BEC306C COMPUTER ORGANIZATION AND ARCHITECTURE

Module-1

Basic Structure of Computers: Computer Types, Functional Units, Basic Operational Concepts, Bus Structures, Software, Performance -Processor Clock, Basic Performance Equation (upto1.6.2of Chap1of Text).

Machine Instructions and Programs: Numbers, Arithmetic Operations and Characters, IEEE standard for Floating Point Numbers, Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing (up to 2.4.6 of Chap 2 and 6.7.1 of Chap 6 of Text).

MODULE QUESTION BANK

1. With a neat diagram, describe the functional units of a computer. Give few examples for I/O devices.
2. With a neat diagram, explain the basic operational concept of a computer.
3. With a neat diagram, discuss the operational concepts in a computer highlighting the role of PC, MAR, MDR, and IR.
4. Explain the bus structure
5. Explain the single bus structure of computers
6. Explain system software functions in a computer
7. What is an operating system? Explain the user program and OS routine sharing the processor.
8. Briefly explain different key parameters affecting the processor performance.
9. Explain the basic performance equation of a computer
10. List out and explain the three systems used for representing signed numbers and also brief about the modular number system concept.
11. Perform subtraction on the following pairs of numbers using 5-digit signed 2's complement format. Indicate overflow in each case i) +10 and -8 ii) +12nad +9 iii) -15 and -9 iv) -114 and +5
12. Discuss IEEE standards for single-precision and double-precision floating point numbers, with standard notations.
13. Represent 85.125 in IEEE floating point single precision.
14. Explain the memory operations with examples.
15. Define byte addressability, Big-endian, and Little-endian assignments.
16. Distinguish between Big-endian and Little-endian memory assignments. With a neat sketch, show how the number 26789435 is stored using these methods.
17. With examples, explain i) Three address ii) Two address iii) One address and iv) zero address instructions.
18. Develop an Assembly Language Program for the expression Y=Ax2+BCx+D using 3-address, 2-address, and 1-address instruction formats. Assume A, B, C, D, and Y as memory locations and x as intermediate data.
19. Illustrate instruction and instruction sequencing with an example
20. Explain condition codes with examples.

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

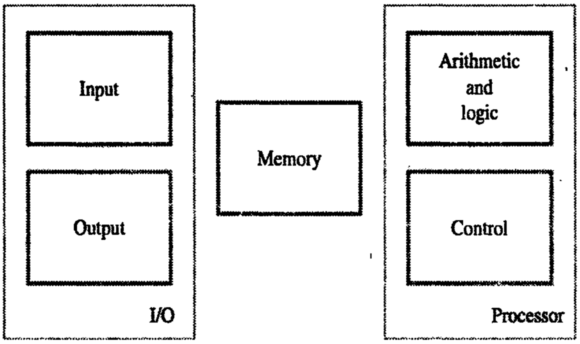


Figure 1.1 Basic functional units of a computer.

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,

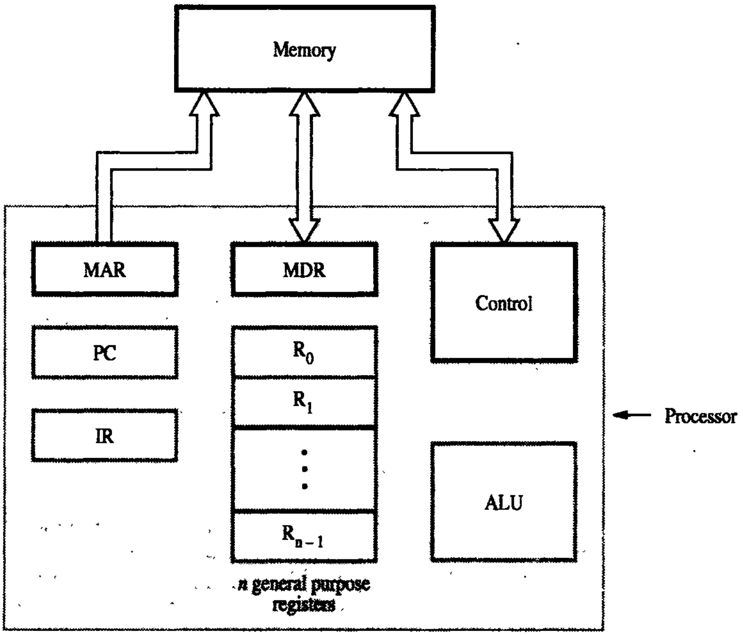


Figure 1.2 Connections between the processor and the memory.

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

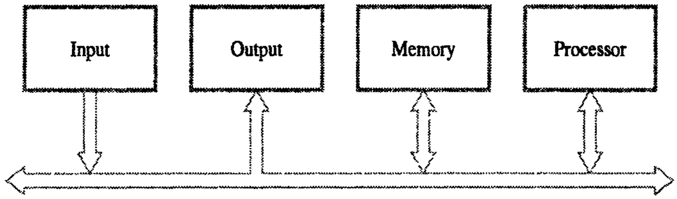


Figure 1.3 Single-bus structure.

