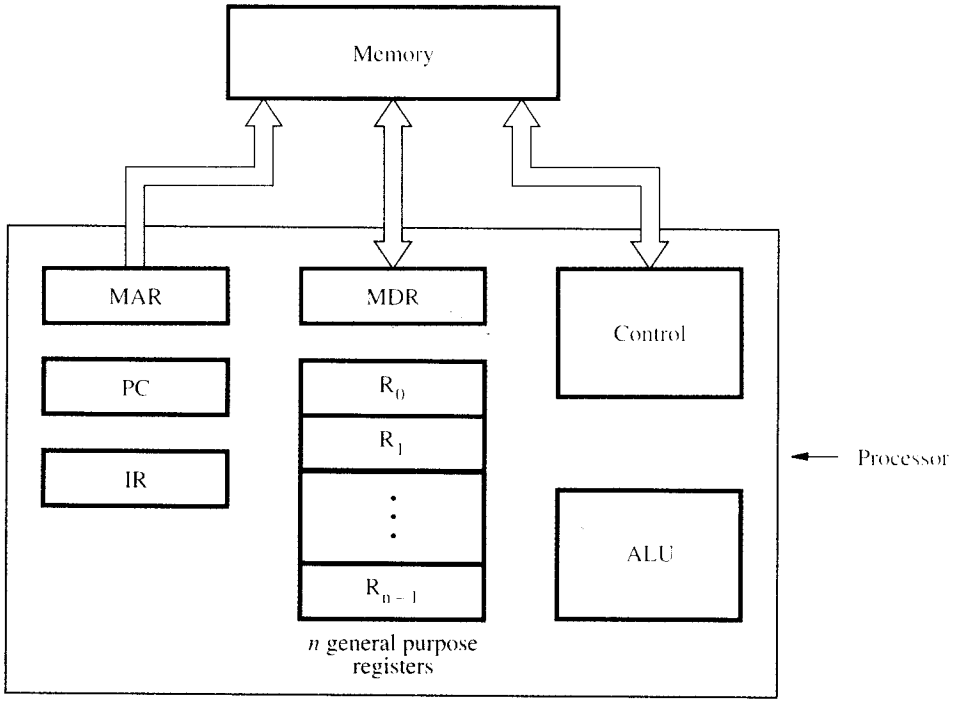

Figure 1.2 Connections between the processor and the memory.

which generate the timing signals that control the various processing elements involved in executing the instruction. The *program counter* (PC) is another specialized register. It keeps track of the execution of a program. It contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. It is customary to say that the PC *points* to the next instruction that is to be fetched from the memory. Besides the IR and PC, Figure 1.2 shows *n general-purpose registers*, R_0 through R_{n-1}. Their roles are explained in Chapter 2.

Finally, two registers facilitate communication with the memory. These are the *memory address register* (MAR) and the *memory data register* (MDR). The MAR holds the address of the location to be accessed. The MDR contains the data to be written into or read out of the addressed location.

Let us now consider some typical operating steps. Programs reside in the memory and usually get there through the input unit. Execution of the program starts when the PC is set to point to the first instruction of the program. The contents of the PC are transferred to the MAR and a Read control signal is sent to the memory. After the time required to access the memory elapses, the addressed word (in this case, the first instruction of the program) is read out of the memory and loaded into the MDR. Next, the contents of the MDR are transferred to the IR. At this point, the instruction is ready to be decoded and executed.

If the instruction involves an operation to be performed by the ALU, it is necessary to obtain the required operands. If an operand resides in the memory (it could also be in a general-purpose register in the processor), it has to be fetched by sending its address to the MAR and initiating a Read cycle. When the operand has been read from the memory into the MDR, it is transferred from the MDR to the ALU. After one or more operands are fetched in this way, the ALU can perform the desired operation. If the result of this operation is to be stored in the memory, then the result is sent to the MDR. The address of the location where the result is to be stored is sent to the MAR, and a Write cycle is initiated. At some point during the execution of the current instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, a new instruction fetch may be started.

In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions with the ability to handle I/O transfers are provided.

Normal execution of programs may be preempted if some device requires urgent servicing. For example, a monitoring device in a computer-controlled industrial process may detect a dangerous condition. In order to deal with the situation immediately, the normal execution of the current program must be interrupted. To do this, the device raises an *interrupt* signal. An interrupt is a request from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate *interrupt-service routine*. Because such diversions may alter the internal state of the processor, its state must be saved in memory locations before servicing the interrupt. Normally, the contents of the PC, the general registers, and some control information are stored in memory. When the interrupt-service routine is completed, the state of the processor is restored so that the interrupted program may continue.

