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

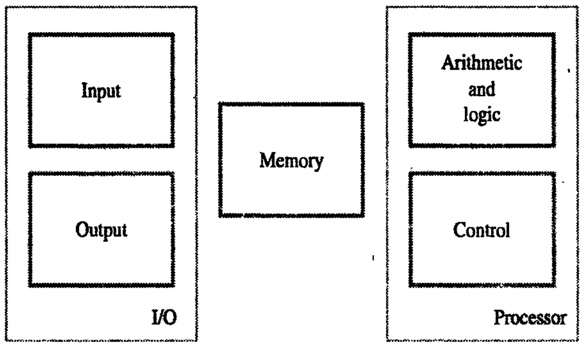


Figure 1.1 Basic functional units of a computer.

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,

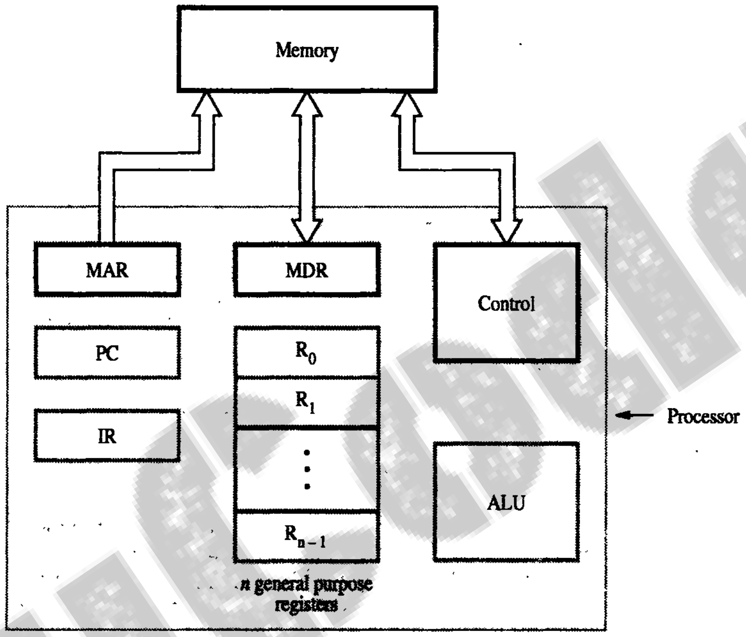


Figure 1.2 Connections between the processor and the memory.

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

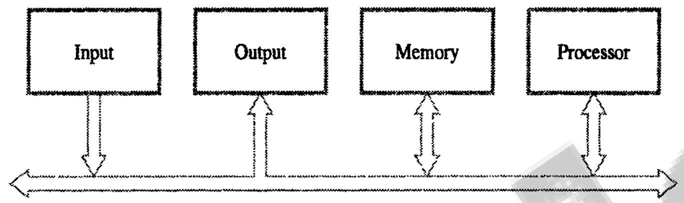


Figure 1.3 Single-bus structure.

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

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

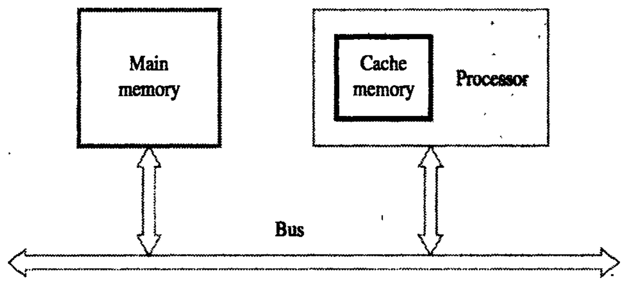


Figure 1.5 The processor cache.

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

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

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

2.1.5 CHARACTERS

In addition to numbers, computers must be able to handle nonnumeric text information consisting of characters. Characters can be letters of the alphabet, decimal digits, punctuation marks, and so on. They are represented by codes that are usually eight bits long. One of the most widely used such codes is the American Standards Committee on Information Interchange (ASCII) code described in Appendix E.

2.2 MEMORY LOCATIONS AND ADDRESSES

Number and character operands, as well as instructions, are stored in the memory of a computer. We will now consider how the memory is organized. The memory consists of many millions of storage *cells,* each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a *word* of information, and n is called the *word length.* The memory of a computer can be schematically represented as a collection of words as shown in Figure 2.5.

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2's-complement number or four ASCII characters, each occupying 8 bits, as shown in Figure 2.6. A unit of 8 bits is called a *byte.* Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section after we have described instructions at the assembly language level.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each item location. It is customary to use numbers from 0 through $2^k - 1$, for some suitable value of k, as the addresses of successive locations in the memory. The 2^k addresses constitute the *address space* of the computer, and the memory can have up to 2^k addressable locations. For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 2^{20} (1,048,576). A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations, where 1G is 2^{30}. Other notational conventions that are commonly used are K (kilo) for the number 2^{10} (1,024), and T (tera) for the number 2^{40}.

