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

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

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

1.1 COMPUTER TYPES

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

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

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

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

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

1.2 FUNCTIONAL UNITS

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

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

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

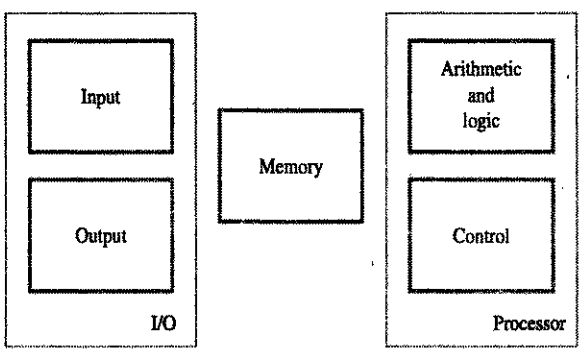


Figure 1.1 Basic functional units of a computer.

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

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

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

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

1.2.1 INPUT UNIT

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

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

1.2.2 MEMORY UNIT

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

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

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

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

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

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

1.2.3 ARITHMETIC AND LOGIC UNIT

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

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

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

1.2.4 OUTPUT UNIT

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

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

1.2.5 CONTROL UNIT

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

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

The operation of a computer can be summarized as follows:

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

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

1.3 BASIC OPERATIONAL CONCEPTS

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

```
Add   LOCA,R0
```

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

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

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

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

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

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

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

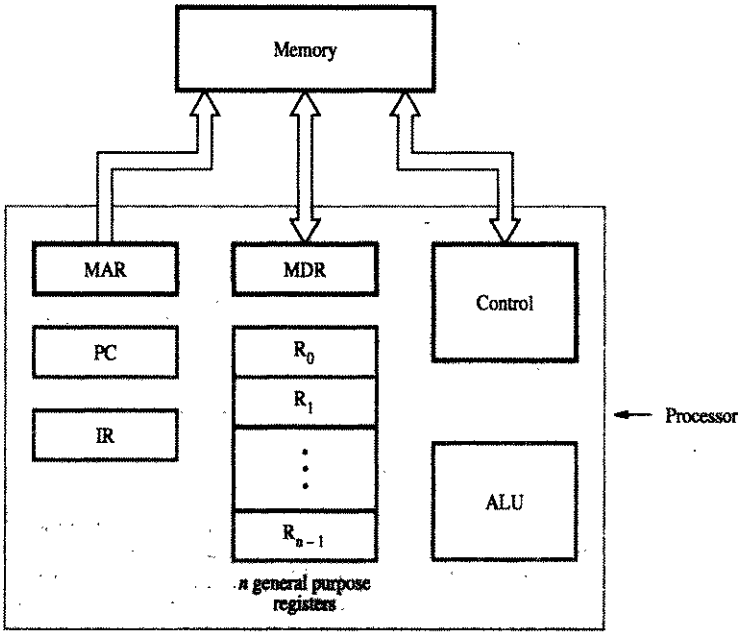


Figure 1.2 Connections between the processor and the memory.

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

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

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

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

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

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

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

1.4 BUS STRUCTURES

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

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

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

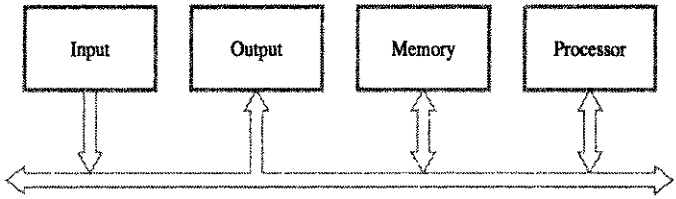


Figure 1.3 Single-bus structure.