The processor unit shown in Figure 1.2 is usually implemented on a single Very Large Scale Integrated (VLSI) chip, with at least one of the cache units of the memory hierarchy contained on the same chip.

1.4 BUS STRUCTURES

So far, we have discussed the functions of individual parts of a computer. To form an operational system, these parts must be connected in some organized way. There are many ways of doing this. We consider the simplest and most common of these here.

To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time. When a word of data is transferred between units, all its bits are transferred in parallel, that is, the bits are transferred simultaneously over many wires, or lines, one bit per line. A group of lines that serves as a connecting path for several devices is called a *bus*. In addition to the lines that carry the data, the bus must have lines for address and control purposes.

The simplest way to interconnect functional units is to use a *single bus*, as shown in Figure 1.3. All units are connected to this bus. Because the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus

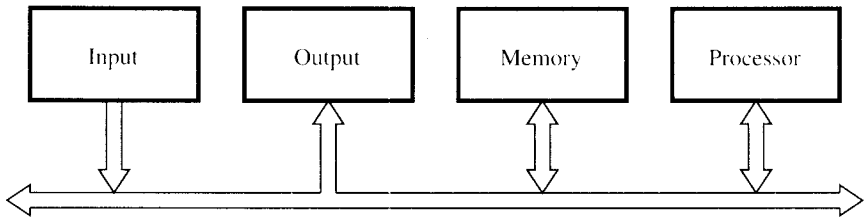

Figure 1.3 Single-bus structure.

control lines are used to arbitrate multiple requests for use of the bus. The main virtue of the single-bus structure is its low cost and its flexibility for attaching peripheral devices. Systems that contain multiple buses achieve more concurrency in operations by allowing two or more transfers to be carried out at the same time. This leads to better performance but at an increased cost.

The devices connected to a bus vary widely in their speed of operation. Some electromechanical devices, such as keyboards and printers, are relatively slow. Others, like magnetic or optical disks, are considerably faster. Memory and processor units operate at electronic speeds, making them the fastest parts of a computer. Because all these devices must communicate with each other over a bus, an efficient transfer mechanism that is not constrained by the slow devices and that can be used to smooth out the differences in timing among processors, memories, and external devices is necessary.

A common approach is to include *buffer registers* with the devices to hold the information during transfers. To illustrate this technique, consider the transfer of an encoded character from a processor to a character printer. The processor sends the character over the bus to the printer buffer. Since the buffer is an electronic register, this transfer requires relatively little time. Once the buffer is loaded, the printer can start printing without further intervention by the processor. The bus and the processor are no longer needed and can be released for other activity. The printer continues printing the character in its buffer and is not available for further transfers until this process is completed. Thus, buffer registers smooth out timing differences among processors, memories, and I/O devices. They prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers. This allows the processor to switch rapidly from one device to another, interweaving its processing activity with data transfers involving several I/O devices.

1.5 SOFTWARE

In order for a user to enter and run an application program, the computer must already contain some system software in its memory. *System software* is a collection of programs that are executed as needed to perform functions such as

- Receiving and interpreting user commands
- Entering and editing application programs and storing them as files in secondary storage devices

- Managing the storage and retrieval of files in secondary storage devices
- Running standard application programs such as word processors, spreadsheets, or games, with data supplied by the user
- Controlling I/O units to receive input information and produce output results
- Translating programs from source form prepared by the user into object form consisting of machine instructions
- Linking and running user-written application programs with existing standard library routines, such as numerical computation packages

System software is thus responsible for the coordination of all activities in a computing system. The purpose of this section is to introduce some basic aspects of system software.

Application programs are usually written in a high-level programming language, such as C, C++, Java, or Fortran, in which the programmer specifies mathematical or text-processing operations. These operations are described in a format that is independent of the particular computer used to execute the program. A programmer using a high-level language need not know the details of machine program instructions. A system software program called a *compiler* translates the high-level language program into a suitable machine language program containing instructions such as the Add and Load instructions discussed in Section 1.3.

Another important system program that all programmers use is a *text editor*. It is used for entering and editing application programs. The user of this program interactively executes commands that allow statements of a source program entered at a keyboard to be accumulated in a *file*. A file is simply a sequence of alphanumeric characters or binary data that is stored in memory or in secondary storage. A file can be referred to by a name chosen by the user.

