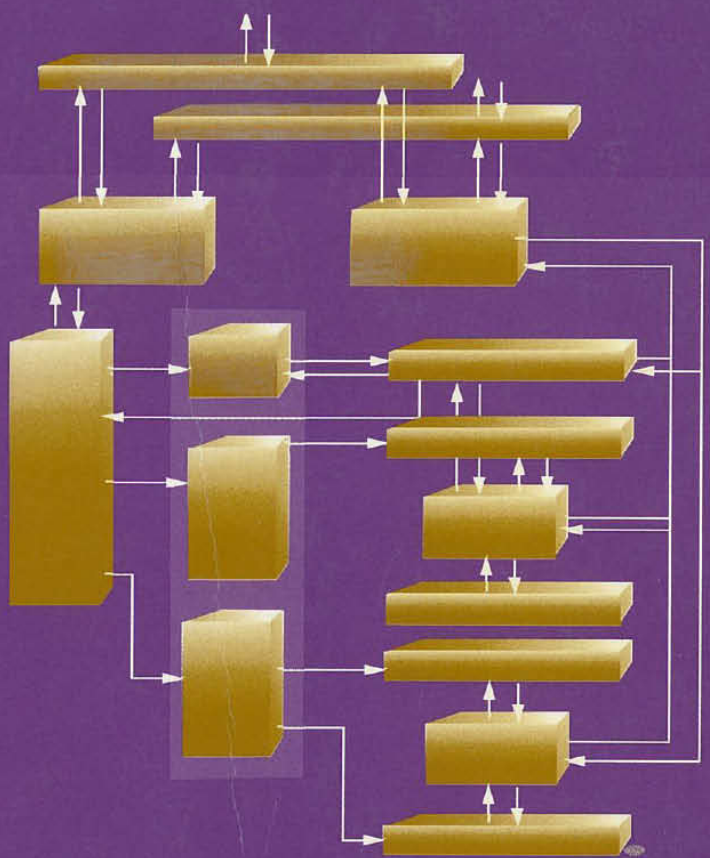
THIRD EDITION

Computer Architecture and Organization



John P. Hayes

McGRAW-HILL INTERNATIONAL EDITIONS
Computer Science Series

McGraw-Hill Series in Computer Organization and Architecture

Bell and Newell: *Computer Structures: Readings and Examples*

Cavanagh: *Digital Computer Arithmetic: Design and Implementation*

Feldman and Retter: *Computer Architecture and Logic Design*

Gear: *Computer Organization and Programming: With an Emphasis on Personal Computers*

Hamacher, Vranesic, and Zaky: *Computer Organization*

Hayes: *Computer Architecture and Organization*

Hayes: *Digital System Design and Microprocessors*

Horvath: *Introduction to Microprocessors Using the MC6809 or the MC68000*

Hwang: *Scalable Parallel and Cluster Computing: Architecture and Programming*

Hwang and Briggs: *Computer Architecture and Parallel Processing*

Lawrence and Mauch: *Real-Time Microcomputer System Design*

Siweiorek, Bell and Newell: *Computer Structures: Principles & Examples*

Stone: *Introduction to Computer Organization and Data Structures*

Stone and Siewiorek: *Introduction to Computer Organization and Data Structures: PDP-11 Edition*

Ward and Halstead: *Computational Structures*

McGraw-Hill Series in Computer Engineering

SENIOR CONSULTING EDITORS

Stephen W. Director, University of Michigan, Ann Arbor

C.L. Liu, University of Illinois, Urbana-Champaign

Bartee: *Computer Architecture and Logic Design*
Bose, Liang: *Neural Network Fundamentals with Graphs, Algorithms, and Applications*
Chang and Sze: *ULSI Technology*
De Micheli: *Synthesis and Optimization of Digital Circuits*
Feldman and Retter: *Computer Architecture: A Designer's Text Based on a Generic RISC*
Hamacher, Vranesic, and Zaky: *Computer Organization*
Hayes: *Computer Architecture and Organization*
Horvath: *Introduction to Microprocessors Using the MC6809 or the MC68000*
Hwang: *Advanced Computer Architecture: Parallelism, Scalability, Programmability*
Hwang: *Scalable Parallel and Cluster Computing: Architecture and Programming*
Kang and Leblebici: *CMOS Digital Integrated Circuits: Analysis and Design*
Kohavi: *Switching and Finite Automata Theory*
Krishna and Shin: *Real-Time Systems*
Lawrence-Mauch: *Real-Time Microcomputer System Design: An Introduction*
Levine: *Vision in Man and Machine*
Navabi: *VHDL: Analysis and Modeling of Digital Systems*
Peatman: *Design with Microcontrollers*
Peatman: *Digital Hardware Design*
Rosen: *Discrete Mathematics and Its Applications*
Ross: *Fuzzy Logic with Engineering Applications*
Sandige: *Modern Digital Design*
Sarrafzadeh and Wong: *An Introduction to VLSI Physical Design*
Schalkoff: *Artificial Neural Networks*
Stadler: *Analytical Robotics and Mechatronics*
Sze: *VLSI Technology*
Taub: *Digital Circuits and Microprocessors*
Wear, Pinkert, Wear, and Lane: *Computers: An Introduction to Hardware and Software Design*

Computer Architecture and Organization

THIRD EDITION

John P. Hayes
University of Michigan

WCB McGraw-Hill

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid
Mexico City Milan New Delhi Seoul Singapore Sydney Taipei Toronto

WCB/McGraw-Hill
A Division of The McGraw-Hill Companies

COMPUTER ARCHITECTURE AND ORGANIZATION
International Editions 1998

7 8 9 10 MPM FC 20 9 8 7 6 5 4 3 2

Library of Congress Cataloging-in-Publication Data

Hayes, John P. (John Patrick) (date)
Computer architecture and organization / John P. Hayes. – 3rd ed.
p. cm. – (Electrical and computer engineering)
Includes bibliographical references and index.
ISBN 0-07-027355-3
1. Construction architecture. 2. Electronic digital computers–Design and construction. I. Title. II. Series: McGraw-Hill series in electrical and computer engineering.
QA76.9.A73H39 1998
621.39'2–dc21 97-45598

www.mhhe.com

When ordering this title, use ISBN 0-07-115997-5

Printed in Singapore

ABOUT THE AUTHOR

JOHN P. HAYES is a professor in the electrical engineering and computer science department at the University of Michigan, where he was the founding director of the Advanced Computer Architecture Laboratory. He teaches and conducts research in the areas of computer architecture; computer-aided design, verification, and testing; VLSI design; and fault-tolerant systems. Dr. Hayes is the author of two patents, more than 150 technical papers, and five books, including *Layout Minimization for CMOS Cells* (Kluwer, 1992, coauthored with R. L. Maziasz) and *Introduction to Digital Logic Design* (Addison-Wesley, 1993). He has served as editor of various journals, including the *IEEE Transactions on Parallel and Distributed Systems* and the *Journal of Electronic Testing,* and was technical program chairman of the 1991 International Computer Architecture Symposium, Toronto.

Dr. Hayes received his undergraduate degree from the National University of Ireland, Dublin, and his M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana-Champaign. Prior to joining the University of Michigan, he was a faculty member at the University of Southern California. Dr. Hayes has also held visiting positions at various academic and industrial organizations, including Stanford University, McGill University, Université de Montréal, and LogicVision Inc. He is a fellow of the Institute of Electrical and Electronics Engineers and a member of the Association for Computing Machinery and Sigma Xi.

To My Father
Patrick J. Hayes
(1910–1968)
In Memoriam

CONTENTS

CHAPTER 1

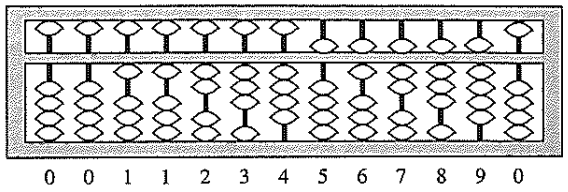
Computing and Computers

This chapter provides a broad overview of digital computers while introducing many of the concepts that are covered in depth later. It first examines the nature and limitations of the computing process. Then it briefly traces the historical development of computing machines and ends with a discussion of contemporary VLSI-based computer systems.

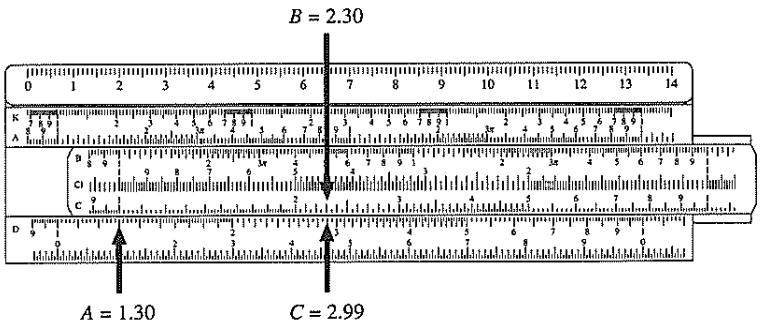
1.1 THE NATURE OF COMPUTING

Throughout history humans have relied mainly on their brains to perform calculations; in other words, they were the computers [Boyer 1989]. As civilization advanced, a variety of computing tools were invented that aided, but did not replace, manual computation. The earliest peoples used their fingers, pebbles, or tally sticks for counting purposes. The Latin words *digitus* meaning "finger" and *calculus* meaning "pebble" have given us *digital* and *calculate* and indicate the ancient origins of these computing concepts.

Two early computational aids that were widely used until quite recently are the abacus and the slide rule, both of which are illustrated in Figure 1.1. The abacus has columns of pebblelike beads mounted on rods. The beads are moved by hand to positions that represent numbers. Manipulating the beads according to certain simple rules enables people to count, add, and perform the other basic operations of arithmetic. The slide rule, on the other hand, represents numbers by lengths marked on rulerlike scales that can be moved relative to one another. By adding a length a on a fixed scale to a length b on a second, sliding scale, their combined length $c = a + b$ can be read off the fixed scale. The slide rule's main scales are logarithmic, so that the process of adding two lengths on these scales effectively multiplies two



(*a*)



(*b*)

Figure 1.1
(*a*) Japanese abacus (*soroban*) displaying the number 0011234567890;
(*b*) slide rule illustrating the multiplication $1.30 \times 2.30 = 2.99$.

numbers.[1] Slide rules are marked with various other scales that allow an experienced user to evaluate complicated expressions such as $2.15 \times 17.9^{-50} \sin \pi$ in several steps.

As the size and complexity of the calculations being carried out increases, two serious limitations of manual computation become apparent.

- The speed at which a human computer can work is limited. A typical elementary operation such as addition or multiplication takes several seconds or minutes. Problems requiring billions of such operations could never be solved manually in a reasonable period of time or at reasonable cost. Fortunately, modern computers routinely tackle and quickly solve such problems.
- Humans are notoriously prone to error, so long calculations done by hand are unreliable unless elaborate precautions are taken to eliminate mistakes. Most sources of human error (distraction, fatigue, and the like) do not affect machines, so they can provide results that are, within broad limits, free from error.

The English computer pioneer Charles Babbage (1792–1871) often cited the following example to justify construction of his first automatic computing

[1]Logarithms are defined by the relation $10^a = A$, where $a = \log_{10} A$. A length marked A on a log scale is proportional to $\log_{10} A = a$. When we add two lengths marked A and B on a slide rule, we are actually adding $a = \log_{10} A$ and $b = \log_{10} B$. Therefore, the result c represents $\log_{10} A + \log_{10} B$. Now $10^a \times 10^b = 10^{a+b}$ implies $c = \log_{10} A + \log_{10} B = \log_{10} (A \times B)$, so if we read c from the first scale, we will obtain the number whose log is c, that is, $A \times B$.

machine, the Difference Engine [Morrison and Morrison 1961]. In 1794 the French government began a project to compute entirely by hand an enormous set of mathematical tables. Among the many required tables were the logs of the numbers from 1 to 200,000 calculated to 19 decimal places. The entire project took two years to complete and employed about 100 people. The mathematical abilities of most of these human computers were limited to addition and subtraction, and they performed their calculations using pen and paper. A few skilled mathematicians provided the instructions. To minimize errors, each number was calculated independently by two human calculators. The final set of tables occupied 17 large volumes. The log table alone contained about 8 million digits.