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

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

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

1.5 SOFTWARE

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

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

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

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

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

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

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

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

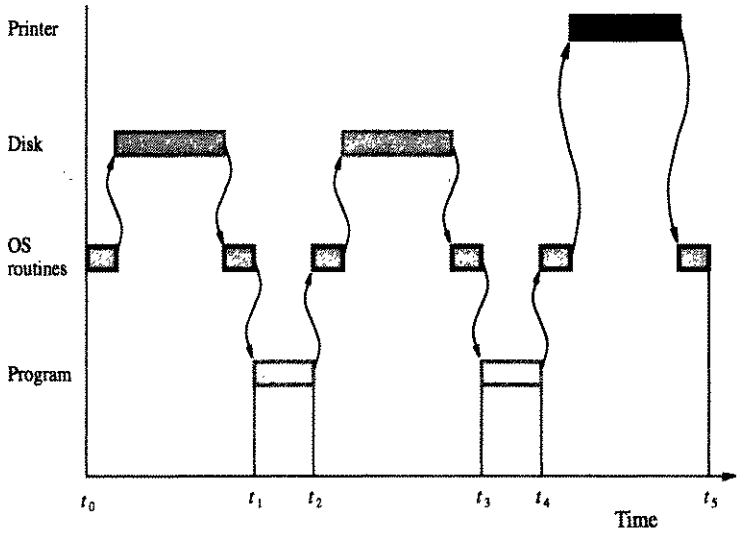


Figure 1.4 User program and OS routine sharing of the processor.

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

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

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

1.6 PERFORMANCE

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

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

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

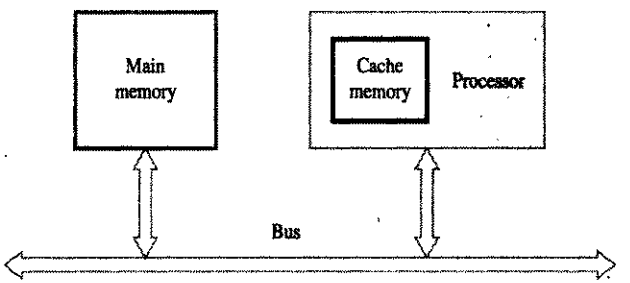


Figure 1.5 The processor cache.

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

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

1.6.1 PROCESSOR CLOCK

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

1.6.2 BASIC PERFORMANCE EQUATION

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

given by

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

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

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

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

1.6.3 PIPELINING AND SUPERSCALAR OPERATION

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

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

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

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

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

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

1.6.4 CLOCK RATE

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

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

1.6.5 INSTRUCTION SET: CISC AND RISC

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

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

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

1.6.6 COMPILER

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

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

1.6.7 PERFORMANCE MEASUREMENT

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

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

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

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

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

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

1.8 HISTORICAL PERSPECTIVE

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

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

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

1.8.1 THE FIRST GENERATION

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

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

1.8.2 THE SECOND GENERATION

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

1.8.3 THE THIRD GENERATION

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

1.8.4 THE FOURTH GENERATION

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

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

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

1.8.5 BEYOND THE FOURTH GENERATION

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

1.8.6 EVOLUTION OF PERFORMANCE

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

1.9 CONCLUDING REMARKS

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

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 ← [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,R*i*

Store R*i*,A

and

Add A,R*i*

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 R*i*,R*j*

or

Add R*i*,R*j*,R*k*

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,R*i*

is the same as

Load A,R*i*

and

Move R*i*,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.

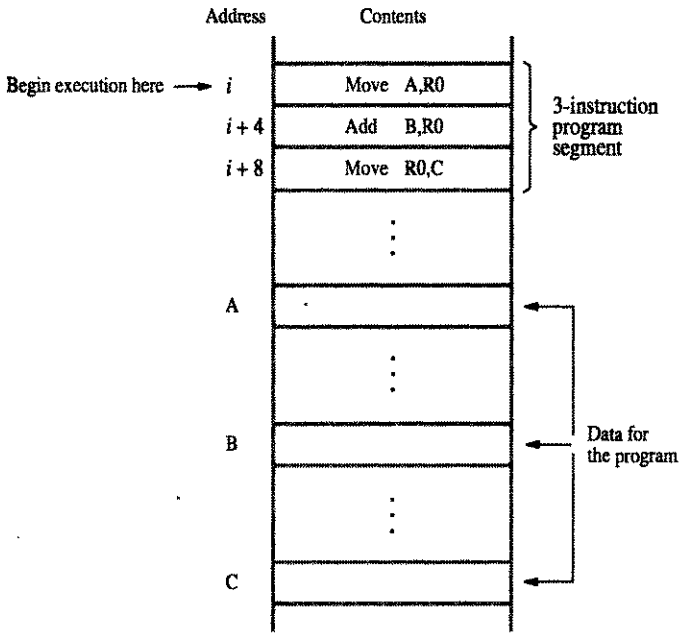


Figure 2.8 A program for C ← [A] + [B].