We do not pursue the details of compilers, editors, or file systems in this book, but let us take a closer look at a key system software component called the *operating system* (OS). This is a large program, or actually a collection of routines, that is used to control the sharing of and interaction among various computer units as they execute application programs. The OS routines perform the tasks required to assign computer resources to individual application programs. These tasks include assigning memory and magnetic disk space to program and data files, moving data between memory and disk units, and handling I/O operations.

In order to understand the basics of operating systems, let us consider a system with one processor, one disk, and one printer. First we discuss the steps involved in running one application program. Once we have explained these steps, we can understand how the operating system manages the execution of more than one application program at the same time. Assume that the application program has been compiled from a high-level language form into a machine language form and stored on the disk. The first step is to transfer this file into the memory. When the transfer is complete, execution of the program is started. Assume that part of the program's task involves reading a data file from the disk into the memory, performing some computation on the data, and printing the results. When execution of the program reaches the point where the data file is needed, the program requests the operating system to transfer the data file from the

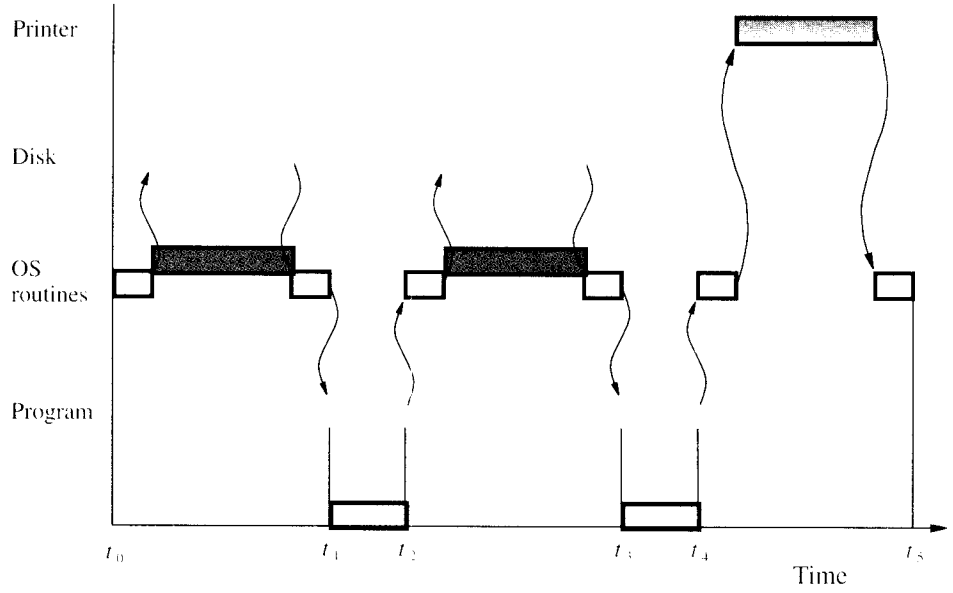

Figure 1.4 User program and OS routine sharing of the processor.

disk to the memory. The OS performs this task and passes execution control back to the application program, which then proceeds to perform the required computation. When the computation is completed and the results are ready to be printed, the application program again sends a request to the operating system. An OS routine is then executed to cause the printer to print the results.

We have seen how execution control passes back and forth between the application program and the OS routines. A convenient way to illustrate this sharing of the processor execution time is by a time-line diagram, such as that shown in Figure 1.4. During the time period t_0 to t_1, an OS routine initiates loading the application program from disk to memory, waits until the transfer is completed, and then passes execution control to the application program. A similar pattern of activity occurs during period t_2 to t_3 and period t_4 to t_5, when the operating system transfers the data file from the disk and prints the results. At t_5, the operating system may load and execute another application program.

Now, let us point out a way that computer resources can be used more efficiently if several application programs are to be processed. Notice that the disk and the processor are idle during most of the time period t_4 to t_5. The operating system can load the next program to be executed into the memory from the disk while the printer is operating. Similarly, during t_0 to t_1, the operating system can arrange to print the previous program's results while the current program is being loaded from the disk. Thus, the operating system manages the concurrent execution of several application programs to make the best possible use of computer resources. This pattern of concurrent execution is called *multiprogramming* or *multitasking*.