2.2.1 BYTE ADDRESSABILITY

We now have three basic information quantities to deal with: the bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte

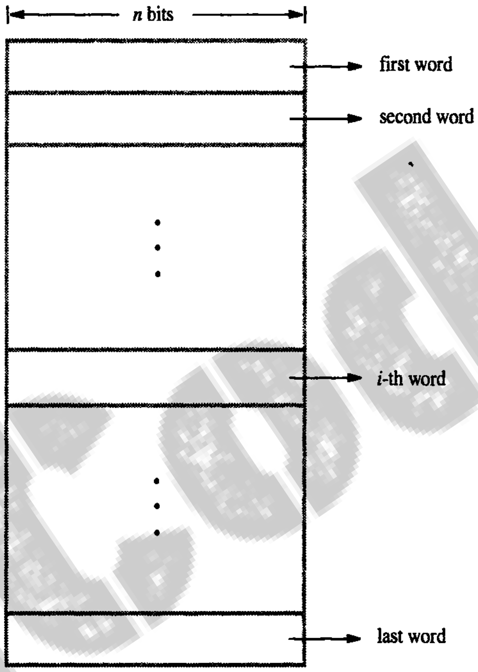
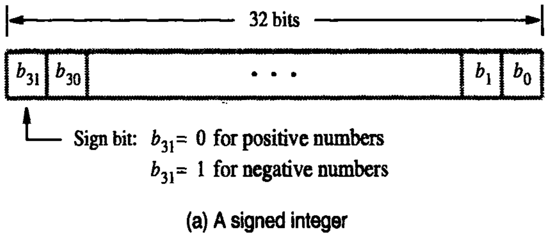


Figure 2.5 Memory words.


32 bits
b_{31} b_{30} • • • b_1 b_0
Sign bit: b_{31} = 0 for positive numbers
b_{31} = 1 for negative numbers
(a) A signed integer

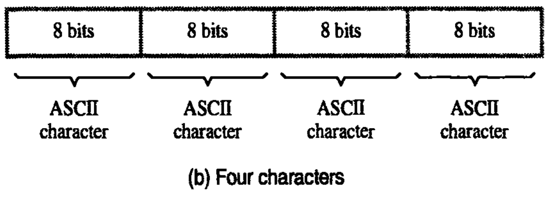


Figure 2.6 Examples of encoded information in a 32-bit word.

locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2, ... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, ..., with each word consisting of four bytes.

2.2.2 BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.7. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number, as described in Section 2.1.1. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, ..., are taken as the addresses of successive words in the memory and are the addresses used when specifying memory read and write operations for words.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 2.6*a*. It is the

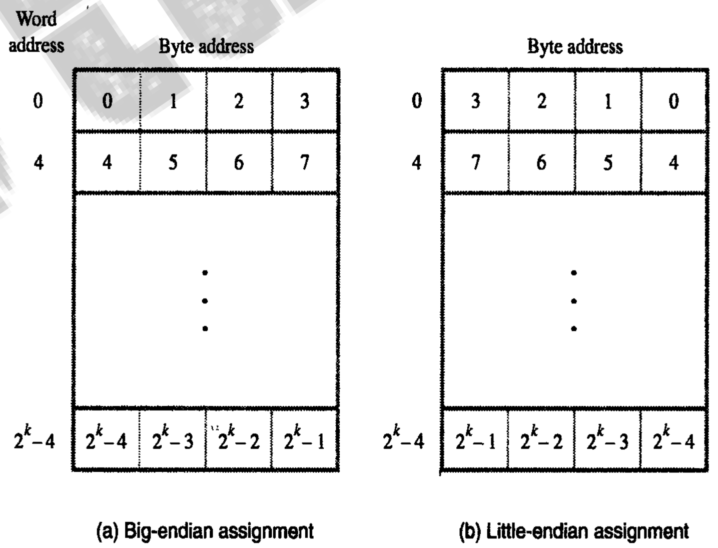

Word address
Byte address
0
4
2^k-4
0 1 2 3
4 5 6 7
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
4
2^k-4
3 2 1 0
7 6 5 4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

Figure 2.7 Byte and word addressing.

most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is, $b_7, b_6, \ldots, b_0$, from left to right. There are computers, however, that use the reverse ordering.

2.2.3 WORD ALIGNMENT

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, ..., as shown in Figure 2.7. We say that the word locations have *aligned* addresses. In general, words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, ..., and for a word length of 64 (2^3 bytes), aligned words begin at byte addresses 0, 8, 16, ...