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

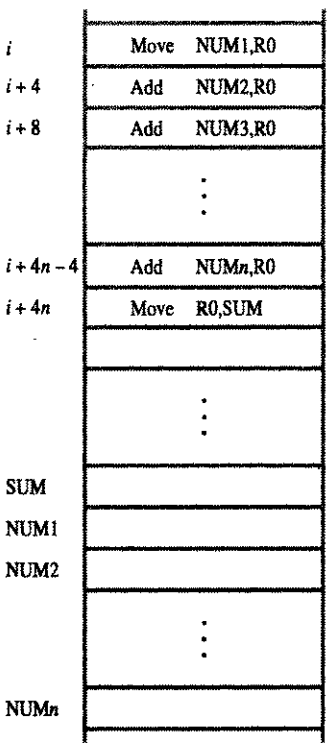


Figure 2.9 A straight-line program for adding n numbers.

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

Move $200_{immediate}$, 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.

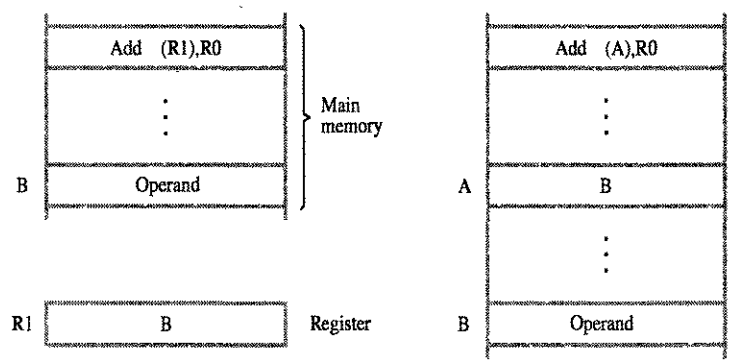
2.5.2 INDIRECTION AND POINTERS

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



(a) Through a general-purpose register (b) Through a memory location

Figure 2.11 Indirect addressing.

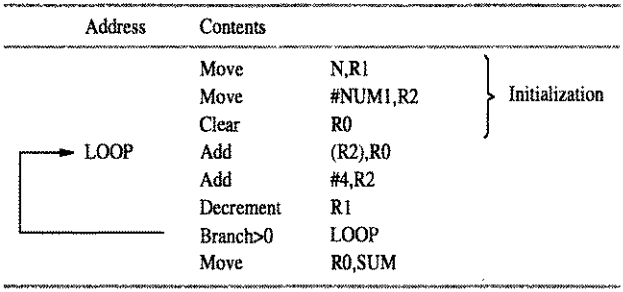


Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

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

$$X(Ri)$$

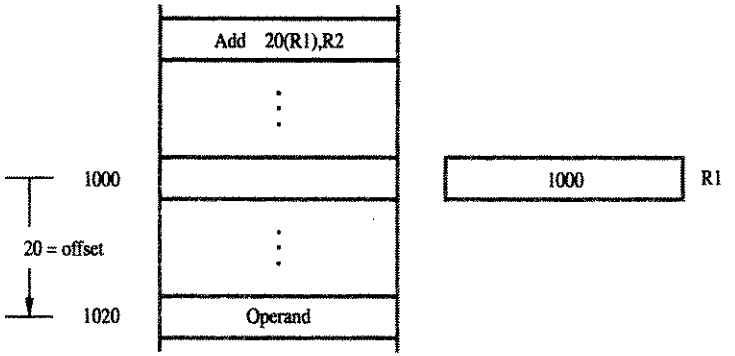
where X denotes the constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

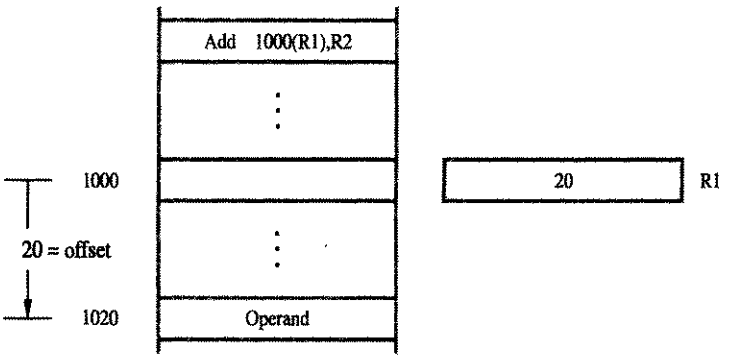
The contents of the index register are not changed in the process of generating the effective address.