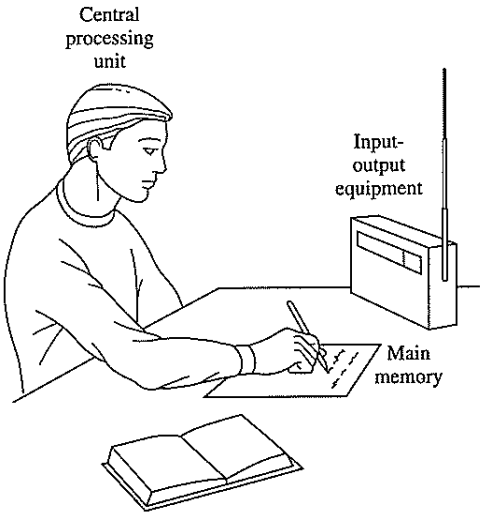
1.1.1 The Elements of Computers

Every computer, human or artificial, contains the following components: a processor able to interpret and execute programs; a memory for storing the programs and the data they process; and input-output equipment for transferring information between the computer and the outside world.

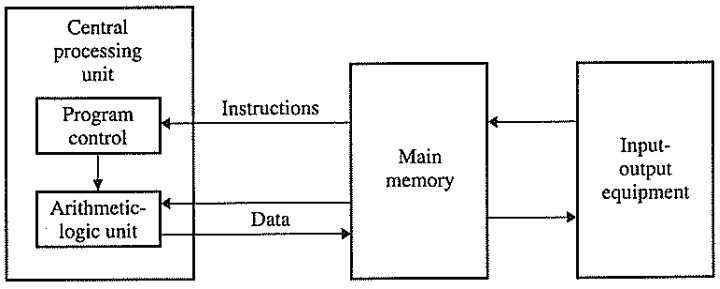
The brain versus the computer. Consider the actions involved in a manual calculation using pencil and paper—for example, filling out an income tax return. The purpose of the paper is *information storage*. The information stored can include a list of instructions—more formally called a *program, algorithm,* or *procedure*—to be followed in carrying out the calculation, as well as the numbers or *data* to be used. During the calculation intermediate results and ultimately the final results are recorded on the paper. The data processing takes place in the human brain, which serves as the (*central*) *processor*. The brain performs two distinct functions: a *control* function that interprets the instructions and ensures that they are performed in the proper sequence and an *execution* function that performs specific steps such as addition, subtraction, multiplication, and division. A pocket calculator often serves as an aid to the brain. Figure 1.2*a* illustrates this view of human computation.

A computer has several key components that roughly correspond to those just mentioned; see Figure 1.2*b*. The *main memory* corresponds to the paper used in the manual calculation. Its purpose is to store instructions and data. The computer's brain is its *central processing unit* (CPU). It contains a *program control unit* (also known as an instruction unit) whose function is to fetch instructions from memory and interpret them. An *arithmetic-logic unit* (ALU), which is part of the CPU's data-processing or execution unit, carries out the instructions. The ALU is so called because many instructions specify either arithmetic (numerical) operations or various forms of nonnumerical operations that loosely correspond to logical reasoning or decision making.

There are important similarities and differences between human beings and artificial computers in the way in which they represent information. In both cases information is usually in *digital* or discrete form. This is contrasted with *analog* or continuous information as used, for example, in the slide rule of Figure 1.1*b*. Distance is a continuous quantity, and on a slide-rule scale it represents, or serves as an analog for, a continuous sequence of numbers. The problem is that such analog quantities have very limited accuracy. The numbers on a slide rule, for example,



(*a*)



(*b*)

Figure 1.2
Main components of (*a*) human computation and (*b*) machine computation.

cannot be read to more than three decimal places. On the other hand, a digital device can easily handle a large number of digits. Even the simple abacus of Figure 1.1*a* can display a number—admittedly just one—to 13 places of accuracy. This advantage of digital data representation over analog is also seen in the higher fidelity of the sound recorded on a compact disc (CD), a digital device, compared to an old-fashioned record (LP), which is an analog device.

Humans employ languages with a wide range of digital symbols, and they usually represent numbers in decimal (base 10) form. It is not practical to build computers to handle symbolic or decimal data directly. Instead, computers process data in binary form, that is, using the two symbols 0 and 1 called *bits* (binary digits). Computers are built from electronic switches that have two natural states: off (0) and on (1). Hence the internal "language" of computers comprises forbidding-looking strings of bits such as 10010011 11011001. To provide communication

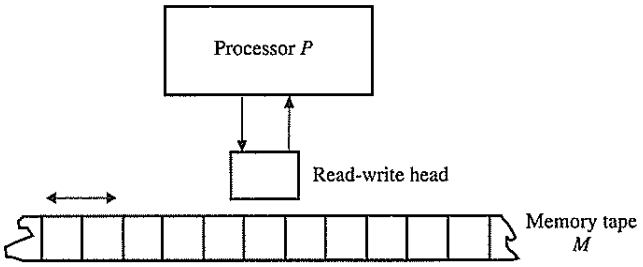


Figure 1.3
A Turing machine.

between a computer and its human users, a means of translating information between human and machine (binary) formats is necessary. The *input-output* equipment shown in Figure 1.2b performs this task.

An abstract computer. We are interested in the computational abilities of general-purpose digital computers. One might raise the following question at the outset: Are there any computations that a "reasonable" computer can never perform? Three notions of reasonableness are widely accepted.

- The computer should not store the answers to all possible problems.
- The computer should only be required to solve problems for which a solution procedure or program can be given.
- The computer should process information at a finite speed.

A reasonable computer can therefore solve a particular problem only if it is supplied with a program that can generate the answer in a finite amount of time.

In the 1930s the English mathematician Alan M. Turing (1912–54) introduced an abstract model of a computer that satisfies all the foregoing criteria [Barwise and Etchemendy 1993]. This model, now called a *Turing machine*, has the structure shown in Figure 1.3. As we noted earlier two essential elements of any computer are a memory and a processor. The memory of a Turing machine is a tape M which resembles that of a tape recorder. Unlike the tape recorder, however, the Turing machine's tape is of unbounded length and is divided lengthwise into squares. Each square can be blank, or it can contain one of a small set of symbols. The Turing machine's processor P is a simple device with a small number of internal configurations or *states*. It is linked to M by a *read-write head* that can read the contents of one square Q and write a new symbol into Q to replace the old one in a single time step. Instead of writing on the tape, the processor can also just read the current symbol and move the tape one square to the left or right of the current square Q.

We can view the Turing machine as having a set of instructions that we will write in the compact, four-part format

$$S_h \quad T_i \quad O_j \quad S_k$$

This instruction is interpreted in the following way: If the present state of the processor P is S_h and the symbol it reads on the square of M under the read-write head is T_i, then perform the action (such as write a new symbol or move the tape)

specified by O_j and change the state of P to S_k. Another way of expressing this instruction, which is more in tune with the style of a modern computer programming language, is

if *oldstate* = S_h **and** *input* = T_i **then** *output* = O_j **and** *newstate* = S_k;

The output operation indicated by O_j can be any one of the following:

1. $O_j = T_j$, meaning write the symbol T_j on the tape to replace the symbol T_i.
2. O_j = R, meaning move the tape so that the read-write head is over the square to the right of the current square. (The tape is moved one square to the left.)
3. O_j = L, meaning move the tape so that the read-write head is over the square to the left of the current square. (The tape is moved one square to the right.)
4. O_j = H, meaning halt the computation.

The foregoing apparently restricted form of instruction, with just a few different symbols to write on M and a few different states for P, turns out to be sufficient to define programs that can perform all reasonable computations. To determine the value of $Z = F(X)$ via a Turing machine, where F is some function of interest, we proceed as follows: The input data X is placed in a suitably coded form on an otherwise blank tape M. The processor P is supplied with a program that specifies a sequence of steps that are designed to compute F. The Turing machine is then started and executes instruction after instruction, moving the tape M and writing intermediate results on it. Eventually, the Turing machine should halt, and the final result Z should be found on the tape.

EXAMPLE 1.1 A TURING MACHINE TO ADD TWO UNARY NUMBERS. Any natural number n, that is, a positive integer selected from the set we usually write as 0, 1, 2, 3, 4, 5,..., can be written in the *unary* form consisting of a sequence of n 1s. For example, 5 can be written as 11111 and 13 as 1111111111111. When we record numbers using tally or check marks only, we are using a unary notation. (Surprisingly, unary numbers still have a small place in computer design [Poppelbaum et al. 1985].)

We will now show how to program a Turing machine to compute the sum of two unary numbers n_1 and n_2. The tape symbols needed are 1 and b, where b denotes a blank. We start with a blank tape (one containing b in every square) and write the two input numbers in the following format:

$$\ldots\ \mathrm{b}\ \mathrm{b}\ \underline{\mathrm{b}}\ \underbrace{1\ 1\ 1\ \ldots\ 1}_{n_1}\ \mathrm{b}\ \underbrace{1\ 1\ 1\ \ldots\ 1}_{n_2}\ \mathrm{b}\ \mathrm{b}\ \mathrm{b}\ \ldots$$

We position the read-write head over the blank square (underlined above) to the left of the left-most 1. Our Turing machine then computes $n_1 + n_2$ by the simple expedient of finding the single blank that separates n_1 and n_2 and replacing it with 1. The machine then finds and deletes the left-most 1 of n_1. The resulting pattern of 1s and bs

$$\ldots\ \mathrm{b}\ \mathrm{b}\ \underline{\mathrm{b}}\ \mathrm{b}\ \underbrace{1\ 1\ \ldots\ 1\ 1\ 1\ 1\ 1\ \ldots\ 1}_{n_1+n_2}\ \mathrm{b}\ \mathrm{b}\ \mathrm{b}\ \ldots$$

appearing on the tape is the required answer in the same unary format as the input data. The behavior of a seven-instruction Turing machine that implements this procedure is given with explanatory comments in Figure 1.4. Observe that although the tape M can have an arbitrarily large number of states, the processor P has only the four states S_0, S_1, S_2, and S_3.

Instruction				Comment
S_0	b	R	S_1	Move read-write head one square to right.
S_1	1	R	S_1	Move read-write head rightward across n_1.
S_1	b	1	S_2	Replace blank between n_1 and n_2 by 1.
S_2	1	L	S_2	Move read-write head leftward across n_1.
S_2	b	R	S_3	Blank square reached; move one square to right.
S_3	1	b	S_3	Replace left-most 1 by blank.
S_3	b	H	S_3	Halt; the result $n_1 + n_2$ is now on the tape.

Figure 1.4
Turing machine program to add two unary numbers.

One of Turing's most remarkable achievements was to prove that a *universal* Turing machine (not unlike the above unary adding machine) can by itself perform *every* reasonable computation. A universal Turing machine is essentially a simulator of Turing machines. If given a description of some particular Turing machine *TM*—a program description like that of Figure 1.4 will do—the universal machine simulates all the operations performed by *TM*. A universal Turing machine needs only t different tape symbols and s different processor states, where $ts < 30$, implying that it can have a very small instruction set. Nevertheless, such a machine can perform any reasonable computation. It can therefore do anything that any real computer can do and so serves as an abstract model of the modern general-purpose computer. The universal Turing machine also captures a little of the flavor of reduced instruction set computers (RISCs), which, despite having relatively few instruction types, are among the most powerful computing machines available today.

1.1.2 Limitations of Computers

We turn next to the question of what problems computers can and cannot solve, either in principle or in practice [Barwise and Etchemendy 1993; Cormen and Leiserson 1990; Garey and Johnson 1979].

Unsolvable problems. Problems exist that no Turing machine and therefore no practical computer can solve. There are well-defined problems, some quite famous, for which no solutions or solution procedures are known. An example from pure mathematics is *Goldbach's conjecture,* formulated by the mathematician Christian Goldbach (1690–1764), which states that every even integer greater than 2 is the sum of exactly two prime numbers. For instance, $8 = 3 + 5$ and $108 = 37 + 71$. Goldbach's conjecture has been tested for an enormous number of even integers and is true in all test cases. Nevertheless, it is not yet known if the conjecture is true for *every* even integer, nor is any reasonable procedure known to determine whether the conjecture is true. The number of even integers is infinite, so a complete or exhaustive examination of all even integers and their prime factors is not feasible.