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

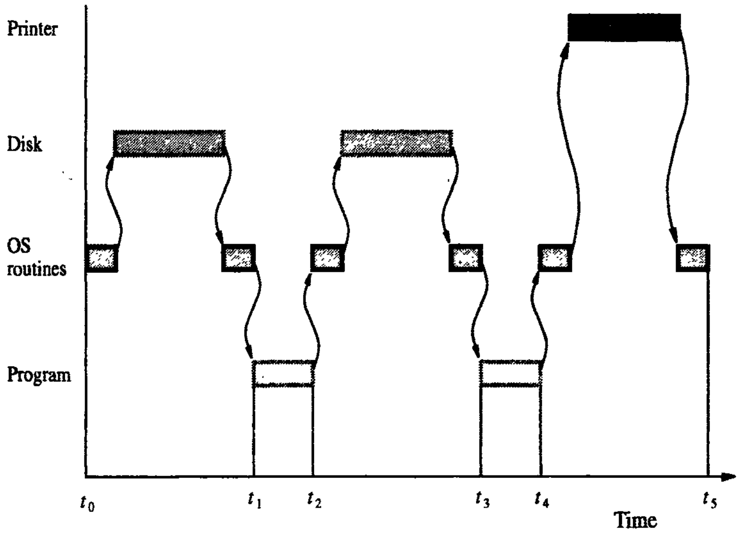


Figure 1.4 User program and OS routine sharing of the processor.

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

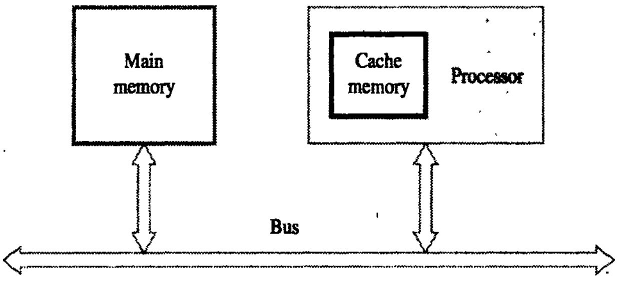


Figure 1.5 The processor cache.

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

$$T = \frac{N \times S}{R} \tag{1.1}$$

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,

2.1 NUMBERS, ARITHMETIC OPERATIONS, AND CHARACTERS

Computers are built using logic circuits that operate on information represented by two-valued electrical signals (see Appendix A). We label the two values as 0 and 1; and we define the amount of information represented by such a signal as a *bit* of information, where bit stands for *binary digit*. The most natural way to represent a number in a computer system is by a string of bits, called a binary number. A text character can also be represented by a string of bits called a character code.

We will first describe binary number representations and arithmetic operations on these numbers, and then describe character representations.

2.1.1 NUMBER REPRESENTATION

Consider an n-bit vector

$$B = b_{n-1} \dots b_1 b_0$$

where $b_i = 0$ or 1 for $0 \leq i \leq n-1$. This vector can represent unsigned integer values V in the range 0 to $2^n - 1$, where

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

We obviously need to represent both positive and negative numbers. Three systems are used for representing such numbers:

- Sign-and-magnitude
- 1's-complement
- 2's-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Figure 2.1 illustrates all three representations using 4-bit numbers. Positive values have identical representations in all systems, but negative values have different representations. In the *sign-and-magnitude* system, negative values are represented by changing the most significant bit (b_3 in Figure 2.1) from 0 to 1 in the B vector of the corresponding positive value. For example, +5 is represented by 0101, and −5 is represented by 1101. In *1's-complement* representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for −3 is obtained by complementing each bit in the vector 0011 to yield 1100. Clearly, the same operation, bit complementing, is done in converting a negative number to the corresponding positive value. Converting either way is referred to as forming the 1's-complement of a given number. The operation of forming the 1's-complement of a given number is equivalent to subtracting that number from $2^n - 1$, that is, from 1111 in the case of the 4-bit numbers in Figure 2.1. Finally, in the *2's-complement* system, forming the 2's-complement of a number is done by subtracting that number from 2^n.

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Figure 2.1 Binary, signed-integer representations.

Hence, the 2's-complement of a number is obtained by adding 1 to the 1's-complement of that number.

Note that there are distinct representations for +0 and −0 in both the sign-and-magnitude and 1's-complement systems, but the 2's-complement system has only one representation for 0. For 4-bit numbers, the value −8 is representable in the 2's-complement system but not in the other systems. The sign-and-magnitude system seems the most natural, because we deal with sign-and-magnitude decimal values in manual computations. The 1's-complement system is easily related to this system, but the 2's-complement system seems unnatural. However, we will show in Section 2.1.3 that the 2's-complement system yields the most efficient way to carry out addition and subtraction operations. It is the one most often used in computers.