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.

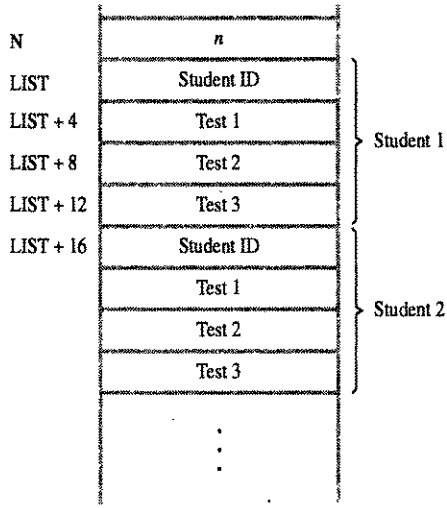


Figure 2.14 A list of students' marks.

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.

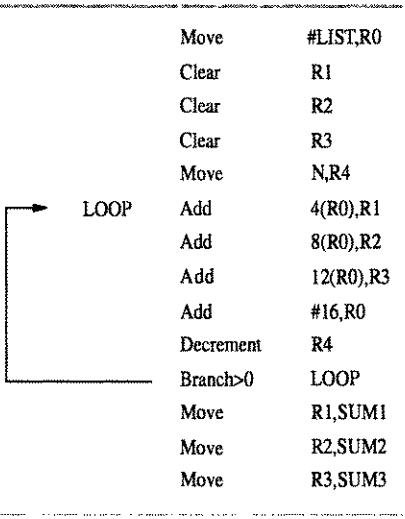


Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

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 R*i* and R*j*. 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 (R*i*,R*j*) 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 R*i*.

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

the contents of various processor registers and memory locations. We consider program debugging in more detail in Chapter 4.

2.6.3 NUMBER NOTATION

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

```
ADD   #93,R1
```

or as a binary number identified by a prefix symbol such as a percent sign, as in

```
ADD   #%01011101,R1
```

Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The hex notation is a direct extension of the BCD code given in Appendix E. The first ten patterns 0000, 0001, . . . , 1001, are represented by the digits 0, 1, . . . , 9, as in BCD. The remaining six 4-bit patterns, 1010, 1011, . . . , 1111, are represented by the letters A, B, . . . , F. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by a dollar sign prefix. Thus, we would write

```
ADD   #$5D,R1
```

2.7 BASIC INPUT/OUTPUT OPERATIONS

Previous sections in this chapter described machine instructions and addressing modes. We have assumed that the data on which these instructions operate are already stored in the memory. We now examine the means by which data are transferred between the memory of a computer and the outside world. Input/Output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer. This subject is discussed in detail in Chapter 4. Here, we introduce a few basic ideas.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as *program-controlled I/O*. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can

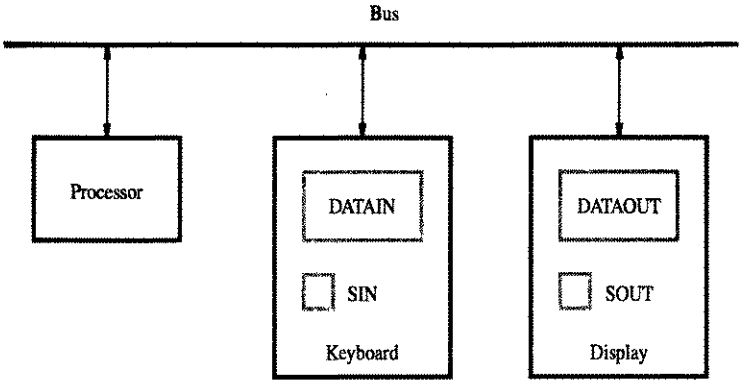


Figure 2.19 Bus connection for processor, keyboard, and display.

execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown in Figure 2.19. The action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Consider the problem of moving a character code from the keyboard to the processor. Striking a key stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN, as shown in Figure 2.19. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1. The buffer registers DATAIN and DATAOUT and the status flags SIN

and SOUT are part of circuitry commonly known as a *device interface*. The circuitry for each device is connected to the processor via a bus, as indicated in Figure 2.19.