Goldbach's conjecture is an example of an unsolved problem that may eventually be solved—we just don't have a suitable solution procedure yet. Turing

machines have proven another class of problems to be unsolvable, so there is no hope of ever solving them; such problems are said to be *undecidable*. An example of an undecidable problem is to determine if an arbitrary polynomial equation of the form

$$a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n = b$$

has a solution consisting entirely of integers. This problem may be answerable for specific equations, but a general procedure or program can never be constructed that can analyze *any* possible polynomial equation and decide if it has an integer solution.

Turing identified an undecidable problem that involves the basic nature of Turing machines. Does a procedure exist to determine if an arbitrary Turing machine with arbitrary input data will ever halt once it has been set in motion? Turing proved that the answer is no, so the *Turing machine halting problem* as this particular problem is called, is also undecidable. This result has some practical implications. A common and costly error made by inexperienced computer programmers is to write programs that contain infinite loops and therefore fail to halt under certain input conditions. It would be useful to have a debugging program that could determine whether any given program contains an infinite loop. The undecidability of the Turing machine halting problem implies that no such infinite-loop-detecting tool can ever be realized.

The Turing machine model of a computer has one unrealistic, if not unreasonable, aspect: The length of the tape memory, and hence the total number of states in the Turing machine, is infinite. Real computers have a finite amount of memory and are therefore referred to as *finite-state machines*. Therefore, Turing machines can perform some computations that, in principle, finite-state machines cannot perform. For example, a finite-state machine cannot multiply two arbitrarily large numbers because it eventually runs out of the states needed to compute the product. The number of states of a typical computer is enormous, so this finiteness limitation has little significance. A typical general-purpose computer has billions of states and can quickly multiply numbers of any practical length.

Intractable problems. Real (finite-state) computers can solve most computational problems to an acceptable degree of accuracy. The question then becomes: Can a computer of reasonable size and cost solve a given problem in a reasonable amount of time? If so, the problem is said to be *tractable*; otherwise, it is *intractable*. Whether a given problem is tractable depends on several factors: the nature of the problem itself, the solution method or program used, and the computing speed or *performance* of the computer available to solve it. Figure 1.5 gives an indication of the speed of modern computers. It shows how the number of basic operations, such as the addition of two numbers, that a CPU can perform has been evolving with advances in computer hardware.

Example 1.2 illustrates the impact of the solution method on problem difficulty.

EXAMPLE 1.2 FINDING AN EULER CIRCUIT IN A GRAPH. A well-known problem associated with the Swiss mathematician Leonhard Euler (1707–1783) is the following: Given a set of connected paths such as the aisles in an exhibition hall (Figure 1.6*a*), is it possible to make a tour of the hall so that one walks along every aisle exactly once and ends up at the starting point? The problem can be represented abstractly by means of a *graph*, as shown in Figure 1.6*b*. Each aisle is modeled by a

Component technology	Date	Number of basic operations per second
Electromechanical: relays	1940	10
Electronic: vacuum tubes (valves)	1945	10^3
Electronic: transistors	1950	10^4
Small-scale integrated circuits	1960	10^5
Medium-scale integrated circuits	1980	10^6
Very large-scale integrated circuits	2000	10^9

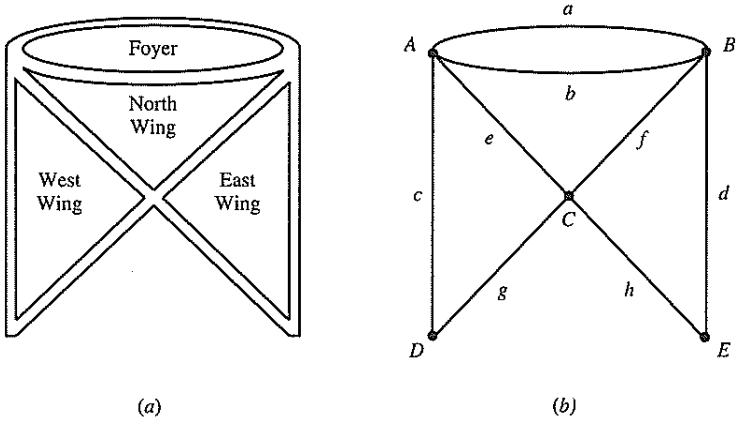
Figure 1.5
Influence of hardware technology on computing speed.

line called an *edge*, and the junction of two or more aisles by a point called a *node*. The graph of Figure 1.6*b* has five nodes *A, B, C, D,* and *E* and eight edges *a, b, c, d, e, f, g,* and *h*. Restated in graph terms, the walking-tour problem becomes that of finding a closed path around the graph that contains every edge exactly once; such a path is known as an *Euler circuit*. We consider two possible ways to determine whether a graph contains an Euler circuit.

A "brute force" or exhaustive approach is to generate a list of the possible orderings or *permutations* of the edges of the graph. Each permutation then corresponds to a potential tour of the exhibition hall. The list of permutations can be written in the form

abcdefgh, acbdefgh, adbcefgh, aebcdfgh, afbcdegh, ... (1.1)

We can search the permutation list and check each entry to see if it specifies an Euler circuit. Clearly, the list is huge, and most of its entries do not represent Euler circuits. For example, the first permutation *abcdefgh* does not represent an Euler circuit, because while it is possible to go from *a* to *b* and from *b* to *c*, it is not possible to go directly from *c* to *d*. A tour starting at node *A* that traverses *a, b,* and *c* must continue along *g*, at which point *f* or *h* may be followed. The permutation *abcgfdhe* appearing



(*a*) (*b*)

Figure 1.6
(*a*) Plan of the aisles in an exhibition hall and (*b*) the corresponding graph model.

somewhere down the list represents a circuit of the desired kind, as can be quickly verified. Thus we conclude that the graph of Figure 1.6*b* does indeed contain an Euler circuit.

The main drawback of this brute-force method is the length the permutation list; the time needed to generate, store, and check it is enormous. Most of the list's entries do not represent Euler circuits, but in the worst case, we might have to search the entire list to find an Euler circuit or prove that none exists. The number of possible permutations of the eight edges in our example is 8!, which denotes eight factorial. Therefore

$$8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40{,}320$$

is the length of list (1.1). When q, the number of edges present, is large, the size of the permutation list $q!$ is approximated by

$$\sqrt{2\pi q}\left(\frac{q}{e}\right)^q$$

which shows that the size of the brute-force procedure in terms of storage requirements and computing speed increases exponentially with q. If q were 80 instead of 8, then we would have $q! = 80! \approx 7.16 \times 10^{118}$. This huge number exceeds the estimated number (10^{10}) of neurons in the human brain. A very fast computer capable of processing a trillion (10^{12}) permutations per second would spend 2.27×10^{99} years dealing with 80! permutations. We can therefore conclude with some confidence that the problem of finding an Euler circuit is intractable via the brute-force approach.

An alternative but very tractable solution procedure for the same problem depends on Euler's discovery that a graph has the desired circuit if and only if every node is the junction of an even number of edges. Intuitively, this result follows from the fact that every edge used to enter a node must be paired with an edge used to exit the node. Now the task of determining whether a graph contains an Euler cycle reduces to checking each node in turn and counting the edges that it connects. In the example of Figure 1.6*b*, nodes *A, B, C, D,* and *E* form the junctions of 4, 4, 4, 2, and 2 edges, respectively. It follows immediately that the graph has an Euler circuit. While the brute-force method requires a computation time and a storage capacity that grow exponentially with the number of edges q, the second method has a computational complexity that is proportional to q. The second method can easily solve problems with 80 or more edges.

Because the problem of finding an Euler circuit has an efficient and practical solution procedure, as shown in Example 1.2, we regard the problem itself as inherently tractable. We usually regard a problem as intractable if all its known solution methods grow exponentially with the size of the problem. Many problems, some of great practical importance, are inherently intractable in this way. Only small versions of such intractable problems can be solved in practice, where smallness is measured by some problem-dependent parameter such as the number of input variables present.

An example of an intractable problem related to Example 1.2 is the *traveling salesman problem.* Here the goal is also to make a tour, this time by car or plane through a given set of n cities, and eventually return to the starting point. The distance between each pair of cities is known, and the problem is to determine a tour that minimizes the total distance traveled. Again it is convenient to use a graph model with nodes denoting cities and edges denoting intercity highways with distances marked on them—the graph is tantamount to a roadmap. The best solution procedures known for this problem, although better than the brute-force approach

of listing all possible tours through the n cities, are exponential in n. Quite a few practical problems are closely related to the traveling salesman problem: The scheduling of airline flights, the routing of wires in an electronic circuit, and the sequencing of steps in a factory assembly line are examples. Such difficult computing problems are a major motivation for the design and construction of bigger and faster computers.

An intractable problem can be solved exactly in a reasonable amount of time only when its size n is below some maximum value n_{MAX}. The value of n_{MAX} depends both on the problem itself and on the speed of the computers available to solve it. It might be expected that computer speeds could be increased to make n_{MAX} any desired value. We now present arguments to indicate that this is highly unlikely.

Speed limitations. An algorithm A has *time complexity* of order $f(n)$, denoted $O(f(n))$, if the number of basic operations—the precise nature of these operations is not important—A uses to solve a problem of size n is at most $cf(n)$, where $f(n)$ is some function of n and c is a constant. The function $f(n)$ therefore indicates the rate at which the computing time that A needs to obtain a solution grows with the problem size n.

To gauge the impact of computing speed on the size n_{MAX} of the largest solvable problem, we consider four algorithms A_1, A_2, A_3, and A_4 of varying degrees of difficulty. Let the time complexities of A_1, A_2, A_3, and A_4 be $O(n)$, $O(n^2)$, $O(n^{100})$, and $O(2^n)$, respectively. Because A_4 has a time complexity that is exponential in n, it is the only obviously intractable procedure. Suppose that all four algorithms are programmed on a computer M having a speed of S basic operations per second. Let n_i denote the size of the largest problem that algorithm A_i can solve in a fixed time period of T seconds. Let n_i' denote the size of the largest problem that the same algorithm A_i can solve in T seconds on a new computer M' that is 100 times faster than M; the speed of M' is therefore $100S$ operations per second. M' could be implemented by a different and faster hardware technology than M. It could also—at least in principle—be implemented by a "supercomputer" consisting of 100 copies of M all working in parallel on the same problem, a technique referred to as *parallel processing*.

Figure 1.7 shows the values of n_i' relative to n_i for the four algorithms. In the case of the intractable algorithm A_4, the increase in the size of the largest problem that can be handled on moving from M to M' is insignificant. This is also true for A_3, even though it does not fall within the strict definition of intractability. To increase the size of the maximum problem that A_3 and A_4 can solve in the given

Algorithm	Time complexity	Maximum problem size	
		Computer M	Computer M'
A_1	$O(n)$	n_1	$n_1' = 100n_1$
A_2	$O(n^2)$	n_2	$n_2' = 10n_2$
A_3	$O(n^{100})$	n_3	$n_3' = 1.047n_3$
A_4	$O(2^n)$	n_4	$n_4' = n_4 + 6.644$

Figure 1.7
Effect of computer speedup by 100 on four algorithms.

time period by a factor of 100, we would need computers with speeds of $10^{200}S$ and $10^{30n^4}S$, respectively. It is reasonable to expect that problems of these magnitudes can *never* be solved by the given algorithms on realistic computers.

Because so many important problems are intractable, we often devise approximate or inexact methods to solve them. Two major techniques follow.

1. We replace the intractable problem Q with a tractable problem Q' whose solution approximates that of Q.
2. We examine a relatively small set of possible solutions to Q using reasonable, intuitive, and often poorly understood selection criteria and take the "best" of these as the solution to Q. Methods that are designed to produce acceptable, if not optimal, answers using a reasonable amount of computing time are sometimes called *heuristic* procedures.

To illustrate the heuristic approach, consider again the traveling salesman problem. The salesman must visit n cities and return to his starting point. All intercity distances are specified, and the objective of the problem is to find a tour that minimizes the total distance traveled by the salesman. We can represent the problem on a graph similar to that of Figure 1.6*b*, whose nodes denote cities and whose edges denote intercity links. A brute-force approach of the kind discussed in Example 1.2, which involves listing all $n!$ possible tours and their distances, is intractable, and no obviously tractable method to obtain a minimum-distance tour is known.