2.1.2 ADDITION OF POSITIVE NUMBERS

Consider adding two 1-bit numbers. The results are shown in Figure 2.2. Note that the sum of 1 and 1 requires the 2-bit vector 10 to represent the value 2. We say that the *sum* is 0 and the *carry-out* is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) end of the bit vectors, propagating carries toward the high-order (left) end.

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline \underset{\substack{\uparrow \\ \text{Carry-out}}}{1}0 \end{array}$$

Figure 2.2 Addition of 1-bit numbers.

2.1.3 ADDITION AND SUBTRACTION OF SIGNED NUMBERS

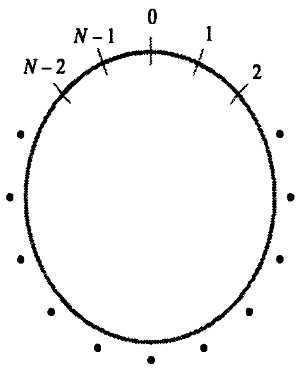
We introduced three systems for representing positive and negative numbers, or, simply, signed numbers. These systems differ only in the way they represent negative values. Their relative merits from the standpoint of ease of performing arithmetic operations can be summarized as follows: The sign-and-magnitude system is the simplest representation, but it is also the most awkward for addition and subtraction operations. The 1's-complement method is somewhat better. The 2's-complement system is the most efficient method for performing addition and subtraction operations.

To understand 2's-complement arithmetic, consider addition modulo N (written as mod N). A helpful graphical device for the description of addition mod N of positive integers is a circle with the N values, 0 through $N-1$, marked along its perimeter, as shown in Figure 2.3*a*. Consider the case $N = 16$. The operation $(7+4)$ mod 16 yields the value 11. To perform this operation graphically, locate 7 on the circle and then move 4 units in the clockwise direction to arrive at the answer 11. Similarly, $(9+14)$ mod $16 = 7$; this is modeled on the circle by locating 9 and moving 14 units in the clockwise direction to arrive at the answer 7. This graphical technique works for the computation of $(a+b)$ mod 16 for any positive numbers a and b, that is, to perform addition, locate a and move b units in the clockwise direction to arrive at $(a+b)$ mod 16.

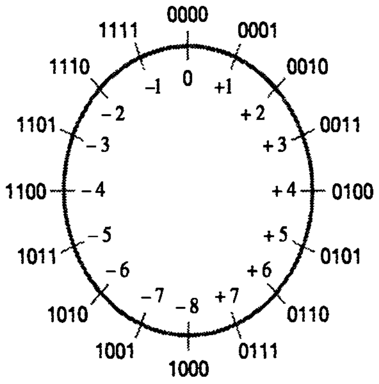
Now consider a different interpretation of the mod 16 circle. Let the values 0 through 15 be represented by the 4-bit binary vectors 0000, 0001, ..., 1111, according to the binary number system. Then reinterpret these binary vectors to represent the signed numbers from -8 through $+7$ in the 2's-complement method (see Figure 2.1), as shown in Figure 2.3*b*.

Let us apply the mod 16 addition technique to the simple example of adding $+7$ to -3. The 2's-complement representation for these numbers is 0111 and 1101, respectively. To add these numbers, locate 0111 on the circle in Figure 2.3*b*. Then move 1101 (13) steps in the clockwise direction to arrive at 0100, which yields the correct answer of $+4$. If we perform this addition by adding bit pairs from right to left, we obtain

$$\begin{array}{r} 0\,1\,1\,1 \\ +\,1\,1\,0\,1 \\ \hline \underset{\substack{\uparrow \\ \text{Carry-out}}}{1}\ 0\,1\,0\,0 \end{array}$$



(a) Circle representation of integers mod N



(b) Mod 16 system for 2's-complement numbers

Figure 2.3 Modular number systems and the 2's-complement system.

Note that if we ignore the carry-out from the fourth bit position in this addition, we obtain the correct answer. In fact, this is always the case. Ignoring this carry-out is a natural result of using mod N arithmetic. As we move around the circle in Figure 2.3*b*, the value next to 1111 would normally be 10000. Instead, we go back to the value 0000.

We now state the rules governing the addition and subtraction of n-bit signed numbers using the 2's-complement representation system.

1. To *add* two numbers, add their n-bit representations, ignoring the carry-out signal from the *most significant bit* (MSB) position. The sum will be the algebraically correct value in the 2's-complement representation as long as the answer is in the range -2^{n-1} through $+2^{n-1} - 1$.