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

2.2.4 ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. There are two ways to indicate the length of the string. A special control character with the meaning "end of string" can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

2.3 MEMORY OPERATIONS

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Load* (or *Read* or *Fetch*) and *Store* (or *Write*).

The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

An information item of either one word or one byte can be transferred between the processor and the memory in a single operation. As described in Chapter 1, the processor contains a small number of registers, each capable of holding a word. These registers are either the source or the destination of a transfer to or from the memory. When a byte is transferred, it is usually located in the low-order (rightmost) byte position of the register.

The details of the hardware implementation of these operations are treated in Chapters 5 and 7. In this chapter, we are taking the ISA viewpoint, so we concentrate on the logical handling of instructions and operands. Specific hardware components, such as processor registers, are discussed only to the extent necessary to understand the execution of machine instructions and programs.

2.4 INSTRUCTIONS AND INSTRUCTION SEQUENCING

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing the first two types of instructions. To facilitate the discussion, we need some notation which we present first.

2.4.1 REGISTER TRANSFER NOTATION

We need to describe the transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address. For example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor register names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

$$R1 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

2.4.2 Assembly Language Notation

We need another type of notation to represent machine instructions and programs. For this, we use an *assembly language* format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

```
Move   LOC,R1
```

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

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

2.4.3 Basic Instruction Types

The operation of adding two numbers is a fundamental capability in any computer. The statement

$$C = A + B$$

in a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. We will use the variable names to refer to the corresponding memory location addresses. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action

$$C \leftarrow [A] + [B]$$

to take place in the computer. To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands — A, B, and C. This *three-address* instruction can be represented symbolically as

```
Add   A,B,C
```

Operands A and B are called the *source* operands, C is called the *destination* operand, and Add is the operation to be performed on the operands. A general instruction of this type has the format

```
Operation   Source1,Source2,Destination
```

If k bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain $3k$ bits for addressing purposes in addition to the bits needed to denote the Add operation. For a modern processor with a 32-bit address space, a 3-address instruction is too large to fit in one word for a reasonable word length. Thus, a format that allows multiple words to be used for a single instruction would be needed to represent an instruction of this type.

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that *two-address* instructions of the form

```
Operation   Source,Destination
```

are available. An Add instruction of this type is

```
Add   A,B
```

which performs the operation $B \leftarrow [A] + [B]$. When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.

A single two-address instruction cannot be used to solve our original problem, which is to add the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

```
Move   B,C
```

which performs the operation $C \leftarrow [B]$, leaving the contents of location B unchanged. The word "Move" is a misnomer here; it should be "Copy." However, this instruction name is deeply entrenched in computer nomenclature. The operation $C \leftarrow [A] + [B]$

can now be performed by the two-instruction sequence

```
Move  B,C
Add   A,C
```

In all the instructions given above, the source operands are specified first, followed by the destination. This order is used in the assembly language expressions for machine instructions in many computers. But there are also many computers in which the order of the source and destination operands is reversed. We will see examples of both orderings in Chapter 3. It is unfortunate that no single convention has been adopted by all manufacturers. In fact, even for a particular computer, its assembly language may use a different order for different instructions. In this chapter, we will continue to give the source operands first.

We have defined three- and two-address instructions. But, even two-address instructions will not normally fit into one word for usual word lengths and address sizes. Another possibility is to have machine instructions that specify only one memory operand. When a second operand is needed, as in the case of an Add instruction, it is understood implicitly to be in a unique location. A processor register, usually called the *accumulator,* may be used for this purpose. Thus, the *one-address* instruction

```
Add  A
```

means the following: Add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator. Let us also introduce the one-address instructions

```
Load  A
```

and

```
Store  A
```

The Load instruction copies the contents of memory location A into the accumulator, and the Store instruction copies the contents of the accumulator into memory location A. Using only one-address instructions, the operation $C \leftarrow [A] + [B]$ can be performed by executing the sequence of instructions

```
Load   A
Add    B
Store  C
```

Note that the operand specified in the instruction may be a source or a destination, depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Some early computers were designed around a single accumulator structure. Most modern computers have a number of general-purpose processor registers — typically 8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor. Because the number of registers is relatively small, only a few bits are needed to specify which register takes part in an operation. For example, for 32 registers, only 5 bits are needed. This is much less than the number of bits needed to give the address of a location in the memory. Because the use of registers allows faster processing and results in shorter instructions, registers are used to store data temporarily in the processor during processing.

Let R*i* represent a general-purpose register. The instructions

```
Load   A,Ri
Store  Ri,A
```

and

```
Add   A,Ri
```

are generalizations of the Load, Store, and Add instructions for the single-accumulator case, in which register R*i* performs the function of the accumulator. Even in these cases, when only one memory address is directly specified in an instruction, the instruction may not fit into one word.