Real traveling salesmen often use the following simple heuristic: Go to the previously unvisited city that is closest to the current city and return to the start in the final leg of the tour. Hence for each of the n legs, the only computation needed is to compare the distances between the current city and each of at most $n - 1$ other cities. The city that is the shortest distance away (if there are several such cities, select any one of them) is visited next. Because this heuristic makes decisions that are optimal on a local basis only, it will not always find an overall optimum. Nevertheless, for most practical problems this heuristic provides a solution of minimum or near-minimum length, but there is no guarantee that it will do so in any particular case.

Computers are continually being applied to new problems whose computational requirements far exceed those of older problems. For example, the processing of high-quality speech and visual images for multimedia applications can require speeds measured in trillions of basic operations per second. To meet the ever-increasing demand for high-performance computation, we need better algorithms and heuristics, as well as faster computers. Although computers continue to increase in speed because of advances in hardware technology, the rate of increase (see Figure 1.5) has not kept pace with demand. As a result, we still need to find new ways to improve the performance of computers at reasonable cost—which is the basic rationale for the study of computer architecture and organization.

1.2 THE EVOLUTION OF COMPUTERS

Calculating machines capable of performing the elementary operations of arithmetic (addition, subtraction, multiplication, and division) appeared in the 16th century, and perhaps earlier [Randell 1982; Augarten 1984]. These were clever

mechanical devices constructed from gears, levers, and the like. The French philosopher Blaise Pascal (1623–62) invented an early and influential mechanical calculator that could add and subtract decimal numbers. Decimal numerals were engraved on counter wheels much like those in a car's odometer. Pascal's main technical innovation was a ratchet device for automatically transferring a carry from a digit d_i to the digit d_{i+1} on its left whenever d_i passed from 9 to 0. In Germany, Gottfried Leibniz (1646–1716) extended Pascal's design to one that could also perform multiplication and division. Mechanical computing devices such as these remained academic curiosities until the 19th century, when the commercial production of mechanical four-function calculators began.

1.2.1 The Mechanical Era

Various attempts were made to build general-purpose programmable computers from the same mechanical devices used in calculators. This technology posed some daunting problems, and they were not satisfactorily solved until the introduction of electronic computing techniques in the mid-20th century.

Babbage's Difference Engine. In the 19th century Charles Babbage designed the first computers to perform multistep operations automatically, that is, without a human intervening in every step [Morrison and Morrison 1961]. Again the technologies were entirely mechanical. Babbage's first computing machine, which he called the Difference Engine, was intended to compute and print mathematical tables automatically, thereby avoiding the many errors occurring in tables that are computed and typeset by hand. The Difference Engine performed only one arithmetic operation: addition. However, the *method of (finite) differences* embodied in the Difference Engine can calculate many complex and useful functions by means of addition alone.

EXAMPLE 1.3 COMPUTING x^2 BY THE METHOD OF DIFFERENCES. Consider the task of calculating a table of the squares $y_j = x_j^2$, for $x_j = 1, 2, 3, \ldots$ using the method of differences. To understand the underlying concept, suppose we already have the list of squares given in Figure 1.8*a*. Subtract each square $y_j = x_j^2$ from the next value $y_{j+1} = (x_j + 1)^2$ in the list. The result $(x_j + 1)^2 - x_j^2 = 2x_j + 1$ is called the first difference of y and is denoted by $\Delta^1 y_j$; the corresponding list of values in Figure 1.8*a* is 3, 5, 7, ... If we subtract two consecutive first-difference values, we obtain $2(x_j + 1) + 1 - (2x_j + 1) = 2$, which is the second difference $\Delta^2 y_j$ of y. Note that the second difference is constant for all j.

The Difference Engine evaluates x^2 by taking the constant second difference $\Delta^2 y_j$ and adding it to the first difference $\Delta^1 y_j$. The result is

$$\Delta^1 y_{j+1} = \Delta^1 y_j + \Delta^2 y_j \tag{1.2}$$

which is the next value of the first difference. At the same time, the engine calculates

$$y_{j+1} = y_j + \Delta^1 y_j \tag{1.3}$$

which is the next value of x^2. By repeatedly executing the two addition steps (1.2) and (1.3), the Difference Engine can generate any desired sequence of consecutive squares. It must be "primed" by manually inserting the initial values $y_1 = 1$, $\Delta^1 y_1 = 3$,

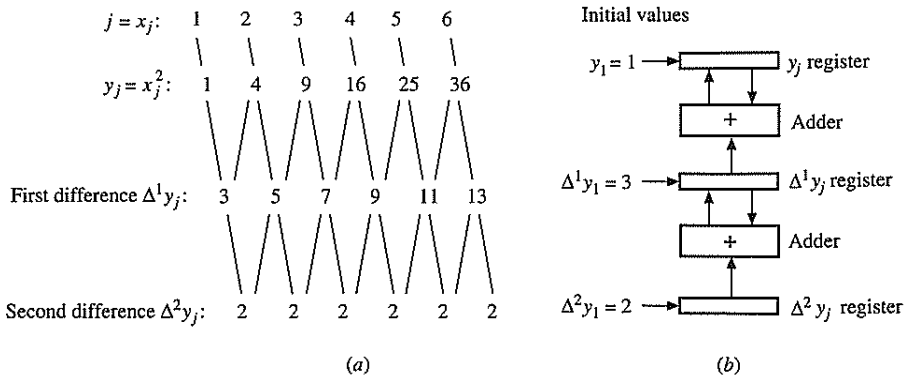


Figure 1.8
Computing x^2 by the method of differences: (*a*) a representative computation and (*b*) the corresponding Difference Engine configuration.

and $\Delta^2 y_1 = 2$ for $j = 1$, which appear at the left end of the corresponding lists in Figure 1.8*a*. Then the Difference Engine computes $\Delta^1 y_2 = 3 + 2 = 5$ according to (1.2) and $y_2 = 1 + 3 = 4$ according to (1.3). It never has to recompute $\Delta^2 y_j$, which remains unchanged at 2 for all j. Once the values for $j = 2$ are known, the Difference Engine can calculate $\Delta^1 y_3$ and y_3, and so on indefinitely. At the end of the computation illustrated in Figure 1.8*a*, we have $y_6 = 36$, $\Delta^1 y_6 = 13$, and $\Delta^2 y_6 = 2$. One more iteration yields $\Delta^1 y_7 = 13 + 2 = 15$ and $y_7 = 36 + 13 = 49$, which is, of course, 7^2.

Figure 1.8*b* outlines the essential features of a small Difference Engine that executes the foregoing procedure. It contains several *registers;* these are memory devices, each of which stores a single number. Here we need three registers to store the three numbers y_j, $\Delta^1 y_j$, and $\Delta^2 y_j$. The engine employs a pair of processing units called *adders* to perform the addition steps specified by (1.2) and (1.3). Each adder takes the contents of two registers, calculates their sum, and returns it to one of the registers so that the sum becomes that register's new contents. The arrows in Figure 1.8*b* indicate the manner in which information flows through the Difference Engine during operation.

We can easily show that the nth difference of x^n is always a constant, from which it follows the nth difference of any nth-order polynomial of the form

$$y(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \tag{1.4}$$

is also a constant K. A Difference Engine can therefore calculate $y(x)$ by evaluating a set of n difference equations of the form

$$\Delta^i y_j = \Delta^i y_{j-1} + \Delta^{i+1} y_{j-1}$$

where $0 \le i \le n - 1$, $\Delta^0 y_j = y_j$, and $\Delta^n y_j = K$. Many useful functions encountered in science and engineering are expressible as polynomials like (1.4) and therefore can be evaluated by the method of differences. The trigonometric sine function, for instance, can be written as

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots \tag{1.5}$$

The first k terms of (1.5) form a $(2k - 1)$th-order polynomial that approximates $\sin x$. A higher-order polynomial will produce more accurate results.

Babbage constructed a small portion of his first Difference Engine in 1832, which served as a demonstration prototype. He later designed an improved version (Difference Engine No. 2), which was to handle seventh-order polynomials and have 31 decimal digits of accuracy. Like some of his modern successors, Babbage conceived his computers on a grand scale that strained the limits of the technology—and funds—available to build them. He never completed Difference Engine No. 2, mainly because of the difficulty of fabricating its 4000 or so high-precision mechanical parts. The complexity of this 3-ton machine can be appreciated from Figure 1.9, which is based on one of Babbage's own drawings. The vertical "figure-wheel columns" constitute the registers for storing 31-digit numbers, while the adders are implemented by the rack-and-lever mechanism underneath. It was not until 1991 that a working version of Difference Engine No. 2 was actually constructed (at a cost of around $500,000) by the Science Museum in London to celebrate the bicentennial of Babbage's birth [Swade 1993].

The Analytical Engine. Another reason for Babbage's failure to complete his Difference Engine was that he conceived of a much more powerful computing machine that he called the Analytical Engine. This machine is considered to be the first general-purpose programmable computer ever designed.

The overall organization of the Analytical Engine is outlined in Figure 1.10. It contains in rudimentary form many of the basic features found in all subsequent computers—compare Figure 1.10 to Figure 1.2. The main components of the

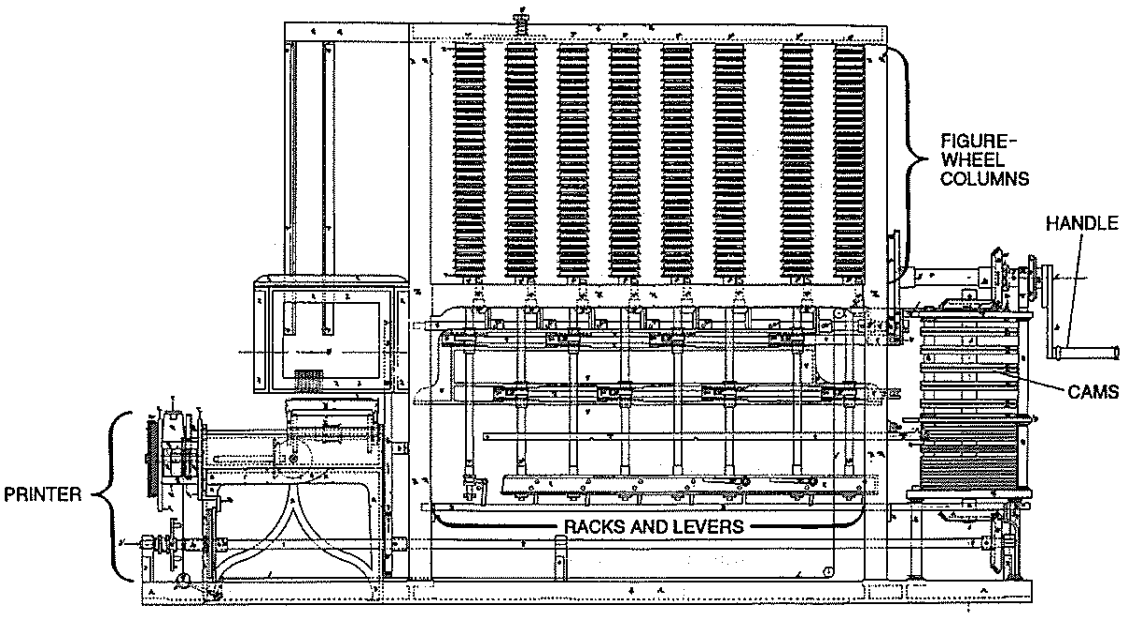


Figure 1.9
Diagram by Babbage of Difference Engine No. 2 [Courtesy of the National Science Museum/Science & Society Picture Library].

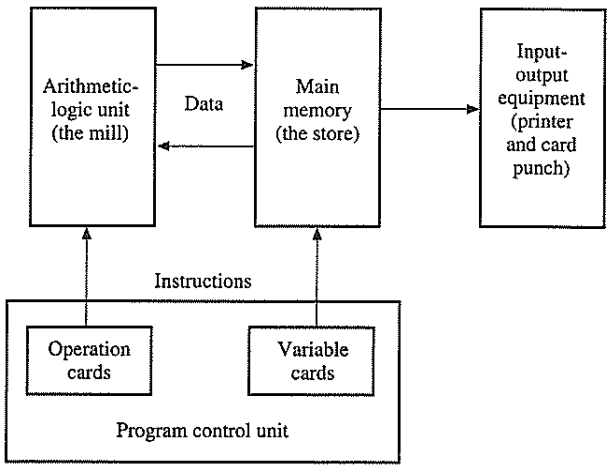


Figure 1.10
Structure of Babbage's Analytical Engine.