1.6 PERFORMANCE

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes programs is affected by the design of its hardware and its machine language instructions. Because programs are usually written in a high-level language, performance is also affected by the compiler that translates programs into machine language. For best performance, it is necessary to design the compiler, the machine instruction set, and the hardware in a coordinated way. We do not describe the details of compiler design in this book. We concentrate on the design of instruction sets and hardware.

In Section 1.5, we described how the operating system overlaps processing, disk transfers, and printing for several programs to make the best possible use of the resources available. The total time required to execute the program in Figure 1.4 is $t_5 - t_0$. This *elapsed time* is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk, and the printer. To discuss the performance of the processor, we should consider only the periods during which the processor is active. These are the periods labeled Program and OS routines in Figure 1.4. We will refer to the sum of these periods as the *processor time* needed to execute the program. In what follows, we will identify some of the key parameters that affect the processor time and point out the chapters in which the relevant issues are discussed. We encourage the readers to keep this broad overview of performance in mind as they study the material presented in subsequent chapters.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory, which are usually connected by a bus as shown in Figure 1.3. The pertinent parts of this figure are repeated in Figure 1.5, including the cache memory as part of the processor unit. Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache. When the execution of an instruction calls for data located in the main memory, the data are

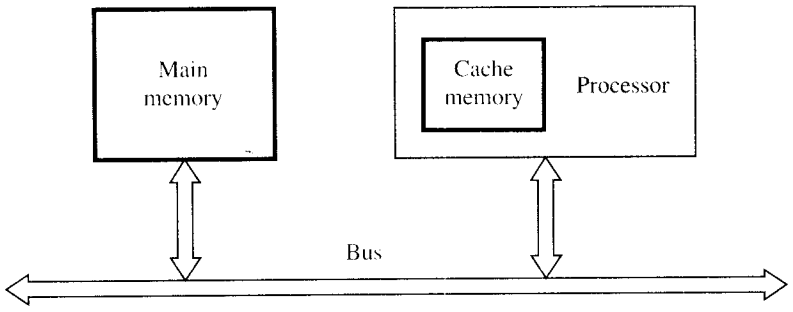

Figure 1.5 The processor cache.

fetched and a copy is placed in the cache. Later, if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip. The internal speed of performing the basic steps of instruction processing on such chips is very high and is considerably faster than the speed at which instructions and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache. For example, suppose a number of instructions are executed repeatedly over a short period of time, as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to data that are used repeatedly. Design, operation, and performance issues for the main memory and the cache are discussed in Chapter 5.

1.6.1 PROCESSOR CLOCK

Processor circuits are controlled by a timing signal called a *clock*. The clock defines regular time intervals, called *clock cycles*. To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each step can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects processor performance. Its inverse is the clock rate, $R = 1/P$, which is measured in cycles per second. Processors used in today's personal computers and workstations have clock rates that range from a few hundred million to over a billion cycles per second. In standard electrical engineering terminology, the term "cycles per second" is called *hertz* (Hz). The term "million" is denoted by the prefix *Mega* (M), and "billion" is denoted by the prefix *Giga* (G). Hence, 500 million cycles per second is usually abbreviated to 500 Megahertz (MHz), and 1250 million cycles per second is abbreviated to 1.25 Gigahertz (GHz). The corresponding clock periods are 2 and 0.8 nanoseconds (ns), respectively.

1.6.2 BASIC PERFORMANCE EQUATION

We now focus our attention on the processor time component of the total elapsed time. Let T be the processor time required to execute a program that has been prepared in some high-level language. The compiler generates a machine language object program that corresponds to the source program. Assume that complete execution of the program requires the execution of N machine language instructions. The number N is the actual number of instruction executions, and is not necessarily equal to the number of machine instructions in the object program. Some instructions may be executed more than once, which is the case for instructions inside a program loop. Others may not be executed at all, depending on the input data used. Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock rate is R cycles per second, the program execution time is

given by

$$T = \frac{N \times S}{R} \qquad [1.1]$$

This is often referred to as the *basic performance equation*.

The performance parameter T for an application program is much more important to the user than the individual values of the parameters N, S, or R. To achieve high performance, the computer designer must seek ways to reduce the value of T, which means reducing N and S, and increasing R. The value of N is reduced if the source program is compiled into fewer machine instructions. The value of S is reduced if instructions have a smaller number of basic steps to perform or if the execution of instructions is overlapped. Using a higher-frequency clock increases the value or R, which means that the time required to complete a basic execution step is reduced.