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations:

```
READWAIT   Branch to READWAIT if SIN = 0
           Input from DATAIN to R1
```

The Branch operation is usually implemented by two machine instructions. The first instruction tests the status flag and the second performs the branch. Although the details vary from computer to computer, the main idea is that the processor monitors the status flag by executing a short *wait loop* and proceeds to transfer the input data when SIN is set to 1 as a result of a key being struck. The Input operation resets SIN to 0.

An analogous sequence of operations is used for transferring output to the display. An example is

```
WRITEWAIT  Branch to WRITEWAIT if SOUT = 0
           Output from R1 to DATAOUT
```

Again, the Branch operation is normally implemented by two machine instructions. The wait loop is executed repeatedly until the status flag SOUT is set to 1 by the display when it is free to receive a character. The Output operation transfers a character from R1 to DATAOUT to be displayed, and it clears SOUT to 0.

We assume that the initial state of SIN is 0 and the initial state of SOUT is 1. This initialization is normally performed by the device control circuits when the devices are placed under computer control before program execution begins.

Until now, we have assumed that the addresses issued by the processor to access instructions and operands always refer to memory locations. Many computers use an arrangement called *memory-mapped I/O* in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT. Thus, no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the processor using instructions that we have already discussed, such as Move, Load, or Store. For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

```
MoveByte   DATAIN,R1
```

Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction

```
MoveByte   R1,DATAOUT
```

The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively. The MoveByte operation code signifies that the operand size is a byte, to distinguish it from the operation code

Move that has been used for word operands. We have established that the two data buffers in Figure 2.19 may be addressed as if they were two memory locations. It is possible to deal with the status flags SIN and SOUT in the same way, by assigning them distinct addresses. However, it is more common to include SIN and SOUT in *device status* registers, one for each of the two devices. Let us assume that bit b_3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively. The read operation just described may now be implemented by the machine instruction sequence

```
READWAIT  Testbit    #3,INSTATUS
          Branch=0   READWAIT
          MoveByte   DATAIN,R1
```

The write operation may be implemented as

```
WRITEWAIT Testbit    #3,OUTSTATUS
          Branch=0   WRITEWAIT
          MoveByte   R1,DATAOUT
```

The Testbit instruction tests the state of one bit in the destination location, where the bit position to be tested is indicated by the first operand. If the bit tested is equal to 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is ready, that is, when the bit tested becomes equal to 1, the data are read from the input buffer or written into the output buffer.

The program shown in Figure 2.20 uses these two operations to read a line of characters typed at a keyboard and send them out to a display device. As the characters are read in, one by one, they are stored in a data area in the memory and then echoed

```
        Move        #LOC,R0          Initialize pointer register R0 to point to the
                                       address of the first location in memory
                                       where the characters are to be stored.
READ    TestBit     #3,INSTATUS      Wait for a character to be entered
        Branch=0    READ               in the keyboard buffer DATAIN.
        MoveByte    DATAIN,(R0)      Transfer the character from DATAIN into
                                       the memory (this clears SIN to 0).
ECHO    TestBit     #3,OUTSTATUS     Wait for the display to become ready.
        Branch=0    ECHO
        MoveByte    (R0),DATAOUT     Move the character just read to the display
                                       buffer register (this clears SOUT to 0).
        Compare     #CR,(R0)+        Check if the character just read is CR
                                       (carriage return). If it is not CR, then
        Branch≠0    READ               branch back and read another character.
                                       Also, increment the pointer to store the
                                       next character.
```

Figure 2.20 A program that reads a line of characters and displays it.

back out to the display. The program finishes when the carriage return character, CR, is read, stored, and sent to the display. The address of the first byte location of the memory data area where the line is to be stored is LOC. Register R0 is used to point to this area, and it is initially loaded with the address LOC by the first instruction in the program. R0 is incremented for each character read and displayed by the Autoincrement addressing mode used in the Compare instruction.