Analytical Engine are a memory called the *store* and an ALU called the *mill*; the latter was designed to perform the four basic arithmetic operations. To control the operation of the machine, Babbage proposed to use punched cards of a type developed earlier for controlling the Jacquard loom. A program for the Analytical Engine was composed of two sequences of punched cards: *operation cards* used to select the operation to be performed by the mill, and *variable cards* to specify the locations in the store from which inputs were to be taken or results sent. An action such as $a \times b = c$ would be specified by an instruction consisting of an operation card denoting multiply and variable cards specifying the store locations assigned to *a, b,* and *c*. Babbage intended the results to be printed on paper or punched on cards.

One of Babbage's key innovations was a mechanism to enable a program to alter the sequence of its operations automatically. In modern terms he conceived of conditional-branch or **if–then–else** instructions. They were to be implemented by testing the sign of a computed number; one course of action was taken if the sign were positive, another if negative. Babbage also designed a device to advance or reverse the flow of punched cards to permit branching to any desired instruction within a program. This type of conditional branching distinguishes the Analytical Engine from the Difference Engine: a program for the latter could only execute a fixed set of instructions in a fixed order. Conditional branching is the source of much of the power of the Analytical Engine and subsequent computers; it is the feature that makes them truly general purpose.

Again Babbage proposed to build the Analytical Engine on a grand scale using the same mechanical technology as his Difference Engines. The store, for instance, was to have a capacity of a thousand 50-digit numbers. He estimated that the addition of two numbers would take a second, and multiplication, a minute. Babbage spent much of the latter half of his life refining the design of the Analytical Engine, but only a small part of it was ever constructed.

Later developments. Many improvements were made to the design of four-function mechanical calculators in the 19th century, which led to their widespread

use. The Comptometer, designed by the American Dorr E. Felt (1862–1930) in 1885, was one of the earliest calculators to use depressible keys for entering data and commands; it also printed its results on paper. A later innovation was the use of electric motors to drive the mechanical components, thus making calculators "electromechanical" and greatly increasing their speed. Another important development was the use of punched cards to sort and tabulate large amounts of data. The punched-card tabulating machine was invented by Herman Hollerith (1860–1929) and used to process the data collected in the 1880 United States census. In 1896 Hollerith formed a company to manufacture his electromechanical equipment. This company subsequently merged with several others and in 1924 was renamed the International Business Machines Corp. (IBM).

No significant attempts to build general-purpose, program-controlled computers were made after Babbage's death until the 1930s [Randell 1982]. In Germany, Konrad Zuse built a small mechanical computer, the Z1, in 1938, apparently unaware of Babbage's work. Unlike previous computers, the Z1 used binary, instead of decimal, arithmetic. A subsequent Zuse machine, the Z3, which was completed in 1941, is believed to have been the first operational general-purpose computer. Zuse's work was interrupted by the Second World War and had little influence on the subsequent development of computers. Of great influence, however, was a general-purpose electromechanical computer proposed in 1937 by Howard Aiken (1900–73), a physicist at Harvard University. Aiken arranged to have IBM construct this computer according to his basic design. Work began on Aiken's Automatic Sequence Controlled Calculator, later called the Harvard Mark I, in 1939; it became operational in 1944. Like Babbage's machines, the Mark I employed decimal counter wheels for its main memory. It could store seventy-two 23-digit numbers. The computer was controlled by a punched paper tape, which combined the functions of Babbage's operation and variable cards. Although less ambitious than the Analytical Engine, the Mark I was in many ways the realization of Babbage's dream.

1.2.2 Electronic Computers

A mechanical computer has two serious drawbacks: Its computing speed is limited by the inertia of its moving parts, and the transmission of digital information by mechanical means is quite unreliable. In an electronic computer, on the other hand, the "moving parts" are electrons, which can be transmitted and processed reliably at speeds approaching that of light (300,000 km/s). Electronic devices such as the vacuum tube or electronic valve, which was developed in the early 1900s, permit the processing and storage of digital signals at speeds far exceeding those of any mechanical device.

The first generation. The earliest attempt to construct an electronic computer using vacuum tubes appears to have been made in the late 1930s by John V. Atanasoff (1903–95) at Iowa State University [Randell 1982]. This special-purpose machine was intended for solving linear equations, but it was never completed. The first widely known general-purpose electronic computer was the Electronic Numerical Integrator and Calculator (ENIAC) that John W. Mauchly (1907–80) and J.

Presper Eckert (1919–95) built at the University of Pennsylvania. Like Babbage's Difference Engine, a motivation for the ENIAC was the need to construct mathematical tables automatically—this time ballistic tables for the U.S. Army. Work on the ENIAC began in 1943 and was completed in 1946. It was an enormous machine weighing about 30 tons and containing more than 18,000 vacuum tubes. It was also substantially faster than any previous computer. While the Harvard Mark I required about 3 s to perform a 10-digit multiplication, the ENIAC required only 3 ms.

The ENIAC had a set of electronic memory units called accumulators with a combined capacity of twenty 10-digit decimal numbers. Each digit was stored in a 10-bit ring counter, where the binary pattern 1000000000 denoted the decimal digit 0, 0100000000 denoted 1, 0010000000 denoted 2, and so on. The ring counter was the electronic equivalent of the decimal counter wheel of earlier mechanical calculators. Like counter wheels, the ENIAC's accumulators combined the function of storage with addition and subtraction. Additional units performed multiplication, division, and the extraction of square roots. The ENIAC was programmed by the cumbersome process of plugging and unplugging cables and by manually setting a master programming unit to specify multistep operations. Results were punched on cards or printed on an electric typewriter. In computing ability, the ENIAC is roughly comparable to a modern pocket calculator!

Like the Analytical Engine, the Harvard Mark I and the ENIAC stored their programs and data in separate memories. Entering or altering the programs was a tedious task. The idea of storing programs and their data in the same high-speed memory—the *stored-program* concept—is attributed to the ENIAC's designers, notably the Hungarian-born mathematician John von Neumann (1903–57) who was a consultant to the ENIAC project. The concept was first published in a 1945 proposal by von Neumann for a new computer, the Electronic Discrete Variable Computer (EDVAC). Besides facilitating the programming process, the stored-program concept enables a program to modify its own instructions. (Such self-modifying programs have undesirable aspects, however, and are rarely used.)

The EDVAC differed from most of its predecessors in that it stored and processed numbers in true binary or base 2 form. To minimize hardware costs, data was processed serially, or bit by bit. The EDVAC had two kinds of memory: a fast main memory with a capacity of 1024 or 1K words (numbers or instructions) and a slower secondary memory with a capacity of 20K words. Prior to their execution, a set of instructions forming a program was placed in the EDVAC's main memory. The instructions were then transferred one at a time from the main memory to the CPU for execution. Each instruction had a well-defined structure of the form

$$A_1 \quad A_2 \quad A_3 \quad A_4 \quad \text{OP} \tag{1.6}$$

meaning: Perform the operation OP (addition, multiplication, etc.) on the contents of main memory locations or "addresses" A_1 and A_2 and then place the result in memory location A_3. The fourth address A_4 specifies the location of the next instruction to be executed. A variant of this instruction format implements conditional branching, where the next instruction address is either A_3 or A_4, depending on the relative sizes of the numbers stored in A_1 and A_2. Yet another instruction type specifies input-output operations that transfer words between main memory and secondary memory or between secondary memory and a printer. The EDVAC became operational in 1951.

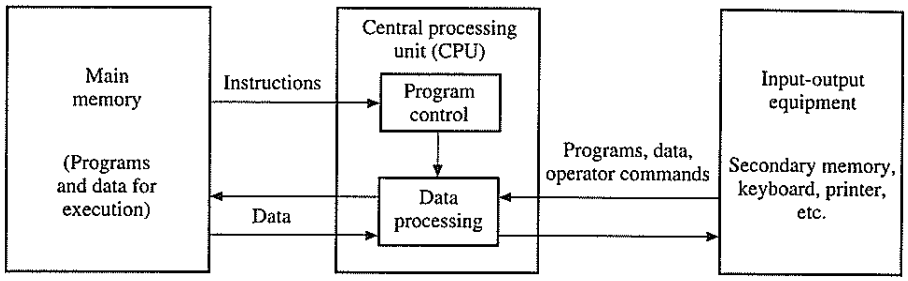


Figure 1.11
Organization of a first-generation computer.

In 1947 von Neumann and his colleagues began to design a new stored-program electronic computer, now referred to as the IAS computer, at the Institute for Advanced Studies in Princeton. Like the EDVAC, it had the general structure depicted in Figure 1.11, with a CPU for executing instructions, a main memory for storing active programs, a secondary memory for backup storage, and miscellaneous input-output equipment. Unlike the EDVAC, however, the IAS machine was designed to process all bits of a binary number simultaneously or in parallel. Several reports describing the IAS computer were published [Burks, Goldstine, and von Neumann 1946] and had far-reaching influence. In its overall design the IAS is quite modern, and it can be regarded as the prototype of most subsequent general-purpose computers. Because of its pervasive influence, we will examine the IAS computer in more detail below.

In the late 1940s and 1950s, the number of vacuum-tube computers grew rapidly. We usually refer to computers of this period as *first generation,* reflecting a somewhat narrow view of computer history [Randell 1982]. Besides those mentioned already, important early computers included the Whirlwind I constructed at the Massachusetts Institute of Technology and a series of machines designed at Manchester University [Siewiorek, Bell, and Newell 1982]. In 1947 Eckert and Mauchly formed Eckert-Mauchly Corp. to manufacture computers commercially. Their first successful product was the Universal Automatic Computer (UNIVAC) delivered in 1951. IBM, which had earlier constructed the Harvard Mark I, introduced its first electronic stored-program computer, the 701, in 1953. Besides their use of vacuum tubes in the CPU, first-generation computers experimented with various technologies for main and secondary memory. The Whirlwind introduced the ferrite-core memory in which a bit of information was stored in magnetic form on a tiny ring of magnetic material. Ferrite cores remained the principal technology for main memories until the 1970s.

The earliest computers had their instructions written in a binary code known as *machine language* that could be executed directly. An instruction in machine language meaning "add the contents of two memory locations" might take the form

00111011000000001001100100000111

Machine-language programs are extremely difficult for humans to write and so are very error-prone. A substantial improvement is obtained by allowing operations and operand addresses to be expressed in an easily understood symbolic

form such as

ADD X1, X2

This symbolic format, which is referred to as an *assembly language,* came into use in the 1950s, as computer programs were growing in size and complexity. An assembly language requires a special "system" program (an assembler) to translate it into machine language before it can be executed. First-generation computers were supplied with almost no system software; often little more than an assembler was available to the user. Moreover, assembly and machine languages varied widely from computer to computer so first-generation software was far from portable.

The IAS computer. It is instructive to examine the design of the Princeton IAS computer. Because of the size and high cost of the CPU's electronic hardware, the designers made every effort to keep the CPU, and therefore its instruction set, small and simple. Cost also heavily influenced the design of the memory subsystem. Because fast memories were expensive, the size of the main memory (initially 1K words but expandable to 4K) was less than most users would have wished. Consequently, a larger (16K words) but cheaper secondary memory based on an electromechanical magnetic drum technology was provided for bulk storage. Essentially similar cost-performance considerations remain central to computer design today, despite vast changes over the years in the available technologies and their actual costs.

The basic unit of information in the IAS computer is a 40-bit *word,* which is the standard packet of information stored in a memory location or transferred in one step between the CPU and the main memory M. Each location in M can be used to store either a single 40-bit number or else a pair of 20-bit instructions. The IAS's number format is *fixed-point,* meaning that it contains an implicit binary point in some fixed position. Numbers are usually treated as signed binary fractions lying between –1 and +1, but they can also be interpreted as integers. Examples of the IAS's binary number format are

01101000000 0000000000 0000000000 0000000000 = +.8125

10011000000 0000000000 0000000000 0000000000 = –0.8125

Numbers that lie outside the range ±1 must be suitably scaled for processing by IAS.