2. To *subtract* two numbers X and Y, that is, to perform $X - Y$, form the 2's-complement of Y and then add it to X, as in rule 1. Again, the result will be the algebraically correct value in the 2's-complement representation system if the answer is in the range -2^{n-1} through $+2^{n-1} - 1$.

Figure 2.4 shows some examples of addition and subtraction. In all these 4-bit examples, the answers fall into the representable range of -8 through $+7$. When answers do not fall within the representable range, we say that arithmetic overflow has occurred. The next section discusses such situations. The four addition operations (*a*) through (*d*) in Figure 2.4 follow rule 1, and the six subtraction operations (*e*) through (*j*) follow rule 2. The subtraction operation requires the subtrahend (the bottom value) to be

```
(a)     0010     (+2)          (b)      0100     (+4)
      + 0011     (+3)                 + 1010     (-6)
      ------     ----                 ------     ----
        0101     (+5)                   1110     (-2)

(c)     1011     (-5)          (d)      0111     (+7)
      + 1110     (-2)                 + 1101     (-3)
      ------     ----                 ------     ----
        1001     (-7)                   0100     (+4)

(e)     1101     (-3)                   1101
      - 1001     (-7)          ⇒      + 0111
      ------     ----                 ------     ----
                                        0100     (+4)

(f)     0010     (+2)                   0010
      - 0100     (+4)          ⇒      + 1100
      ------     ----                 ------     ----
                                        1110     (-2)

(g)     0110     (+6)                   0110
      - 0011     (+3)          ⇒      + 1101
      ------     ----                 ------     ----
                                        0011     (+3)

(h)     1001     (-7)                   1001
      - 1011     (-5)          ⇒      + 0101
      ------     ----                 ------     ----
                                        1110     (-2)

(i)     1001     (-7)                   1001
      - 0001     (+1)          ⇒      + 1111
      ------     ----                 ------     ----
                                        1000     (-8)

(j)     0010     (+2)                   0010
      - 1101     (-3)          ⇒      + 0011
      ------     ----                 ------     ----
                                        0101     (+5)
```

Figure 2.4 2's-complement add and subtract operations.

2's-complemented. This operation is done in exactly the same manner for both positive and negative numbers.

We often need to represent a number in the 2's-complement system by using a number of bits that is larger than some given size. For a positive number, this is achieved by adding 0s to the left. For a negative number, the leftmost bit, which is the sign bit, is a 1, and a longer number with the same value is obtained by replicating the sign bit to the left as many times as desired. To see why this is correct, examine the mod 16 circle of Figure 2.3*b*. Compare it to larger circles for the mod 32 or mod 64 cases. The representations for values -1, -2, etc., would be exactly the same, with 1s added to the left. In summary, to represent a signed number in 2's-complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called *sign extension.*

The simplicity of either adding or subtracting signed numbers in 2's-complement representation is the reason why this number representation is used in modern computers. It might seem that the 1's-complement representation would be just as good as the 2's-complement system. However, although complementation is easy, the result obtained after an addition operation is not always correct. The carry-out, c_n, cannot be ignored. If $c_n = 0$, the result obtained is correct. If $c_n = 1$, then a 1 must be added to the result to make it correct. The need for this correction cycle, which is conditional on the carry-out from the add operation, means that addition and subtraction cannot be implemented as conveniently in the 1's-complement system as in the 2's-complement system.

2.1.4 OVERFLOW IN INTEGER ARITHMETIC

In the 2's-complement number representation system, n bits can represent values in the range -2^{n-1} to $+2^{n-1} - 1$. For example, using four bits, the range of numbers that can be represented is -8 through $+7$, as shown in Figure 2.1. When the result of an arithmetic operation is outside the representable range, an *arithmetic overflow* has occurred.

When adding unsigned numbers, the carry-out, c_n, from the most significant bit position serves as the overflow indicator. However, this does not work for adding signed numbers. For example, when using 4-bit signed numbers, if we try to add the numbers $+7$ and $+4$, the output sum vector, S, is 1011, which is the code for -5, an incorrect result. The carry-out signal from the MSB position is 0. Similarly, if we try to add -4 and -6, we get $S = 0110 = +6$, another incorrect result, and in this case, the carry-out signal is 1. Thus, overflow may occur if both summands have the same sign. Clearly, the addition of numbers with different signs cannot cause overflow. This leads to the following conclusions:

1. Overflow can occur only when adding two numbers that have the same sign.
2. The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.

A simple way to detect overflow is to examine the signs of the two summands X and Y and the sign of the result. When both operands X and Y have the same sign, an overflow occurs when the sign of S is not the same as the signs of X and Y.