Program-controlled I/O requires continuous involvement of the processor in the I/O activities. Almost all of the execution time for the program in Figure 2.20 is accounted for in the two wait loops, while the processor waits for a character to be struck or for the display to become available. It is desirable to avoid wasting processor execution time in this situation. Other I/O techniques, based on the use of interrupts, may be used to improve the utilization of the processor. Such techniques will be discussed in Chapter 4.

2.8 STACKS AND QUEUES

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. This section will describe stacks, as well as a closely related data structure called a queue.

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A *stack* is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, *last-in–first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 2.21 shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and −28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the *stack pointer* (SP). It could be one of the general-purpose registers or a register dedicated to this function. If we assume a

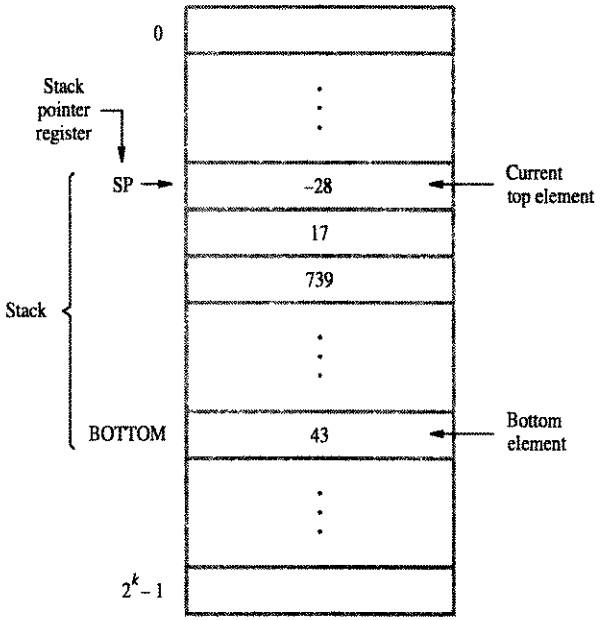


Figure 2.21 A stack of words in the memory.

byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract  #4,SP
Move      NEWITEM,(SP)
```

where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move. The pop operation can be implemented as

```
Move  (SP),ITEM
Add   #4,SP
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.22 shows the effect of each of these operations on the stack in Figure 2.21.

If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction

```
Move  NEWITEM,-(SP)
```

2.10 ADDITIONAL INSTRUCTIONS

So far, we have introduced the following instructions: Move, Load, Store, Clear, Add, Subtract, Increment, Decrement, Branch, Testbit, Compare, Call, and Return. These 13 instructions, along with the addressing modes in Table 2.1, have allowed us to write routines to illustrate machine instruction sequencing, including branching and the subroutine structure. We also illustrated the basic memory-mapped I/O operations.

Even this small set of instructions has a number of redundancies. The Load and Store instructions can be replaced by Move, and the Increment and Decrement instructions can be replaced by Add and Subtract, respectively. Also, Clear can be replaced by a Move instruction containing an immediate operand of zero. Therefore, only 8 instructions would have been sufficient for our purposes. But, it is not unusual to have some redundancy in practical machine instruction sets. Certain simple operations can usually be accomplished in a number of different ways. Some alternatives may be more efficient than others. In this section we introduce a few more important instructions that are found in most instruction sets.

2.10.1 LOGIC INSTRUCTIONS

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits, as described in Appendix A. It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel. For example, the instruction

```
Not  dst
```

complements all bits contained in the destination operand, changing 0s to 1s, and 1s to 0s. In Section 2.1.1, we saw that adding 1 to the 1's-complement of a signed positive number forms the negative version in 2's-complement representation. For example, in Figure 2.1, +3 (0011) is converted to −3 (1101) by adding 1 to the 1's-complement of 0011. If 3 is contained in register R0, the instructions

```
Not  R0
Add  #1,R0
```

achieve the conversion. Many computers have a single instruction

```
Negate  R0
```

that accomplishes the same thing.