An IAS instruction consists of an 8-bit *opcode* (operation code) OP followed by a 12-bit *address* A that identifies one of up to 2^{12} = 4K 40-bit words stored in M. The IAS computer thus has a *one-address* instruction format, which we represent symbolically as

OP A

This format may appear very restrictive compared with the EDVAC's four-address instruction format (1.6). The IAS's shorter format clearly saves memory space. The fact that it does not restrict the machine's computational capabilities follows from two key aspects of the IAS's design that have been incorporated into all later computers:

1. The CPU contains a small set of high-speed storage devices called *registers,* which serve as implicit storage locations for operands and results. For example,

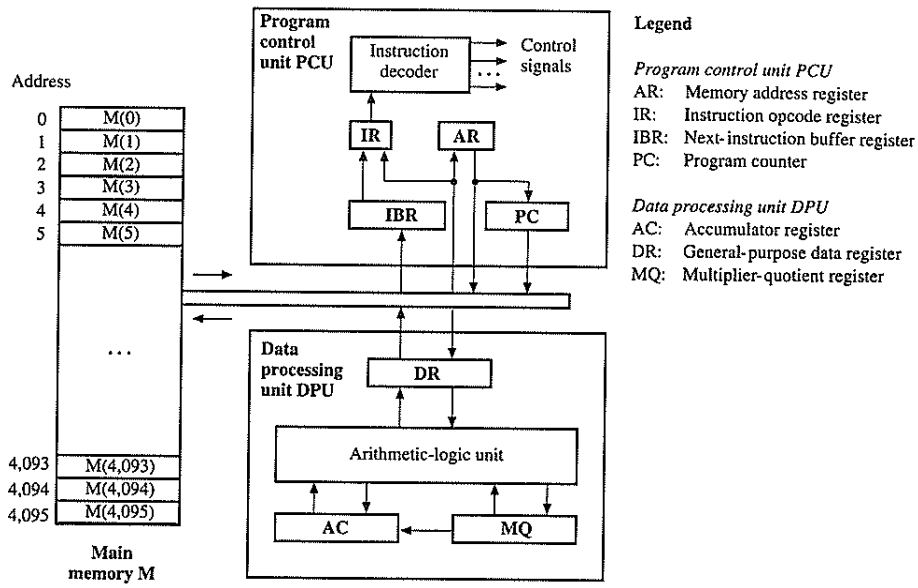


Figure 1.12
Organization of the CPU and main memory of the IAS computer.

an instruction of the form

$$\text{ADD} \quad \text{X} \tag{1.7}$$

fetches the contents of the memory location X from main memory and adds it to the contents of a CPU register known as the accumulator register AC. The resulting sum is then placed in AC. Hence X and AC play the role of the three memory addresses A_1, A_2, and A_3 appearing in (1.6).

2. A program's instructions are stored in M in approximately the sequence in which they are executed. Hence the address of the next instruction word is usually that of the current instruction plus one. Therefore, the EDVAC's next-instruction address A_4 can be replaced by a CPU register (the program counter PC), which stores the address of the current instruction word and is incremented by one when the CPU needs a new instruction word. Branch instructions are provided to permit the instruction execution sequence to be varied.

Figure 1.12 gives a programmer's perspective of the IAS, using modern notation and terminology. One of the two main parts of the CPU is responsible for fetching instructions from main memory and interpreting them; this part is variously known as the *program control unit (PCU)* or the *I-unit* (instruction unit). The second major part of the CPU is responsible for executing instructions and is known as the *data processing unit (DPU)*, the *datapath,* or the *E-unit* (execution unit).

The major components of the PCU are the *instruction register IR,* which stores the opcode that is currently being executed, and the *program counter PC,* which automatically stores and keeps track of the address of the next instruction to be

fetched. The PCU has circuits to interpret opcodes and to issue control signals to the DPU, M, and other circuits involved in executing instructions. The PCU can modify the instruction execution sequence when required to do so by branch instructions. There is also a 12-bit *address register* AR in the PCU that holds the address of a data operand to be fetched from or sent to main memory. Because the IAS has the unusual feature of fetching two instructions at a time from M, it contains a second register, the *instruction buffer register* (IBR), for holding a second instruction.

The main components of the DPU are the ALU, which contains the circuits that perform addition, multiplication, etc., as required by the possible opcodes, and several *data registers* to store data words temporarily during program execution. The IAS has two general-purpose 40-bit data registers: AC (accumulator) and DR (data register). It also has a third, special-purpose data register MQ (multiplier-quotient) intended for use by multiply and divide instructions.

Main memory M is a 4096 word or 4096 × 40-bit array of storage cells. Each storage location in M is associated with a unique 12-bit number called its *address*, which the CPU uses to refer to that location. To read data from a particular memory location, the CPU must have its address X (which it can store in PC or AR). The CPU accomplishes the read operation by sending the address X to M accompanied by control signals that specify "read." M responds by transferring a copy of M(X), the word stored at address X, to the CPU, where it is loaded into DR. In a similar way the CPU writes new data into main memory by sending to M the destination address X, a data word D to be stored, and control signals that specify "write."

Instruction set. The IAS machine had around 30 types of instructions. These were chosen to provide a balance between application needs—the machine's focus was on numerical computation for scientific applications—and computer hardware costs as they existed at the time. To represent instructions, we will use a notation called a *hardware description language* (HDL) or *register-transfer language* (RTL) that approximates the assembly language used to prepare programs for the computer; the designers of the IAS computer also used such a descriptive language [Burks, Goldstine, and von Neumann 1946]. The HDL introduced here and used throughout this book is largely self-explanatory. Storage locations in M or the CPU are referred to by acronym. The transfer of information is denoted by the assignment symbol :=, which suggests the left-going arrow ←. Hence, AC := MQ means transfer (copy) the contents of register MQ to register AC without altering the contents of MQ. Elements of main memory M are denoted by appending to M an address in parentheses. For example, M(X) denotes the 40-bit memory word with address X, while M(X,0:19) denotes the half-word consisting of bits 0 through 19 of M(X).

Figure 1.13 illustrates our descriptive notation for a simple three-instruction IAS program that adds two numbers. The numbers to be added are stored in the main memory locations with addresses 100 and 101; their sum is placed in memory location 102. Note the role played by the accumulator AC as an intermediate source and destination of data.

The set of instructions defined for the IAS computer is given in Figure 1.14 [Burks, Goldstine, and von Neumann 1946], omitting only those intended for

Instruction	Comment
AC := M(100)	Load the contents of memory location 100 into the accumulator.
AC := AC + M(101)	Add the contents of memory location 101 to the accumulator.
M(102) := AC	Store the contents of the accumulator in memory location 102.

Figure 1.13
An IAS program to add two numbers stored in main memory.

input-output operations. We have divided them into three categories: data-transfer, data-processing, and program-control instructions. Observe that some instructions have all their operands in CPU registers; others have one operand in memory location M(X). The data-processing instructions do most of the "real" work; all the others play supporting roles. Because only one memory address X can be specified at a time, multioperand instructions such as add and multiply must use CPU registers to store some of their operands. Consequently, it is necessary to precede or follow a typical data-processing instruction by data-transfer instructions that load input operands into CPU registers or transfer results from the CPU to memory. This requirement is illustrated by the add operation in Figure 1.13, where two data-transfer instructions and one add instruction are needed to accomplish a single addition operation. Hence the IAS like many of its successors contains quite a few data-transfer instructions whose purpose is to shuttle information unchanged (except possibly in sign) between CPU registers and memory. The IAS's data-processing instructions perform all the basic operations of arithmetic on signed 40-bit numbers. The IAS can also perform nonnumerical operations, but with some difficulty, because it treats all its operands as numbers.

The group of instructions called program-control or branch instructions determine the sequence in which instructions are executed. Recall that the program counter PC specifies the address of the next instruction to be executed. Instructions are normally executed in a fixed order determined by incrementing the program counter PC. The program-control instructions are designed to change this order. The IAS has two unconditional branch instructions (also called "jump" or "go to" instructions), which load part of X into PC and cause the next instruction to be taken from the left half or right half of M(X). The two conditional branch instructions permit a program branch to occur if and only if AC contains a nonnegative number. These instructions allow the results of a computation to alter the instruction execution sequence and so are of great importance.

The last two instructions listed in Figure 1.14 are "address-modify" instructions that permit 12-bit addresses to be computed in the CPU and then inserted directly into instructions stored in M. Address-modify instructions allow a program to alter itself, enabling, for example, the same data-processing instruction to refer to different operands at different times. Modifying programs during their execution is now considered obsolete and undesirable, but it was an important feature of early computers like IAS.

Instruction execution. The IAS fetches and executes instructions in several steps that form an *instruction cycle*. Since two instructions are packed into a 40-bit

Instruction type	Instruction	Description
Data transfer	AC := MQ	Transfer contents of register MQ to register AC.
	AC := M(X)	Transfer contents of memory location X to AC.
	M(X) := AC	Transfer contents of AC to memory location X.
	MQ := M(X)	Transfer M(X) to MQ.
	AC := –M(X)	Transfer minus M(X) to AC.
	AC := |M(X)|	Transfer absolute value of M(X) to AC.
	AC := –|M(X)|	Transfer minus |M(X)| to AC.
Data processing	AC := AC + M(X)	Add M(X) to AC putting the result in AC.
	AC := AC + |M(X)|	Add absolute value of M(X) to AC.
	AC := AC – M(X)	Subtract M(X) from AC.
	AC := AC – |M(X)|	Subtract |M(X)| from AC.
	AC.MQ := MQ × M(X)	Multiply MQ by M(X) putting the double-word product in AC and MQ.
	MQ.AC := AC ÷ M(X)	Divide AC by M(X) putting the quotient in AC and the remainder in MQ.
	AC := AC × 2	Multiply AC by two (1-bit left shift).
	AC := AC ÷ 2	Divide AC by two (1-bit right shift).
Program control	**go to** M(X, 0:19)	Take next instruction from left half of M(X)
	go to M(X, 20:39)	Take next instruction from right half of M(X).
	if AC ≥ 0 **then** **go to** M(X, 0:19)	If AC contains a nonnegative number, then take next instruction from left half of M(X).
	if AC ≥ 0 **then** **go to** M(X, 20:39)	If AC contains a nonnegative number, then take next instruction from right half of M(X).
	M(X, 8:19) := AC (28:39)	Replace left instruction address field in M(X) by 12 right-most bits of AC.
	M(X, 28:39) := AC(28:39)	Replace right instruction address field in M(X) by 12 right-most bits of AC.

Figure 1.14
Instruction set of the IAS computer.

word, the IAS fetches two instructions in each instruction cycle. One instruction has its opcode placed in the instruction register IR and its address field (if any) placed in the address register AR. The other instruction is transferred to the IBR register for possible later execution. Whenever the next instruction needed by the CPU is not in IBR, the program counter PC is incremented to generate the next instruction address.

Once the desired instruction has been loaded into the CPU, its execution phase begins. The PCU decodes the instruction's opcode, and the PCU's subsequent actions depend on the opcode's bit pattern. Typically, these actions involve one or two register-transfer (micro) operations of the form $S := f(S_1, S_2, \ldots, S_k)$, where the

S_i's are the locations of operands and f is a data-transfer or arithmetic operation. For example, the add instruction AC := AC + M(X) is executed by the following two register-transfer operations:

$$DR := M(AR);$$

$$AC := AC + DR$$

First, the contents of the memory location M(AR) specified by the address register AR are transferred to the data register DR. Then the contents of DR and the accumulator AC are added via the DPU's arithmetic-logic unit, and the result is placed in AC. The unconditional branch instruction **go to** M(X,0:19) has an address field containing some address X; after fetching this instruction, X is placed in AR. This instruction is then executed via the single register-transfer operation PC := AR, which makes PC point to the desired next instruction stored in the half-word M(X,0:19).