2.1.5 CHARACTERS

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

2.2 MEMORY LOCATIONS AND ADDRESSES

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

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

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

2.2.1 BYTE ADDRESSABILITY

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

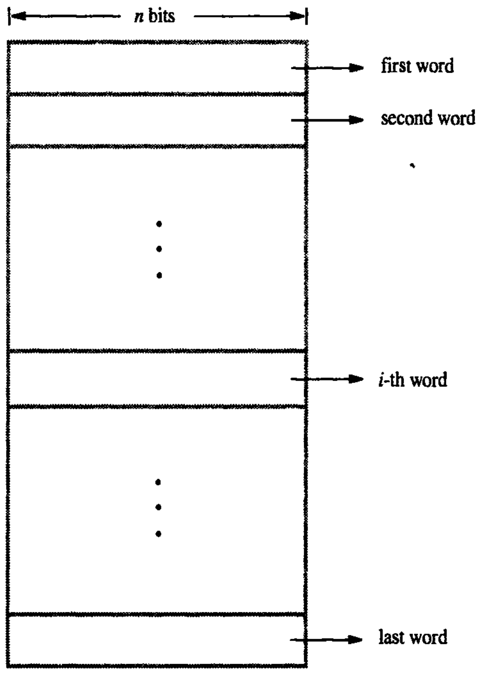
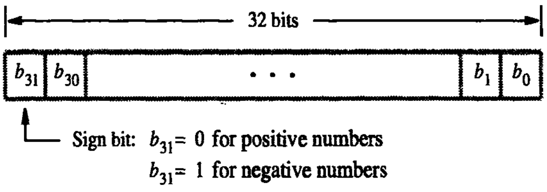
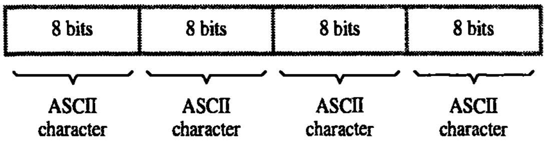


Figure 2.5 Memory words.



32 bits
b_{31}
b_{30}
b_1
b_0
Sign bit: $b_{31} = 0$ for positive numbers
$b_{31} = 1$ for negative numbers

(a) A signed integer



(b) Four characters

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

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

2.2.2 Big-Endian and Little-Endian Assignments

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

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

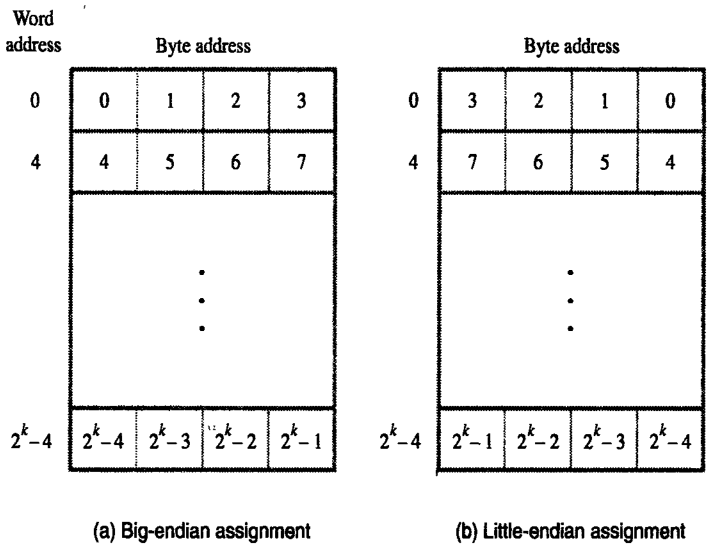


Figure 2.7 Byte and word addressing.

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

2.2.3 WORD ALIGNMENT

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

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

2.2.4 ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS

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

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

2.3 MEMORY OPERATIONS

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

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

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

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

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

2.4 INSTRUCTIONS AND INSTRUCTION SEQUENCING

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

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

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

2.4.1 REGISTER TRANSFER NOTATION

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

$$R1 \leftarrow [LOC]$$

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

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

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

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

2.4.2 Assembly Language Notation

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

```
Move   LOC,R1
```

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

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

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

2.4.3 Basic Instruction Types

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

$$C = A + B$$

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

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

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

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

Add A,B,C

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

Operation Source1,Source2,Destination

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

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

Operation Source,Destination

are available. An Add instruction of this type is

Add A,B

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

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

Move B,C

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

can now be performed by the two-instruction sequence

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

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

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

```
Add  A
```

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

```
Load  A
```

and

```
Store  A
```

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

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

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

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

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

Load A,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 $\leftarrow$ [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.