When a processor has several general-purpose registers, many instructions involve only operands that are in the registers. In fact, in many modern processors, computations can be performed directly only on data held in processor registers. Instructions such as

```
Add   Ri,Rj
```

or

```
Add   Ri,Rj,Rk
```

are of this type. In both of these instructions, the source operands are the contents of registers R*i* and R*j*. In the first instruction, R*j* also serves as the destination register, whereas in the second instruction, a third register, R*k*, is used as the destination. Such instructions, where only register names are contained in the instruction, will normally fit into one word.

It is often necessary to transfer data between different locations. This is achieved with the instruction

```
Move   Source,Destination
```

which places a copy of the contents of Source into Destination. When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. Thus,

```
Move   A,Ri
```

is the same as

```
Load   A,Ri
```

and

```
Move   Ri,A
```

is the same as

```
Store   Ri,A
```

In this chapter, we will use Move instead of Load or Store.

In processors where arithmetic operations are allowed only on operands that are in processor registers, the C = A + B task can be performed by the instruction sequence

```
Move   A,Ri
Move   B,Rj
Add    Ri,Rj
Move   Rj,C
```

In processors where one operand may be in the memory but the other must be in a register, an instruction sequence for the required task would be

```
Move   A,Ri
Add    B,Ri
Move   Ri,C
```

The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor. Hence, a substantial increase in speed is achieved when several operations are performed in succession on data in processor registers without the need to copy data to or from the memory. When machine language programs are generated by compilers from high-level languages, it is important to minimize the frequency with which data is moved back and forth between the memory and processor registers.

We have discussed three-, two-, and one-address instructions. It is also possible to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a *pushdown stack*. In this case, the instructions are called *zero-address* instructions. The concept of a pushdown stack is introduced in Section 2.8, and a computer that uses this approach is discussed in Chapter 11.

2.4.4 INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING

In the preceding discussion of instruction formats, we used the task C ← [A] + [B] for illustration. Figure 2.8 shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand per instruction and has a number of processor registers. We assume that the word length is 32 bits and the memory is byte addressable. The three instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second and third instructions start at addresses $i + 4$ and $i + 8$. For simplicity, we also assume that a full memory address can be directly specified in a single-word instruction, although this is not usually possible for address space sizes and word lengths of current processors.

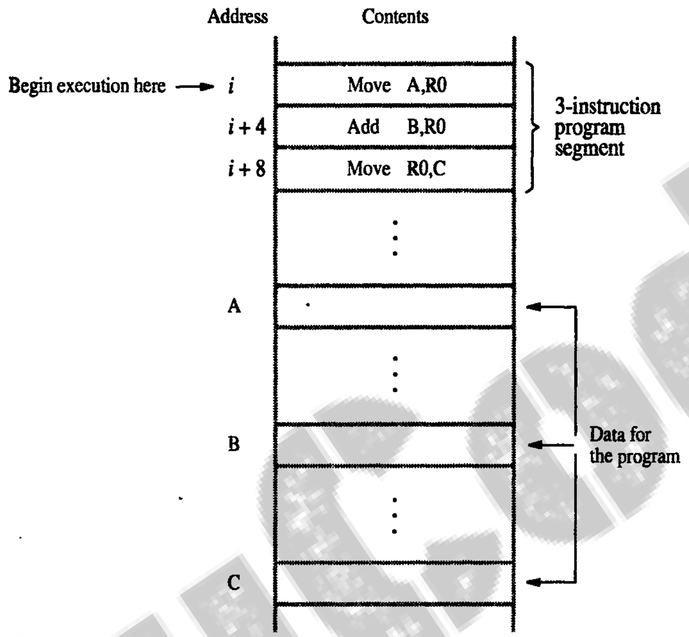


Figure 2.8 A program for C ← [A] + [B].

Let us consider how this program is executed. The processor contains a register called the *program counter* (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Move instruction at location $i + 8$ is executed, the PC contains the value $i + 12$, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin. In most processors, the execute phase itself is divided into a small number of distinct phases corresponding to fetching operands, performing the operation, and storing the result.

2.4.5 BRANCHING

Consider the task of adding a list of n numbers. The program outlined in Figure 2.9 is a generalization of the program in Figure 2.8. The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2, ..., NUMn, and a separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Add instructions, it is possible to place a single Add instruction in a program loop, as shown in Figure 2.10. The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch>0. During each pass through this loop, the address of

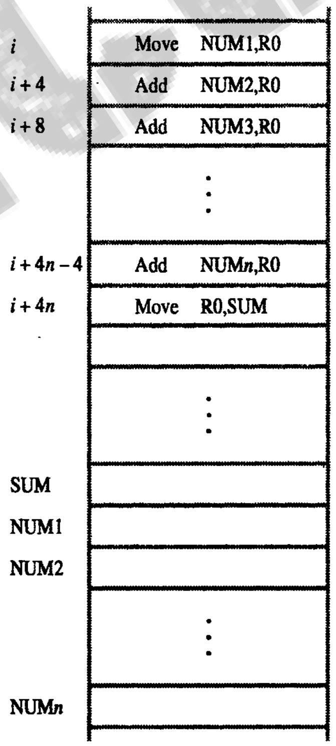


Figure 2.9 A straight-line program for adding n numbers.