Now consider an application for the logic instruction And, which performs the bit-wise AND operation on the source and destination operands. Suppose that four ASCII characters are contained in the 32-bit register R0. In some task, we wish to determine if the leftmost character is Z. If it is, a conditional branch to YES is to be made. From Appendix E, we find that the ASCII code for Z is 01011010, which is expressed in

hexadecimal notation as 5A. The three-instruction sequence

```
And        #$FF000000,R0
Compare    #$5A000000,R0
Branch=0   YES
```

implements the desired action. The And instruction clears all bits in the rightmost three character positions of R0 to zero, leaving the leftmost character unchanged. This is the result of using an immediate source operand that has eight 1s at its left end, and 0s in the 24 bits to the right. The Compare instruction compares the remaining character at the left end of R0 with the binary representation for the character Z. The Branch instruction causes a branch to YES if there is a match.

The And instruction is often used in practical programming tasks where all bits of an operand except for some specified field are to be cleared to 0. In our example, the leftmost eight bits of R0 constitute the specified field.

2.10.2 SHIFT AND ROTATE INSTRUCTIONS

There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use a logical shift. For a number, we use an arithmetic shift, which preserves the sign of the number.

Logical Shifts

Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a logical left shift instruction is

```
LShiftL   count,dst
```

The count operand may be given as an immediate operand, or it may be contained in a processor register. To complete the description of the left shift operation, we need to specify the bit values brought into the vacated positions at the right end of the destination operand, and to determine what happens to the bits shifted out of the left end. Vacated positions are filled with zeros, and the bits shifted out are passed through the Carry flag, C, and then dropped. Involving the C flag in shifts is useful in performing arithmetic operations on large numbers that occupy more than one word. Figure 2.30*a* shows an example of shifting the contents of register R0 left by two bit positions. The logical shift right instruction, LShiftR, works in the same manner except that it shifts to the right. Figure 2.30*b* illustrates this operation.

Digit-Packing Example

Consider the following short task that illustrates the use of both shift operations and logic operations. Suppose that two decimal digits represented in ASCII code are located

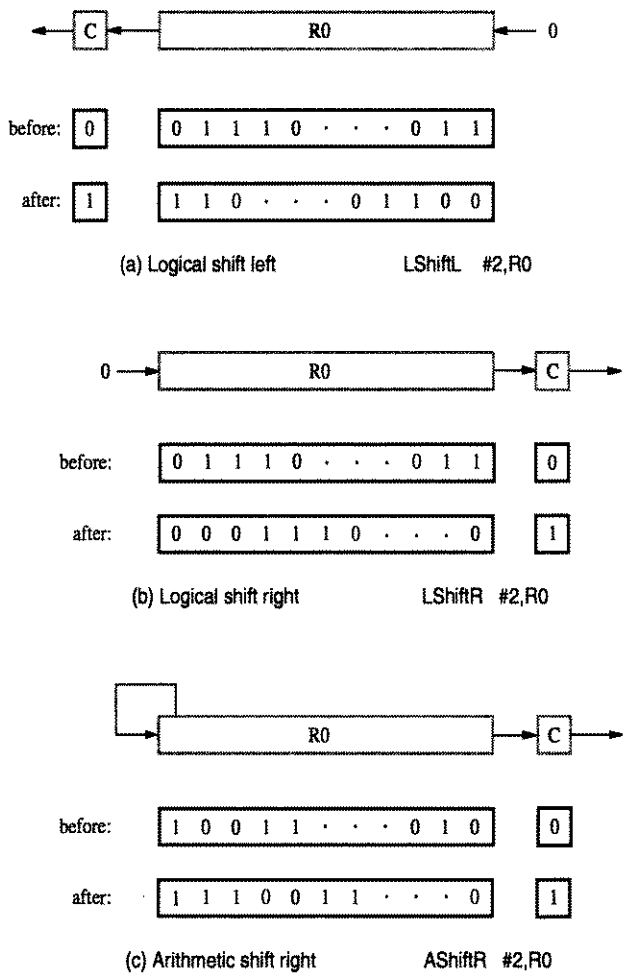


Figure 2.30 Logical and arithmetic shift instructions.

in memory at byte locations LOC and LOC + 1. We wish to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in *packed-BCD* format. Tables E.1 and E.2 in Appendix E show that the rightmost four bits of the ASCII code for a decimal digit correspond to the BCD code for the digit. Hence, the required task is to extract the low-order four bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED.