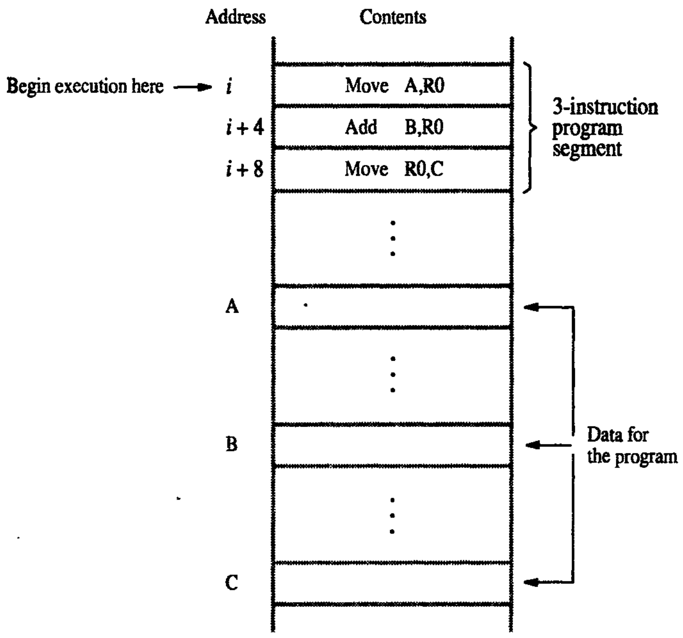


Figure 2.8 A program for C ← [A] + [B].

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, ..., NUM*n*, and a separate Add instruction is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in memory location SUM.

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

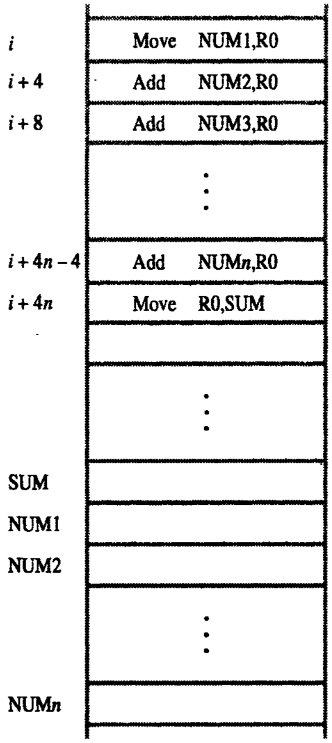


Figure 2.9 A straight-line program for adding *n* numbers.

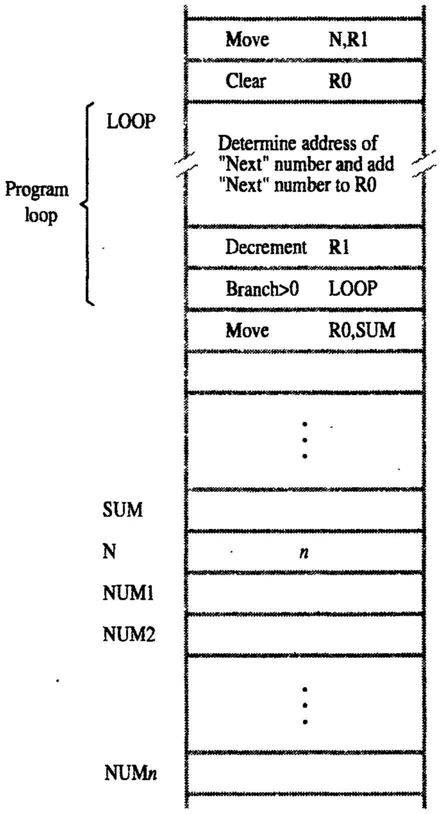


Figure 2.10 Using a loop to add *n* numbers.

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

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

```
Decrement   R1
```

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

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

In the program in Figure 2.10, the instruction

```
Branch>0   LOOP
```

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

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

2.4.6 CONDITION CODES

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

Four commonly used flags are

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

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

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

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

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

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

2.4.7 GENERATING MEMORY ADDRESSES

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

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

representing a fraction. In the 2's-complement system, the signed value F, represented by the n-bit binary fraction

$$B = b_0.b_{-1}b_{-2}\dots b_{-(n-1)}$$

is given by

$$F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-(n-1)} \times 2^{-(n-1)}$$

where the range of F is

$$-1 \leq F \leq 1 - 2^{-(n-1)}$$

Consider the range of values representable in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to $\pm 2.15 \times 10^9$. If we consider them to be fractions, the range is approximately $\pm 4.55 \times 10^{-10}$ to ± 1. Neither of these ranges is sufficient for scientific calculations, which might involve parameters like Avogadro's number (6.0247×10^{23} mole^{-1}) or Planck's constant (6.6254×10^{-27}erg$\cdot$s). Hence, we need to easily accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on them in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds. In such a case, the binary point is said to float, and the numbers are called *floating-point numbers*. This distinguishes them from fixed-point numbers, whose binary point is always in the same position.