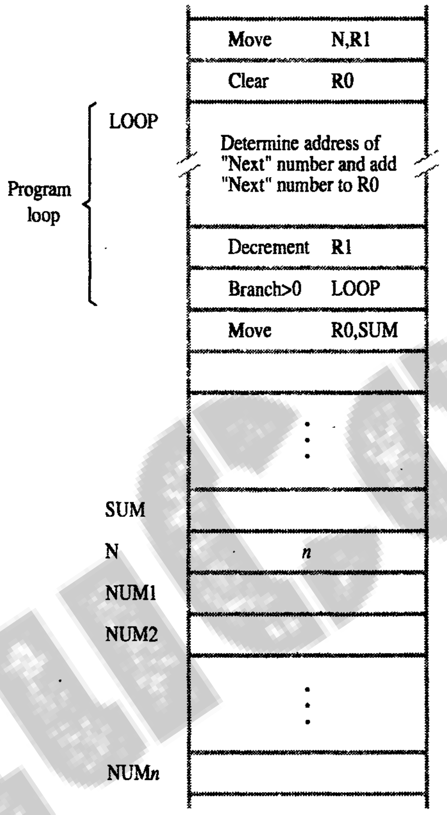


Figure 2.10 Using a loop to add n numbers.

the next list entry is determined, and that entry is fetched and added to R0. The address of an operand can be specified in various ways, as will be described in Section 2.5. For now, we concentrate on how to create and control a program loop.

Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction

Decrement R1

reduces the contents of R1 by 1 each time through the loop. (A similar type of operation is performed by an Increment instruction, which adds 1 to its operand.) Execution of the loop is repeated as long as the result of the decrement operation is greater than zero.

We now introduce *branch* instructions. This type of instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 2.10, the instruction

```
Branch>0    LOOP
```

(branch if greater than 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R0. At the end of the *n*th pass through the loop, the Decrement instruction produces a value of zero, and, hence, branching does not occur. Instead, the Move instruction is fetched and executed. It moves the final result from R0 into memory location SUM.

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

2.4.6 CONDITION CODES

The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called *condition code flags*. These flags are usually grouped together in a special processor register called the *condition code register* or *status register*. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The N and Z flags indicate whether the result of an arithmetic or logic operation is negative or zero. The N and Z flags may also be affected by instructions that transfer data, such as Move, Load, or Store. This makes it possible for a later conditional branch instruction to cause a branch based on the sign and value of the operand that was moved. Some computers also provide a special Test instruction that examines a value in a register or in the memory and sets or clears the N and Z flags accordingly.

The V flag indicates whether overflow has taken place. As explained in Section 2.1.4, overflow occurs when the result of an arithmetic operation is outside the range of values that can be represented by the number of bits available for the operands. The processor sets the V flag to allow the programmer to test whether overflow has occurred and branch to an appropriate routine that corrects the problem. Instructions such as BranchIfOverflow are provided for this purpose. Also, as we will see in Chapter 4, a program interrupt may occur automatically as a result of the V bit being set, and the operating system will resolve what to do.

The C flag is set to 1 if a carry occurs from the most significant bit position during an arithmetic operation. This flag makes it possible to perform arithmetic operations on operands that are longer than the word length of the processor. Such operations are used in multiple-precision arithmetic, which is discussed in Chapter 6.

The instruction Branch>0, discussed in Section 2.4.5, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. Many other conditional branch instructions are provided to enable a variety of conditions to be tested. The conditions are given as logic expressions involving the condition code flags.

In some computers, the condition code flags are affected automatically by instructions that perform arithmetic or logic operations. However, this is not always the case. A number of computers have two versions of an Add instruction, for example. One version, Add, does not affect the flags, but a second version, AddSetCC, does. This provides the programmer — and the compiler — with more flexibility when preparing programs for pipelined execution, as we will discuss in Chapter 8.

2.4.7 GENERATING MEMORY ADDRESSES

Let us return to Figure 2.10. The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in that block must refer to a different address during each pass. How are the addresses to be specified? The memory operand address cannot be given directly in a single Add instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, R*i*, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation, and many others like it, give rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same. We will discuss these in the next section.