EXAMPLE 1.4 AN IAS PROGRAM TO PERFORM VECTOR ADDITION. Let A = A(1), A(2), ..., A(1000) and B = B(1), B(2), ..., B(1000) be two vectors, that is, one-dimensional arrays, of numbers to be added. The desired vector sum C = A + B is defined by

$$C(1), C(2), \ldots, C(1000) = A(1) + B(1), A(2) + B(2), \ldots, A(1000) + B(1000)$$

For simplicity we will assume that the numbers processed by the IAS, including the vector elements A(I), B(I), and C(I) are 40-bit integers, and that the input vectors are prestored in the IAS's main memory M. We need to perform the add operation

$$C(I) := A(I) + B(I)$$

1000 times, specifically for I = 1, 2, ..., 1000. Using the operations available in the IAS instruction set, the basic addition step above can be realized by the following three-instruction sequence (compare Figure 1.13):

$$\begin{aligned} AC &:= A(I) \\ AC &:= AC + B(I) \\ C(I) &:= AC \end{aligned} \qquad (1.8)$$

Clearly, a program with 1000 copies of these three instructions, each with a different index I, would implement the vector addition. However, such a program, besides being very inconvenient to write, would not fit in M along with the three vectors A, B, and C. We need some type of loop or iterative program that contains one copy of (1.8) but can modify the index I to step through all elements of the vectors.

Figure 1.15 shows such a program. The vectors A, B, and C are assumed to be stored sequentially, beginning at locations 1001, 2001, and 3001, respectively. The symbol to the left of each instruction in Figure 1.15 is its location in M. For instance, 2L (2R) denotes the left (right) half of M(2). The first location M(0) is used to store a counting variable N and is initially set to 999. N is systematically decremented by one after each addition step; when it reaches –1, the program halts. The conditional branch instruction in 5R performs this termination test. The three instructions in locations 3L, 3R, and 4L are the key ones that implement (1.8). The address-modify instructions in 8L, 9L, and 10L decrement the address parts of the three instructions in 3L, 3R, and

Location	Instruction or data	Comment
0	999	Constant (count N).
1	1	Constant.
2	1000	Constant.
3L	AC := M(2000)	Load A(I) into AC.
3R	AC := AC + M(3000)	Compute A(I) + B(I).
4L	M(4000) := AC	Store sum C(I).
4R	AC := M(0)	Load count N into AC.
5L	AC := AC – M(1)	Decrement count N by one.
5R	**if** AC ≥ 0 **then go to** M(6, 20:39)	Test N and branch to 6R if nonnegative.
6L	**go to** M(6, 0:19)	Halt.
6R	M(0) := AC	Update count N.
7L	AC := AC + M(1)	Increment AC by one.
7R	AC := AC + M(2)	Modify address in 3L.
8L	M(3, 8:19) := AC(28:39)	
8R	AC := AC + M(2)	Modify address in 3R.
9L	M(3, 28:39) := AC(28:39)	
9R	AC := AC + M(2)	Modify address in 4L.
10L	M(4, 8:19) := AC(28:39)	
10R	**go to** M(3, 0:19)	Branch to 3L.

Figure 1.15
An IAS program for vector addition.

4L, respectively. Thus the program continuously modifies itself during execution. Figure 1.15 shows the program before execution commences. At the end of the computation, the first three instructions will have changed to the following:

3L AC := M(1001)

3R AC := AC + M(2001)

4L M(3001) := AC

Critique. In the years that have elapsed since the IAS computer was completed, numerous improvements in computer design have appeared. Hindsight enables us to point out some of the IAS's shortcomings.

1. The program self-modification process illustrated in the preceding example for decrementing the index I is inefficient. In general, writing and debugging a program whose instructions change themselves is difficult and error-prone. Further, before every execution of the program, the original version must be reloaded into M. Later computers employ special instruction types and registers for index control, which eliminates the need for address-modify instructions.

2. The small amount of storage space in the CPU results in a great deal of unproductive data-transfer traffic between the CPU and main memory M; it also adds to program length. Later computers have more CPU registers and a special memory called a *cache* that acts as a buffer between the CPU registers and M.
3. No facilities were provided for structuring programs. For example, the IAS has no procedure call or return instructions to link different programs.
4. The instruction set is biased toward numerical computation. Programs for nonnumerical tasks such as text processing were difficult to write and executed slowly.
5. Input-output (IO) instructions were considered of minor importance—in fact, they are not mentioned in Burks, Goldstine, and von Neumann [1946] beyond noting that they are necessary. IAS had two basic and rather inefficient IO instruction types [Estrin 1953]. The input instruction INPUT(X, N) transferred N words from an input device to the CPU and then to N consecutive main memory locations, starting at address X. The OUTPUT(X, N) instruction transferred N consecutive words from the memory region with starting address X to an output device.

1.2.3 The Later Generations

In spite of their design deficiencies and the limitations on size and speed imposed by early electronic technology, the IAS and other first-generation computers introduced many features that are central to later computers: the use of a CPU with a small set of registers, a separate main memory for instruction and data storage, and an instruction set with a limited range of operations and addressing capabilities. Indeed the term *von Neumann computer* has become synonymous with a computer of conventional design.

The second generation. Computer hardware and software evolved rapidly after the introduction of the first commercial computers around 1950. The vacuum tube quickly gave way to the transistor, which was invented at Bell Laboratories in 1947, and a second generation of computers based on transistors superseded the first generation of vacuum tube–based machines. Like a vacuum tube, a transistor serves as a high-speed electronic switch for binary signals, but it is smaller, cheaper, sturdier, and requires much less power than a vacuum tube. Similar progress occurred in the field of memory technology, with ferrite cores becoming the dominant technology for main memories until superseded by all-transistor memories in the 1970s. Magnetic disks became the principal technology for secondary memories, a position that they continue to hold.

Besides better electronic circuits, the second generation, which spans the decade 1954–64, introduced some important changes in the design of CPUs and their instruction sets. The IAS computer still served as the basic model, but more registers were added to the CPU to facilitate data and address manipulation. For example, index registers were introduced to store an index variable I of the kind appearing in the statement

$$C(I) := A(I) + B(I) \tag{1.9}$$

Index registers make it possible to have *indexed* instructions, which increment or decrement a designated index I before (or after) they execute their main operation. Consequently, repeated execution of an indexed operation like (1.9) allows it to step automatically through a large array of data. The index value I is stored in a CPU register and not in the program, so the program itself does not change during execution. Another innovation was the introduction of two program-control instructions, now referred to as *call* and *return*, to facilitate the linking of programs; see also Example 1.5.

"Scientific" computers of the second generation, such as the IBM 7094 which appeared in 1962, introduced floating-point number formats and supporting instructions to facilitate numerical processing. Floating point is a type of scientific notation where a number such as 0.0000000709 is denoted by 7.09×10^{-8}. A *floating-point number* consists of a pair of fixed-point numbers, a mantissa M and an exponent E, and has the value $M \times B^{-E}$. In the preceding example $M = 7.09$, $E = -8$, and $B = 10$. In their computer representation M and E are encoded in binary and embedded in a word of suitable size; the base B is implicit. Floating-point numbers eliminate the need for number scaling; floating-point numbers are automatically scaled as they are processed. The hardware needed to implement floating-point arithmetic instructions directly is relatively expensive. Consequently, many computers (then and now) rely on software subroutines to implement floating-point operations via fixed-point arithmetic.

Input-output operations. Computer designers soon realized that IO operations, that is, the transfer of information to and from peripheral devices like printers and secondary memory, can severely degrade overall computer performance if done inefficiently. Most IO transfers have main memory as their final source or destination and involve the transfer of large blocks of information, for instance, moving a program from secondary to main memory for execution. Such a transfer can take place via the CPU, as in the following fragment of a hypothetical IO program:

Location	Instruction	Comment
LOOP	AC := D(I)	Input word from IO device D into AC.
	M(I) := AC	Output word from AC to main memory.
	I := I + 1	Increment index I.
	if I ≤ MAX **go to** LOOP	Test for end of loop.

Clearly, the IO operation ties up the CPU with a trivial data-transfer task. Moreover, many IO devices transfer data at low speeds compared to that of the CPU because of their inherent reliance on electromechanical rather than electronic technology. Thus the CPU is idle most of the time when executing an IO program directed at a relatively slow device such as a printer. To eliminate this bottleneck, computers such as the IBM 7094 introduced *input-output processors* (*IOPs*), or *channels* in IBM parlance, which are special-purpose processing units designed exclusively to control IO operations. They do so by executing IO programs (see preceding sample), but channeling the data through registers in the IO processor, rather than through the CPU. Hence IO data transfers can take place independently

of the CPU, permitting the CPU to execute user programs while IO operations are taking place.

Programming languages. An important development of the mid-1950s was the introduction of "high level" programming languages, which are far easier to use than assembly languages because they permit programs to be written in a form much closer to a computer user's problem specification. A high-level language is intended to be usable on many different computers. A special program called a *compiler* translates a user program from the high-level language in which it is written into the machine language of the particular computer on which the program is to be executed.

The first successful high-level programming language was FORTRAN (from FORmula TRANslation), developed by an IBM group under the direction of John Backus from 1954 to 1957. FORTRAN permits the specification of numerical algorithms in a form approximating normal algebraic notation. For example, the vector addition task in Figure 1.16 can be expressed by the following two-line program in the original version of FORTRAN:

```
      DO 5 I = 1, 1000
5     C(I) = A(I) + B(I)
```

FORTRAN has continued to be widely used for scientific programming and, like natural languages, it has changed over the years. The version of FORTRAN known as FORTRAN90 introduced in 1990 replaces the preceding DO loop with the single vector statement

$$C(1:1000) = A(1:1000) + B(1:1000) \tag{1.10}$$

High-level languages were also developed in the 1950s for business applications. These are characterized by instructions that resemble English statements and operate on textual as well as numerical data. One of the earliest such languages was Common Business Oriented Language (COBOL), which was defined in 1959 by a group representing computer users and manufacturers and sponsored by the U.S. Department of Defense. Like FORTRAN, COBOL has continued (in various revised forms) to be among the most widely used programming languages. FORTRAN and COBOL are the forerunners of other important high-level languages, including Basic, Pascal, C, and Java, the latter dating from the mid-1990s.

EXAMPLE 1.5 A NONSTANDARD ARCHITECTURE: STACK COMPUTERS. Although most computers follow the von Neumann model, a few alternatives were explored quite early in the electronic era. In the stack organization illustrated in Figure 1.16*a* a stack memory replaces the accumulator and other CPU registers used for temporary data storage. A *stack* resembles the array of contiguous storage locations found in main memory, but it has a very different mode of access. Stack locations have no external addresses; all read and write operations refer to one end of the stack called the *top of the stack* TOS. A *push* operation writes a word into the next unused location TOS + 1 and causes this location to become the new TOS. A *pop* operation reads the word stored in the current TOS and causes the location TOS – 1 below TOS to become the new TOS. Hence TOS serves as a dynamic entry point to the stack, which expands and contracts in response to push and pop operations, respectively. The region above the stack (shaded in Figure 1.16*a*) is unused, but it is available for future use. Among