Because the position of the binary point in a floating-point number is variable, it must be given explicitly in the floating-point representation. For example, in the familiar decimal scientific notation, numbers may be written as 6.0247×10^{23}, 6.6254×10^{-27}, -1.0341×10^2, -7.3000×10^{-14}, and so on. These numbers are said to be given to five *significant digits*. The *scale factors* (10^{23}, 10^{-27}, and so on) indicate the position of the decimal point with respect to the significant digits. By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be *normalized*. Note that the base, 10, in the scale factor is fixed and does not need to appear explicitly in the machine representation of a floating-point number. The sign, the significant digits, and the exponent in the scale factor constitute the representation. We are thus motivated to define a floating-point number representation as one in which a number is represented by its sign, a string of significant digits, commonly called the *mantissa*, and an exponent to an implied base for the scale factor.

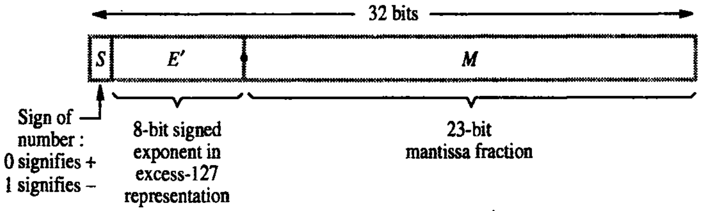
6.7.1 IEEE STANDARD FOR FLOATING-POINT NUMBERS

We start with a general form and size for floating-point numbers in the decimal system, and then relate this form to a comparable binary representation. A useful form is

$$\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y_1Y_2}$$

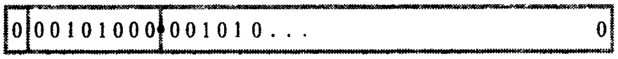
where X_i and Y_i are decimal digits. Both the number of significant digits (7) and the exponent range (± 99) are sufficient for a wide range of scientific calculations. It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits, which is a standard computer word length. A 24-bit mantissa can approximately represent a 7-digit decimal number, and an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. One bit is needed for the sign of the number. Since the leading nonzero bit of a normalized binary mantissa must be a 1, it does not have to be included explicitly in the representation. Therefore, a total of 32 bits is needed.

This standard for representing floating-point numbers in 32 bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE) [1]. The standard describes both the representation and the way in which the four basic arithmetic operations are to be performed. The 32-bit representation is given in Figure 6.24*a*. The sign of the number is given in the first bit, followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer $E' = E + 127$.


32 bits
S
E'
M
Sign of number : 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction

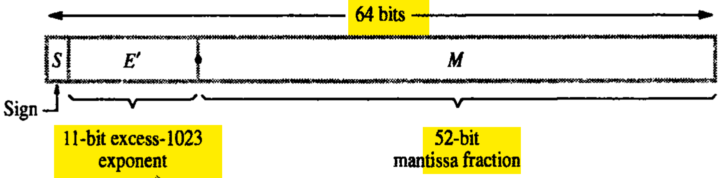
$$\text{Value represented} = \pm 1.M \times 2^{E' - 127}$$

(a) Single precision



$$\text{Value represented} = 1.001010\ldots0 \times 2^{-87}$$

(b) Example of a single-precision number



$$\text{Value represented} = \pm 1.M \times 2^{E' - 1023}$$

(c) Double precision

Figure 6.24 IEEE standard floating-point formats.

This is called the excess-127 format. Thus, E' is in the range $0 \leq E' \leq 255$. The end values of this range, 0 and 255, are used to represent special values, as described below. Therefore, the range of E' for normal values is $1 \leq E' \leq 254$. This means that the actual exponent, E, is in the range $-126 \leq E \leq 127$. The *excess-x* representation for exponents enables efficient comparison of the relative sizes of two floating-point numbers. (See Problem 6.27.)

The last 23 bits represent the mantissa. Since binary normalization is used, the most significant bit of the mantissa is always equal to 1. This bit is not explicitly represented; it is assumed to be to the immediate left of the binary point. Hence, the 23 bits stored in the M field actually represent the fractional part of the mantissa, that is, the bits to the right of the binary point. An example of a single-precision floating-point number is shown in Figure 6.24*b*.

The 32-bit standard representation in Figure 6.24*a* is called a *single-precision* representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value. To provide more precision and range for floating-point numbers, the IEEE standard also specifies a *double-precision* format, as shown in Figure 6.24*c*. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E' has the range $1 \leq E' \leq 2046$ for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

A computer must provide at least single-precision representation to conform to the IEEE standard. Double-precision representation is optional. The standard also specifies certain optional extended versions of both of these formats. The extended versions are intended to provide increased precision and increased exponent range for the representation of intermediate values in a sequence of calculations. For example, the dot product of two vectors of numbers can be computed by accumulating the sum of products in extended precision. The inputs are given in a standard precision, either single or double, and the answer is truncated to the same precision. The use of extended formats helps to reduce the size of the accumulated round-off error in a sequence of calculations. Extended formats also enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on. In addition to requiring the four basic arithmetic operations, the standard requires that the operations of remainder, square root, and conversion between binary and decimal representations be provided.