We must emphasize that N, S, and R are not independent parameters; changing one may affect another. Introducing a new feature in the design of a processor will lead to improved performance only if the overall result is to reduce the value of T. A processor advertised as having a 900-MHz clock does not necessarily provide better performance than a 700-MHz processor because it may have a different value of S.

1.6.3 PIPELINING AND SUPERSCALAR OPERATION

In the discussion above, we assumed that instructions are executed one after another. Hence, the value of S is the total number of basic steps, or clock cycles, required to execute an instruction. A substantial improvement in performance can be achieved by overlapping the execution of successive instructions, using a technique called *pipelining*. Consider the instruction

```
Add   R1,R2,R3
```

which adds the contents of registers R1 and R2, and places the sum into R3. The contents of R1 and R2 are first transferred to the inputs of the ALU. After the add operation is performed, the sum is transferred to R3. The processor can read the next instruction from the memory while the addition operation is being performed. Then, if that instruction also uses the ALU, its operands can be transferred to the ALU inputs at the same time that the result of the Add instruction is being transferred to R3. In the ideal case, if all instructions are overlapped to the maximum degree possible, execution proceeds at the rate of one instruction completed in each clock cycle. Individual instructions still require several clock cycles to complete. But, for the purpose of computing T, the effective value of S is 1.

Pipelining is discussed in detail in Chapter 8. As we will see, the ideal value $S = 1$ cannot be attained in practice for a variety of reasons. However, pipelining increases the rate of executing instructions significantly and causes the effective value of S to approach 1.

A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor. This means that multiple functional units are used,

creating parallel paths through which different instructions can be executed in parallel. With such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle. This mode of operation is called *superscalar execution*. If it can be sustained for a long time during program execution, the effective value of S can be reduced to less than one. Of course, parallel execution must preserve the logical correctness of programs, that is, the results produced must be the same as those produced by serial execution of program instructions. Many of today's high-performance processors are designed to operate in this manner.

1.6.4 CLOCK RATE

There are two possibilities for increasing the clock rate, R. First, improving the *integrated-circuit* (IC) technology makes logic circuits faster, which reduces the time needed to complete a basic step. This allows the clock period, P, to be reduced and the clock rate, R, to be increased. Second, reducing the amount of processing done in one basic step also makes it possible to reduce the clock period, P. However, if the actions that have to be performed by an instruction remain the same, the number of basic steps needed may increase.

Increases in the value of R that are entirely caused by improvements in IC technology affect all aspects of the processor's operation equally with the exception of the time it takes to access the main memory. In the presence of a cache, the percentage of accesses to the main memory is small. Hence, much of the performance gain expected from the use of faster technology can be realized. The value of T will be reduced by the same factor as R is increased because S and N are not affected. The impact on performance of changing the way in which instructions are divided into basic steps is more difficult to assess. This issue is discussed in Chapter 8.

1.6.5 INSTRUCTION SET: CISC AND RISC

Simple instructions require a small number of basic steps to execute. Complex instructions involve a large number of steps. For a processor that has only simple instructions, a large number of instructions may be needed to perform a given programming task. This could lead to a large value for N and a small value for S. On the other hand, if individual instructions perform more complex operations, fewer instructions will be needed, leading to a lower value of N and a larger value of S. It is not obvious if one choice is better than the other.

A key consideration in comparing the two choices is the use of pipelining. We pointed out earlier that the effective value of S in a pipelined processor is close to 1 even though the number of basic steps per instruction may be considerably larger. This seems to imply that complex instructions combined with pipelining would achieve the best performance. However, it is much easier to implement efficient pipelining in processors with simple instruction sets. The suitability of the instruction set for pipelined execution is an important and often deciding consideration.

The design of the instruction set of a processor and the options available are discussed in Chapter 2. The relative merits of processors with simple instructions and processors with more complex instructions have been studied a great deal [1]. The former are called *Reduced Instruction Set Computers* (RISC), and the latter are referred to as *Complex Instruction Set Computers* (CISC). We give examples of RISC and CISC processors in Chapters 3 and 11, and discuss their merits. Although we use the terms RISC and CISC in order to be compatible with contemporary descriptions, we caution the reader not to assume that they correspond to clearly defined classes of processors. A given processor design is a result of many trade-offs. The terms RISC and CISC refer to design principles and techniques, which we discuss in several places in the book.