2.5 ADDRESSING MODES

We have now seen some simple examples of assembly language programs. In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways. If we want to keep track of students' names, we can write them in a list. If we want to associate information with each name, for example to record telephone numbers or marks in various courses, we may organize this information in the form of a table. Programmers use organizations called *data structures* to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run. The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*. In this section we present the most important addressing modes found in modern processors. A summary is provided in Table 2.1.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

2.5.1 IMPLEMENTATION OF VARIABLES AND CONSTANTS

Variables and constants are the simplest data types and are found in almost every computer program. In assembly language, a variable is represented by allocating a register or a memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

The programs in Section 2.4 used only two addressing modes to access variables. We accessed an operand by specifying the name of the register or the address of the memory location where the operand is located. The precise definitions of these two modes are:

Register mode — The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

Absolute mode — The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called *Direct*.)

The instruction

```
Move    LOC,R2
```

uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

```
Integer A, B;
```

in a high-level language program will cause the compiler to allocate a memory location to each of the variables A and B. Whenever they are referenced later in the program, the compiler can generate assembly language instructions that use the Absolute mode to access these variables.

Next, let us consider the representation of constants. Address and data constants can be represented in assembly language using the Immediate mode.

Immediate mode — The operand is given explicitly in the instruction.

For example, the instruction

$$\text{Move} \quad 200_{immediate}, \text{R0}$$

places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

```
Move    #200,R0
```

Constant values are used frequently in high-level language programs. For example, the statement

$$A = B + 6$$

contains the constant 6. Assuming that A and B have been declared earlier as variables and may be accessed using the Absolute mode, this statement may be compiled as follows:

```
Move  B,R1
Add   #6,R1
Move  R1,A
```

Constants are also used in assembly language to increment a counter, test for some bit pattern, and so on.

2.5.2 INDIRECTION AND POINTERS

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the *effective address* (EA) of the operand.

Indirect mode — The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

We denote indirection by placing the name of the register or the memory address given in the instruction in parentheses as illustrated in Figure 2.11 and Table 2.1.

To execute the Add instruction in Figure 2.11*a*, the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in Figure 2.11*b*. In this case, the processor first reads the contents of memory location A, then requests a

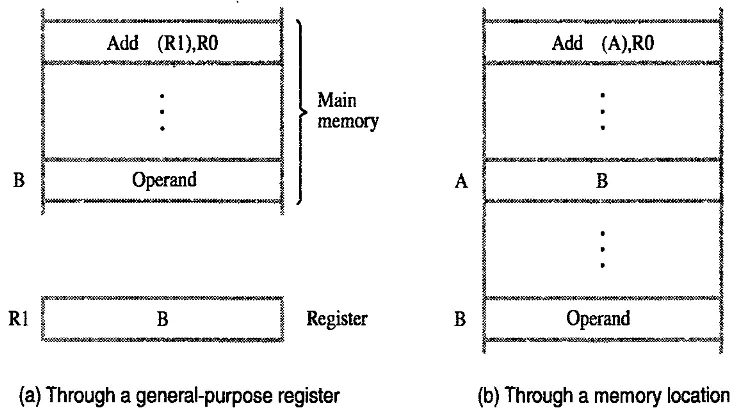

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure 2.11 Indirect addressing.

```
Address     Contents

            Move         N,R1        ⎫
            Move         #NUM1,R2    ⎬ Initialization
            Clear        R0          ⎭
LOOP        Add          (R2),R0
            Add          #4,R2
            Decrement    R1
            Branch>0     LOOP
            Move         R0,SUM
```

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

second read operation using the value B as an address to obtain the operand.

The register or memory location that contains the address of an operand is called a *pointer*. Indirection and the use of pointers are important and powerful concepts in programming. Consider the analogy of a treasure hunt: In the instructions for the hunt you may be told to go to a house at a given address. Instead of finding the treasure there, you find a note that gives you another address where you will find the treasure. By changing the note, the location of the treasure can be changed, but the instructions for the hunt remain the same. Changing the note is equivalent to changing the contents of a pointer in a computer program. For example, by changing the contents of register R1 or location A in Figure 2.11, the same Add instruction fetches different operands to add to register R0.

Let us now return to the program in Figure 2.10 for adding a list of numbers. Indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 2.12. Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value *n* from memory location N into R1 and uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop in Figure 2.12 implement the unspecified instruction block starting at LOOP in Figure 2.10. The first time through the loop, the instruction

```
Add   (R2),R0
```

fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Consider the C-language statement

```
A=*B;
```

where B is a pointer variable. This statement may be compiled into

```
Move   B,R1
Move   (R1),A
```

Using indirect addressing through memory, the same action can be achieved with

```
Move (B),A
```

Despite its apparent simplicity, indirect addressing through memory has proven to be of limited usefulness as an addressing mode, and it is seldom found in modern computers. We will see in Chapter 8 that an instruction that involves accessing the memory twice to get an operand is not well suited to pipelined execution.