The instruction sequence shown in Figure 2.31 accomplishes the task using register R0 as a pointer to the ASCII characters in memory, and using registers R1 and R2 to

```
Move       #LOC,R0      R0 points to data.
MoveByte   (R0)+,R1     Load first byte into R1.
LShiftL    #4,R1        Shift left by 4 bit positions.
MoveByte   (R0),R2      Load second byte into R2.
And        #$F,R2       Eliminate high-order bits.
Or         R1,R2        Concatenate the BCD digits.
MoveByte   R2,PACKED    Store the result.
```

Figure 2.31 A routine that packs two BCD digits.

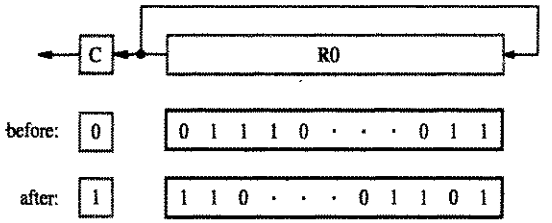
develop the BCD digit codes. When a MoveByte instruction transfers a byte between memory and a 32-bit processor register, we assume that the byte is located in the rightmost eight bit positions of the register. The And instruction is used to mask out all but the four rightmost bits in R2. Note that the immediate source operand is written as $F, which, interpreted as a 32-bit pattern, has 28 zeros in the most-significant bit positions.

Arithmetic Shifts

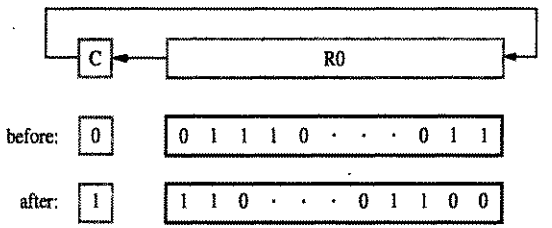
A study of the 2's-complement binary number representation in Figure 2.1 reveals that shifting a number one bit position to the left is equivalent to multiplying it by 2; and shifting it to the right is equivalent to dividing it by 2. Of course, overflow might occur on shifting left, and the remainder is lost in shifting right. Another important observation is that on a right shift the sign bit must be repeated as the fill-in bit for the vacated position. This requirement on right shifting distinguishes arithmetic shifts from logical shifts in which the fill-in bit is always 0. Otherwise, the two types of shifts are very similar. An example of an arithmetic right shift, AShiftR, is shown in Figure 2.30c. The arithmetic left shift is exactly the same as the logical left shift.

Rotate Operations

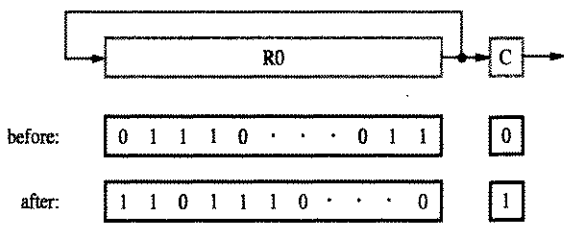
In the shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry flag C. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Two versions of both the left and right rotate instructions are usually provided. In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag. Figure 2.32 shows the left and right rotate operations with and without the C flag being included in the rotation. Note that when the C flag is not included in the rotation, it still retains the last bit shifted out of the end of the register. The mnemonics RotateL, RotateLC, RotateR, and RotateRC, denote the instructions that perform the rotate operations. The main use for Rotate instructions is in algorithms for performing arithmetic operations other than addition and subtraction, which we will encounter in Chapter 6.



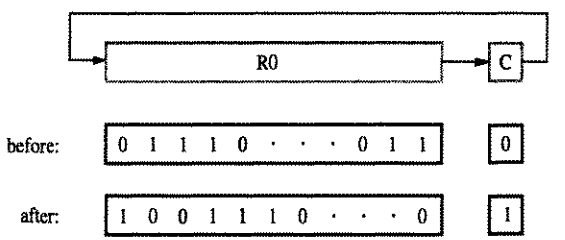
(a) Rotate left without carry RotateL #2,R0



(b) Rotate left with carry RotateLC #2,R0



(c) Rotate right without carry RotateR #2,R0



(d) Rotate right with carry RotateRC #2,R0

Figure 2.32 Rotate instructions.