1.6.6 COMPILER

A compiler translates a high-level language program into a sequence of machine instructions. To reduce N, we need to have a suitable machine instruction set and a compiler that makes good use of it. An *optimizing compiler* takes advantage of various features of the target processor to reduce the product $N \times S$, which is the total number of clock cycles needed to execute a program. We will see in Chapter 8 that the number of cycles is dependent not only on the choice of instructions, but also on the order in which they appear in the program. The compiler may rearrange program instructions to achieve better performance. Of course, such changes must not affect the result of the computation.

Superficially, a compiler appears as a separate entity from the processor with which it is used and may even be available from a different vendor. However, a high-quality compiler must be closely linked to the processor architecture. The compiler and the processor are often designed at the same time, with much interaction between the designers to achieve best results. The ultimate objective is to reduce the total number of clock cycles needed to perform a required programming task.

1.6.7 PERFORMANCE MEASUREMENT

It is important to be able to assess the performance of a computer. Computer designers use performance estimates to evaluate the effectiveness of new features. Manufacturers use performance indicators in the marketing process. Buyers use such data to choose among many available computer models.

The previous discussion suggests that the only parameter that properly describes the performance of a computer is the execution time, T, for the programs of interest. Despite the conceptual simplicity of Equation 1.1, computing the value of T is not simple. Moreover, parameters such as the clock speed and various architectural features are not reliable indicators of the expected performance.

For these reasons, the computer community adopted the idea of measuring computer performance using benchmark programs. To make comparisons possible, standardized programs must be used. The performance measure is the time it takes a computer

to execute a given benchmark. Initially, some attempts were made to create artificial programs that could be used as standard benchmarks. But, synthetic programs do not properly predict performance obtained when real application programs are run.

The accepted practice today is to use an agreed-upon selection of real application programs to evaluate performance. A nonprofit organization called System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers. For general-purpose computers, a suite of benchmark programs was selected in 1989. It was modified somewhat and published in 1995 and again in 2000.

The programs selected range from game playing, compiler, and database applications to numerically intensive programs in astrophysics and quantum chemistry. In each case, the program is compiled for the computer under test, and the running time on a real computer is measured. (Simulation is not allowed.) The same program is also compiled and run on one computer selected as a reference. For SPEC95, the reference is the SUN SPARCstation 10/40. For SPEC2000, the reference computer is an UltraSPARC10 workstation with a 300-MHz UltraSPARC-IIi processor. The SPEC rating is computed as follows

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

Thus a SPEC rating of 50 means that the computer under test is 50 times as fast as the UltraSPARC10 for this particular benchmark. The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed. Let SPEC_i be the rating for program i in the suite. The overall SPEC rating for the computer is given by

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i \right)^{\frac{1}{n}}$$

where n is the number of programs in the suite.

Because the actual execution time is measured, the SPEC rating is a measure of the combined effect of all factors affecting performance, including the compiler, the operating system, the processor, and the memory of the computer being tested. Details about the SPEC benchmark programs and results of the tests conducted can be found on the SPEC web page [2].

1.7 MULTIPROCESSORS AND MULTICOMPUTERS

So far, we have considered computers with one processor. Large computer systems may contain a number of processor units, in which case they are called *multiprocessor* systems. These systems either execute a number of different application tasks in parallel, or they execute subtasks of a single large task in parallel. All processors usually have access to all of the memory in such systems, and the term *shared-memory multiprocessor* systems is often used to make this clear. The high performance of these systems

comes with much increased complexity and cost. In addition to multiple processors and memory units, cost is increased because of the need for more complex interconnection networks.

In contrast to multiprocessor systems, it is also possible to use an interconnected group of complete computers to achieve high total computational power. The computers normally have access only to their own memory units. When the tasks they are executing need to communicate data, they do so by exchanging *messages* over a communication network. This property distinguishes them from shared-memory multiprocessors, leading to the name *message-passing multicomputers*.

Shared-memory multiprocessors and message-passing multicomputers, along with the interconnection networks used in such systems, are described in Chapter 12.

1.8 HISTORICAL PERSPECTIVE

Computers as we know them today have been developed over the past 60 years. A long, slow evolution of mechanical calculating devices preceded the development of computers. Many sources describe this history. Hayes [3], for example, gives an excellent account of computer history, including dates, inventors, designers, research organizations, and manufacturers. Here, we briefly sketch the history of computer development.