Indirect addressing through registers is used extensively. The program in Figure 2.12 shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

2.5.3 INDEXING AND ARRAYS

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode — The effective address of the operand is generated by adding a constant value to the contents of a register.

The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*. We indicate the Index mode symbolically as

$$X(Ri)$$

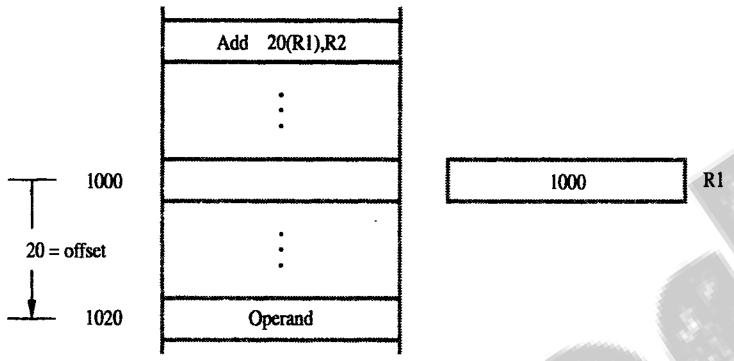
where X denotes the constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

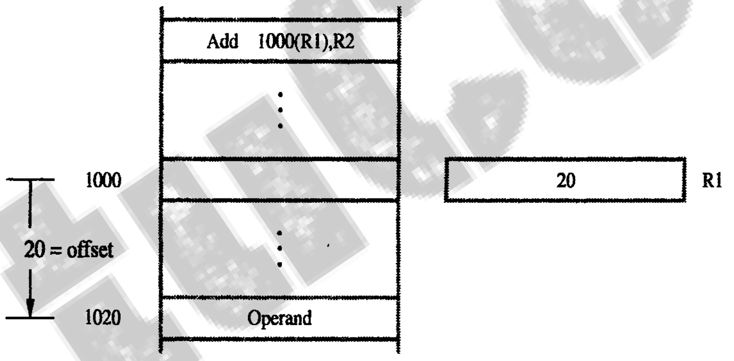
The contents of the index register are not changed in the process of generating the effective address.

In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value. The way in which a symbolic name is associated with a specific numerical value will be discussed in Section 2.6. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is usually represented by fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended (see Section 2.1.3) to the register length before being added to the contents of the register.

Figure 2.13 illustrates two ways of using the Index mode. In Figure 2.13*a*, the index register, R1, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found. An alternative use is illustrated in Figure 2.13*b*. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.



(a) Offset is given as a constant



(b) Offset is in the index register

Figure 2.13 Indexed addressing.

To see the usefulness of indexed addressing, consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores, beginning at location LIST, is structured as shown in Figure 2.14. A four-word memory block comprises a record that stores the relevant information for each student. Each record consists of the student's identification number (ID), followed by the scores the student earned on three tests. There are *n* students in the class, and the value *n* is stored in location N immediately in front of the list. The addresses given in the figure for the student IDs and test scores assume that the memory is byte addressable and that the word length is 32 bits.

We should note that the list in Figure 2.14 represents a two-dimensional array having *n* rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

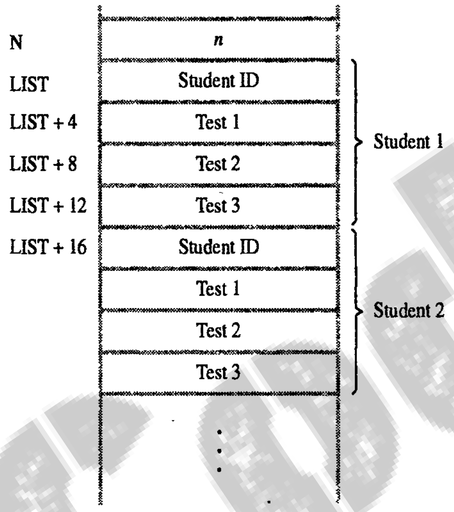


Figure 2.14 A list of students' marks.