We note two basic aspects of operating with floating-point numbers. First, if a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. Figure 6.25 shows an unnormalized value, $0.0010110\ldots \times 2^9$, and its normalized version, $1.0110\ldots \times 2^6$. Since the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively. Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than $+127$. In the first case, we say that *underflow* has occurred, and in the second case, we say that *overflow* has occurred. Both underflow and overflow are arithmetic exceptions that are considered below.

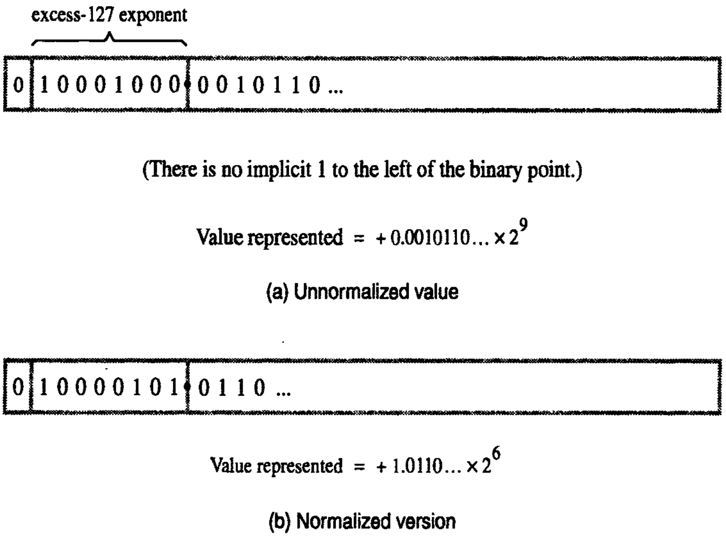


Figure 6.25 Floating-point normalization in IEEE single-precision format.

Special Values

The end values 0 and 255 of the excess-127 exponent E' are used to represent special values. When $E' = 0$ and the mantissa fraction M is zero, the value exact 0 is represented. When $E' = 255$ and $M = 0$, the value ∞ is represented, where ∞ is the result of dividing a normal number by zero. The sign bit is still part of these representations, so there are ± 0 and $\pm\infty$ representations.

When $E' = 0$ and $M \neq 0$, *denormal* numbers are represented. Their value is $\pm 0.M \times 2^{-126}$. Therefore, they are smaller than the smallest normal number. There is no implied one to the left of the binary point, and M is any nonzero 23-bit fraction. The purpose of introducing denormal numbers is to allow for *gradual underflow*, providing an extension of the range of normal representable numbers that is useful in dealing with very small numbers in certain situations. When $E' = 255$ and $M \neq 0$, the value represented is called *Not a Number* (NaN). A NaN is the result of performing an invalid operation such as $0/0$ or $\sqrt{-1}$.

Exceptions

In conforming to the IEEE Standard, a processor must set *exception* flags if any of the following occur in performing operations: underflow, overflow, divide by zero, inexact, invalid. We have already mentioned the first three. *Inexact* is the name for a result that requires rounding in order to be represented in one of the normal formats. An *invalid* exception occurs if operations such as $0/0$ or $\sqrt{-1}$ are attempted. When exceptions occur, the results are set to special values.

If interrupts are enabled for any of the exception flags, system or user-defined routines are entered when the associated exception occurs. Alternatively, the application program can test for the occurrence of exceptions, as necessary, and decide how to proceed.

A more detailed discussion of the floating-point issues raised here and in the next two sections is given in Appendix A of Hennessy and Patterson [2].

6.7.2 Arithmetic Operations on Floating-Point Numbers

In this section, we outline the general procedures for addition, subtraction, multiplication, and division of floating-point numbers. The rules we give apply to the single-precision IEEE standard format. These rules specify only the major steps needed to perform the four operations; for example, the possibility that overflow or underflow might occur is not discussed. Furthermore, intermediate results for both mantissas and exponents might require more than 24 and 8 bits, respectively. These and other aspects of the operations must be carefully considered in designing an arithmetic unit that meets the standard. Although we do not provide full details in specifying the rules, we consider some aspects of implementation, including rounding, in later sections.

If their exponents differ, the mantissas of floating-point numbers must be shifted with respect to each other before they are added or subtracted. Consider a decimal example in which we wish to add 2.9400×10^2 to 4.3100×10^4. We rewrite 2.9400×10^2 as 0.0294×10^4 and then perform addition of the mantissas to get 4.3394×10^4. The rule for addition and subtraction can be stated as follows:

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

Multiply Rule

1. Add the exponents and subtract 127.
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

Divide Rule

1. Subtract the exponents and add 127.
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

The addition or subtraction of 127 in the multiply and divide rules results from using the excess-127 notation for exponents.