In the 300 years before the mid-1900s, a series of increasingly complex mechanical devices, constructed from gear wheels, levers, and pulleys, were used to perform the basic operations of addition, subtraction, multiplication, and division. Holes on punched cards were mechanically sensed and used to control the automatic sequencing of a list of calculations and essentially provide a programming capability. These devices enabled the computation of complete mathematical tables of logarithms and trigonometric functions as approximated by polynomials. Output results were punched on cards or printed on paper. Electromechanical relay devices, such as those used in early telephone switching systems, provided the means for performing logic functions in computers built during World War II. At the same time, the first electronic computer was designed and built at the University of Pennsylvania, based on vacuum tube technology that was in use in radios and military radar equipment. Vacuum tubes were used to perform logic operations and to store data. This technology began the modern era of electronic digital computers.

Development of the technologies used to fabricate the processors, memories, and I/O units of computers has been divided into four generations: the first generation, 1945 to 1955; the second generation, 1955 to 1965; the third generation, 1965 to 1975; and the fourth generation, 1975 to the present.

1.8.1 THE FIRST GENERATION

The key concept of a stored program was introduced by John von Neumann. Programs and their data were located in the same memory, as they are today. Assembly language was used to prepare programs and was translated into machine language for execution.

Basic arithmetic operations were performed in a few milliseconds using vacuum tube technology to implement logic functions. This provided a 100- to 1000-fold increase in speed relative to the earlier mechanical and relay-based electromechanical technology. Mercury delay-line memory was used at first, and I/O functions were performed by devices similar to typewriters. Magnetic core memories and magnetic tape storage devices were also developed.

1.8.2 THE SECOND GENERATION

The transistor was invented at AT&T Bell Laboratories in the late 1940s and quickly replaced the vacuum tube. This basic technology shift marked the start of the second generation. Magnetic core memories and magnetic drum storage devices were more widely used in the second generation. High-level languages, such as Fortran, were developed, making the preparation of application programs much easier. System programs called compilers were developed to translate these high-level language programs into a corresponding assembly language program, which was then translated into executable machine language form. Separate I/O processors were developed that could operate in parallel with the central processor that executed programs, thus improving overall performance. IBM became a major computer manufacturer during this time.

1.8.3 THE THIRD GENERATION

The ability to fabricate many transistors on a single silicon chip, called integrated-circuit technology, enabled lower-cost and faster processors and memory elements to be built. Integrated-circuit memories began to replace magnetic core memories. This technological development marked the beginning of the third generation. Other developments included the introduction of microprogramming, parallelism, and pipelining. Operating system software allowed efficient sharing of a computer system by several user programs. Cache and virtual memories were developed. Cache memory makes the main memory appear faster than it really is, and virtual memory makes it appear larger. System 360 mainframe computers from IBM and the line of PDP minicomputers from Digital Equipment Corporation were dominant commercial products of the third generation.

1.8.4 THE FOURTH GENERATION

In the early 1970s, integrated-circuit fabrication techniques had evolved to the point where complete processors and large sections of the main memory of small computers could be implemented on single chips. Tens of thousands of transistors could be placed on a single chip, and the name *Very Large Scale Integration* (VLSI) was coined to describe this technology. VLSI technology allowed a complete processor to be fabricated

on a single chip; this became known as a microprocessor. Companies such as Intel, National Semiconductor, Motorola, Texas Instruments, and Advanced Micro Devices, were the driving forces of this technology.

Organizational concepts such as concurrency, pipelining, caches, and virtual memories evolved to produce the high-performance computing systems of today as the fourth generation matured. Portable notebook computers, desktop personal computers and workstations, interconnected by local area networks, wide area networks, and the Internet, have become the dominant mode of computing. Centralized computing on mainframes is now used primarily for business applications in large companies.

1.8.5 BEYOND THE FOURTH GENERATION

Generation numbers beyond four have been used occasionally to describe some computer systems that have a dominant organizational or application-driven feature. In recent years, there has been a tendency to use such features rather than a generation number to describe these evolving systems. Computers featuring artificial intelligence, massively parallel machines, and extensively distributed systems are examples of current trends. Perhaps most importantly, the growth of the computer industry is fueled by increasingly powerful and affordable desktop computers and widespread use of the vast information resources on the Internet.