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible program for this task is given in Figure 2.15. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.13*a* to access each of the three scores in a student's record. Register R0 is used as the index register. Before the loop is entered, R0 is set to point to the ID location of the first student record; thus, it contains the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R1, R2, and R3, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R0), 8(R0), and 12(R0). The index register R0 is then incremented by 16 to point to the ID location of the second student. Register R4, initialized to contain the value *n*, is decremented by 1 at the end of each pass through the loop. When the contents of R4 reach 0, all student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R1, R2, and R3, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R0, are not changed when it is used in the Index addressing mode to access the scores. The contents of R0 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

```
              Move        #LIST,R0
              Clear       R1
              Clear       R2
              Clear       R3
              Move        N,R4
   LOOP       Add         4(R0),R1
              Add         8(R0),R2
              Add         12(R0),R3
              Add         #16,R0
              Decrement   R4
              Branch>0    LOOP
              Move        R1,SUM1
              Move        R2,SUM2
              Move        R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

We have introduced the most basic form of indexed addressing. Several variations of this basic form provide for very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

$$(Ri,Rj)$$

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

As an example of where this flexibility may be useful, consider again the student record data structure shown in Figure 2.14. In the program in Figure 2.15, we used different index values in the three Add instructions at the beginning of the loop to access different test scores. Suppose each record contains a large number of items, many more than the three test scores of that example. In this case, we would need the ability to replace the three Add instructions with one instruction inside a second (nested) loop. Just as the successive starting locations of the records (the reference points) are maintained in the pointer register R0, offsets to the individual items relative to the contents of R0 could be maintained in another register. The contents of that register would be incremented in successive passes through the inner loop. (See Problem 2.9.)

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(Ri,Rj)$$

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri,Rj) part of the addressing mode. In other words, this mode implements a three-dimensional array.

2.5.4 RELATIVE ADDRESSING

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general-purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode — The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

```
Branch>0   LOOP
```

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Recall that during the execution of an instruction, the processor increments the PC to point to the next instruction. Most computers use this updated value in computing the effective address in the Relative mode. For example, suppose that the Relative mode is used to generate the branch target address LOOP in the Branch instruction of the program in Figure 2.12. Assume that the four instructions of the loop body, starting at LOOP, are located at memory locations 1000, 1004, 1008, and 1012. Hence, the updated contents of the PC at the time the branch target address is generated will be 1016. To branch to location LOOP (1000), the offset value needed is $X = -16$.

Assembly languages allow branch instructions to be written using labels to denote the branch target as shown in Figure 2.12. When the assembler program processes such an instruction, it computes the required offset value, -16 in this case, and generates the corresponding machine instruction using the addressing mode -16(PC).

2.5.5 ADDITIONAL MODES

So far we have discussed the five basic addressing modes — Immediate, Register, Absolute (Direct), Indirect, and Index — found in most computers. We have given a number of common versions of the Index mode, not all of which may be found in any one computer. Although these modes suffice for general computation, many computers provide additional modes intended to aid certain programming tasks. The two modes described next are useful for accessing data items in successive locations in the memory.

Autoincrement mode — The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

$$(\mathrm{R}i)+$$

Implicitly, the increment amount is 1 when the mode is given in this form. But in a byte addressable memory, this mode would only be useful in accessing successive bytes of some list. To access successive words in a byte-addressable memory with a 32-bit word length, the increment must be 4. Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand. Thus, the increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands. Since the size of the operand is usually specified as part of the operation code of an instruction, it is sufficient to indicate the Autoincrement mode as $(\mathrm{R}i)+$.

If the Autoincrement mode is available, it can be used in the first Add instruction in Figure 2.12 and the second Add instruction can be eliminated. The modified program is shown in Figure 2.16.

As a companion for the Autoincrement mode, another useful mode accesses the items of a list in the reverse order:

Autodecrement mode — The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

$$-(\mathrm{R}i)$$

```
                Move        N,R1          }
                Move        #NUM1,R2      } Initialization
                Clear       R0            }
 ──► LOOP       Add         (R2)+,R0
 │              Decrement   R1
 └───────────   Branch>0    LOOP
                Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

In this mode, operands are accessed in descending address order. The reader may wonder why the address is decremented before it is used in the Autodecrement mode and incremented after it is used in the Autoincrement mode. The main reason for this is given in Section 2.8, where we show how these two modes can be used together to implement an important data structure called a stack.

The actions performed by the Autoincrement and Autodecrement addressing modes can obviously be achieved by using two instructions, one to access the operand and the other to increment or decrement the register that contains the operand address. Combining the two operations in one instruction reduces the number of instructions needed to perform the desired task. However, we will show in Chapter 8 that it is not always advantageous to combine two operations in a single instruction.

2.6 ASSEMBLY LANGUAGE

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Move, Add, Increment, and Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics,* such as MOV, ADD, INC, and BR. Similarly, we use the notation R3 to refer to register 3, and LOC to refer to a memory location. A complete set of such symbolic names and rules for their use constitute a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics in the specification of complete instructions and programs is called the *syntax* of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler.* The assembler program is one of a collection of utility programs that are a part of the system software. The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk. At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a *source program,* and the assembled machine language program is called an *object program.* We will discuss how the assembler program works in Section 2.6.2. First, we present a few aspects of the assembly language itself.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower case letters. We will use capital letters to denote all names and labels in our examples in order to improve the readability of the text. For example, we will write a Move instruction as

```
MOVE    R0,SUM
```