1.8.6 EVOLUTION OF PERFORMANCE

The shift from mechanical and electromechanical devices to the first electronic devices based on vacuum tubes caused a 100- to 1000-fold speed increase, from seconds to milliseconds. The replacement of tubes by transistors led to another 1000-fold increase in speed, when basic operations could be performed in microseconds. Increased density in the fabrication of integrated circuits has led to current microprocessor chips that perform basic operations in a nanosecond or less, achieving a further 1000-fold increase in speed. In addition to developments in technology, there have been many innovations in the architecture of computers, such as the use of caches and pipelining, which have had a significant impact on computer performance.

1.9 CONCLUDING REMARKS

This chapter considered many aspects of computer structures and operation. Much of the terminology needed to deal with the subject was introduced, and an overview of some important design concepts was presented. The subsequent chapters will provide complete explanations of these terms and concepts, and will place the various parts of this chapter into proper perspective.

PROBLEMS

1.1 List the steps needed to execute the machine instruction

Add LOCA,R0

in terms of transfers between the components shown in Figure 1.2 and some simple control commands. Assume that the instruction itself is stored in the memory at location INSTR and that this address is initially in register PC. The first two steps might be expressed as

- Transfer the contents of register PC to register MAR.
- Issue a Read command to the memory, and then wait until it has transferred the requested word into register MDR.

Remember to include the steps needed to update the contents of PC from INSTR to INSTR+1 so that the next instruction can be fetched.

1.2 Repeat Problem 1.1 for the machine instruction

Add R1,R2,R3

which was discussed in Section 1.6.3.

1.3 (*a*) Give a short sequence of machine instructions for the task: "Add the contents of memory location A to those of location B, and place the answer in location C." Instructions

Load LOC,R_i

and

Store R_i,LOC

are the only instructions available to transfer data between the memory and general-purpose register R_i. Add instructions were described in Sections 1.3 and 1.6.3. Do not destroy the contents of either location A or B.

(*b*) Suppose that Move and Add instructions are available with the format

Move/Add Location1,Location2

These instructions move or add a copy of the operand at the first location to the second location, overwriting the original operand at the second location. Location*i* can be in either the memory or the processor register set. Is it possible to use fewer instructions to accomplish the task in Part *a*? If yes, give the sequence.

1.4 (*a*) Section 1.5 discusses how the input and output steps of a collection of programs such as the one shown in Figure 1.4 could be overlapped to reduce the total time needed to execute them. Let each of the six OS routine execution intervals be 1 unit of time, with each disk operation requiring 3 units, printing requiring 3 units, and each program execution interval requiring 2 units of time. Compute the ratio of

best overlapped time to nonoverlapped time for a long sequence of programs. Ignore start-up and ending transients.

(*b*) Section 1.5 indicated that program computation can be overlapped with either input or output operations or both. Ignoring the relatively short time needed for OS routines, what is the ratio of best overlapped time to nonoverlapped time for completing the execution of a collection of programs, where each program has about equal balance among input, compute, and output activities?

1.5 (*a*) Program execution time, T, as defined in Section 1.6.2, is to be examined for a certain high-level language program. The program can be run on a RISC or a CISC computer. Both computers use pipelined instruction execution, but pipelining in the RISC machine is more effective than in the CISC machine. Specifically, the effective value of S in the T expression for the RISC machine is 1.2, but it is only 1.5 for the CISC machine. Both machines have the same clock rate, R. What is the largest allowable value for N, the number of instructions executed on the CISC machine, expressed as a percentage of the N value for the RISC machine, if time for execution on the CISC machine is to be no longer than that on the RISC machine?

(*b*) Repeat Part *a* if the clock rate, R, for the RISC machine is 15 percent higher than that for the CISC machine.

1.6 (*a*) A processor cache, as shown in Figure 1.5, is discussed in Section 1.6. Suppose that execution time for a program is directly proportional to instruction access time and that access to an instruction in the cache is 20 times faster than access to an instruction in the main memory. Assume that a requested instruction is found in the cache with probability 0.96, and also assume that if an instruction is not found in the cache, it must first be fetched from the main memory to the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is usually defined as the speedup factor resulting from the presence of the cache.

(*b*) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat Part *a* for a doubled cache size.

REFERENCES

1. D.A. Patterson and J.L. Hennessy, *Computer Organization and Design — The Hardware/Software Interface*, 2nd ed., Morgan Kaufmann, San Mateo, Calif., 1998.
2. System Performance Evaluation Corporation web page: www.spec.org.
3. J.P. Hayes, *Computer Architecture and Organization*, 3rd ed., McGraw-Hill, New York, 1998.