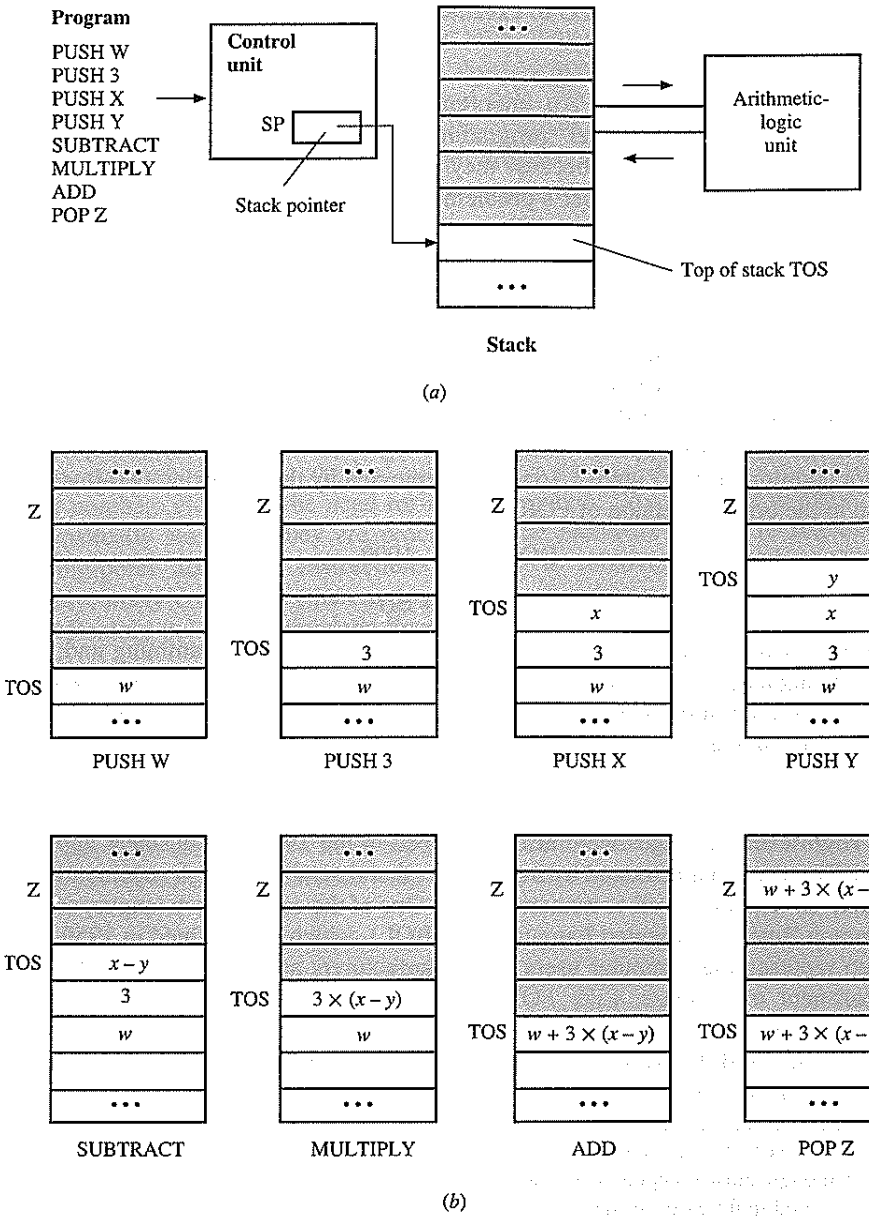


Figure 1.16
(*a*) Essentials of a stack processor; (*b*) stack states during the execution of $z := w + 3 \times (x - y)$.

the earliest stack computers was the Burroughs B5000, first delivered in 1963 [Siewiorek, Bell, and Newell 1982]; a recent example is the Sun picoJava microprocessor designed for fast execution of compiled Java code [O'Connor and Tremblay 1997].

In a stack machine an instruction's operands are stored at the top of the stack, so data-processing instructions do not need to contain addresses as they do in a conven-

tional, von Neumann computer. The add operation $x + y$ is specified for a stack machine by the following sequence of three instructions:

PUSH x

PUSH y

ADD

The first PUSH instruction loads x into TOS. Execution of PUSH y causes x's location to become TOS – 1 and places y in the new TOS immediately above x. To execute ADD, the top two words of the stack are popped into the ALU where they are added, and the sum is pushed back into the stack. Hence in the preceding program fragment, ADD computes $x + y$, which replaces x and y at the top of the stack. The electronic circuits that carry out these actions can be complicated, but they are hidden from the programmer. A key component is a register called the *stack pointer* SP which stores the internal address of TOS, and automatically adjusts the TOS for every push and pop operation. A program counter PC keeps track of instruction addresses in the usual manner.

A stack computer evaluates arithmetic and other expressions using a format known as *Polish notation*, named after the Polish logician Jan Lukasiewicz (1878–1956). Instead of placing an operator between its operands as in $x + y$, the operator is placed to the right of its operands as in $x\ y\ +$. A more complex expression such as $z := w + 3 \times (x - y)$ becomes

$$z\ w\ 3\ x\ y\ -\ \times\ +\ := \tag{1.11}$$

in Polish notation, and the expression is evaluated from left to right. Note that Polish notation eliminates the need for parentheses. The Polish expression (1.11) leads directly to the eight-instruction stack program shown in Figure 1.16*a*. The step-by-step execution of this code fragment is illustrated in Figure 1.16*b*. Here it is assumed that w,x,y,z represent the values of operands stored at the memory addresses W,X,Y, and Z, respectively.

Stack computers such as the B5000 employ a main memory M to store programs and data in much the same way as a conventional computer. For cost reasons, the CPU contains only a small stack—a two-word stack in the B5000 case—implemented by high-speed registers. However, the stack expands automatically into M by treating some main memory locations as if they were stack registers and coupling them with those in the CPU. While stack processors can evaluate complex expressions such as (1.11) efficiently, they are generally slower than von Neumann machines, especially when executing vector operations such as (1.10). Large stack computers were successfully marketed for many years, notably by Burroughs Corp. However, the stack concept eventually became widely used in only two specialized applications:

1. Pocket calculators sometimes employ a stack organization to take advantage of the conciseness of Polish notation when entering data and commands manually via a keypad.
2. Stacks are included in most conventional computers to implement subroutine call and return instructions. In its basic form, a call-subroutine instruction takes the form CALL SUB. It first saves the current contents of PC—the calling routine's return address—by pushing it into a stack region of M that is under the control of a stack pointer SP. Then SUB, the start address of the subroutine being called, is loaded into PC, and its execution begins. Control is returned to the calling program when the subroutine executes a RETURN instruction, whose function is to pop the return address from the top of the stack and load it back into PC.

System management. In the early days, all programs or jobs were run separately, and the computer had to be halted and prepared manually for each new program to be executed. With the improvements in IO equipment and programming methodology that came with the second-generation machines, it became feasible to prepare a batch of jobs in advance, store them on magnetic tape, and then have the computer process the jobs in one continuous sequence, placing the results on another magnetic tape. This mode of system management is termed *batch processing*. Batch processing requires the use of a supervisory program called a *batch monitor*, which is permanently resident in main memory. A batch monitor is a rudimentary version of an *operating system*, a system program (as opposed to a user or application program) designed to manage a computer's resources efficiently and provide a set of common services to its users.

Later operating systems were designed to enable a single CPU to process a set of independent user programs concurrently, a technique called *multiprogramming*. It recognizes that a typical program alternates between program execution when it requires use of the CPU, and IO operations when it requires use of an IOP. Multiprogramming is accomplished by the CPU temporarily suspending execution of its current program, beginning execution of a second program, and returning to the first program later. Whenever possible, a suspended program is assigned an IOP, which performs any needed IO functions. Consequently, multiprogramming attempts to keep a CPU (usually viewed as the computer's most precious resource) and any available IOPs busy by overlapping CPU and IO operations. Multiprogrammed computers that process many user programs concurrently and support users at interactive terminals or workstations are sometimes called *time-sharing* systems.

The third generation. This generation is traditionally associated with the introduction of integrated circuits (ICs), which first appeared commercially in 1961, to replace the discrete electronic circuits used in second-generation computers. The transistor continued as the basic switching device, but ICs allowed large numbers of transistors and associated components to be combined on a tiny piece of semiconductor material, usually silicon. IC technology initiated a long-term trend in computer design toward smaller size, higher speed, and lower hardware cost.

Perhaps the most significant event of the third-generation period (which began around 1965) was recognition of the need to standardize computers in order to allow software to be developed and used more efficiently. By the mid-1960s a few dozen manufacturers of computers around the world were each producing machines that were incompatible with those of other manufacturers. The cost of writing and maintaining programs for a particular computer—the software cost—began to exceed that of the computer's hardware. At the same time many big users of computers, such as banks and insurance companies, were creating huge amounts of application software on which their business operations were becoming very dependent. Switching to a different computer and making one's old software obsolete was thus an increasingly unattractive proposition.

Influenced by these considerations, IBM developed (at a cost of about $5 billion) what was to be the most influential third-generation computer, the System/360, which it announced in 1964 and delivered the following year; see Figure 1.17. System/360 was actually a series of computers distinguished by model numbers

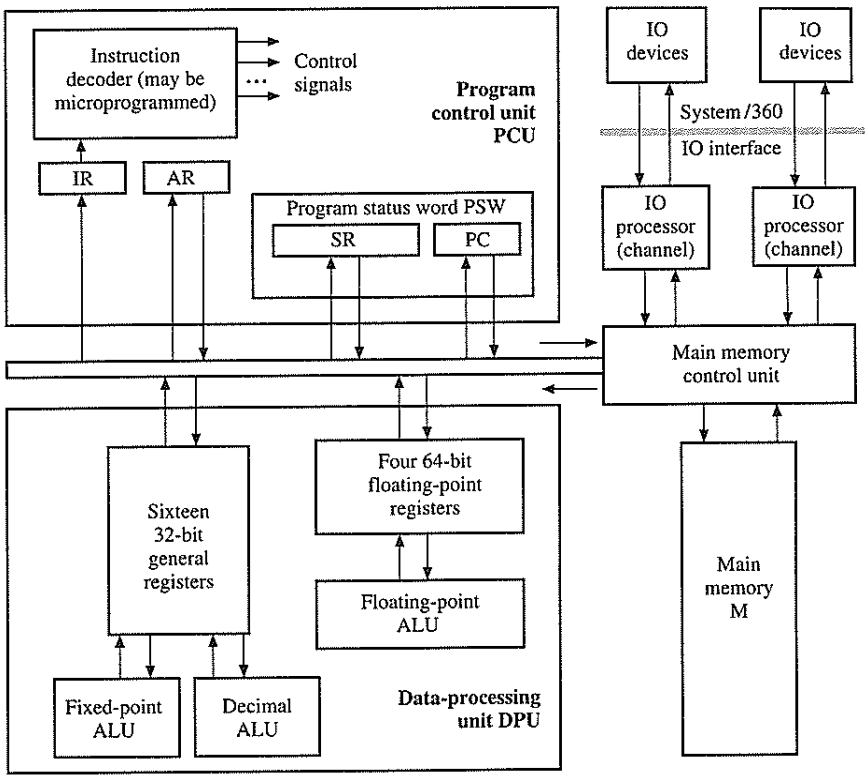


Figure 1.17
Structure of the IBM System/360.

and intended to cover a wide range of computing performance [Siewiorek, Bell, and Newell 1982; Prasad 1989]. The various System/360 models were designed to be *software compatible* with one another, meaning that all models in the series shared a common instruction set. Programs written for one model could be run without modification on any other; only the execution time, memory usage, and the like would change. Software compatibility enabled computer owners to upgrade their systems without having to rewrite large amounts of software. The System/360 models also used a common operating system, OS/360, and the manufacturer supplied specialized software to support such widely used applications as transaction processing and database management. In addition, the System/360 models had many hardware characteristics in common, including the same interface for attaching IO devices.

While the System/360 standardized much of IBM's own product line, it also became a de facto standard for large computers, now referred to as *mainframe* computers, produced by other manufacturers. The long list of makers of System/360-compatible machines includes such companies as Amdahl in the United States and Hitachi in Japan. The System/360 series was also remarkably long-lived. It evolved into various newer mainframe computer series introduced by IBM over the years, all of which maintained software compatibility with the original System/

360; for example, the System/370 introduced in 1970, the 4300 introduced in 1979, and the System/390 introduced in 1990.

The System/360 added only modestly to the basic principles of the von Neumann computer, but it established a number of widely followed conventions and design styles. It had about 200 distinct instruction types (opcodes) with many addressing modes and data types, including fixed-point and floating-point numbers of various sizes. It replaced the small and unstructured set of data registers (AC, MQ, etc.) found in earlier computers with a set of 16 identical general-purpose registers, all individually addressable. This is called the *general-register organization.* The System/360 had separate arithmetic-logic units for processing various data types; the fixed-point ALU was used for address computations including indexing. The 8-bit unit *byte* was defined as the smallest unit of information for data transmission and storage purposes. The System/360 also made 32 bits (4 bytes) the main CPU word size, so that 32 bits and "word" have become synonymous in the context of large computers.

The CPU had two major control states: a *supervisor state* for use by the operating system and a *user state* for executing application programs. Certain program-control instructions were "privileged" in that they could be executed only when the CPU was in supervisor state. These and other special control states gave rise to the concept of a *program status word* (PSW) which was stored in a special CPU register, now generally referred to as a *status register* (SR). The SR register encapsulated the key information used by the CPU to record exceptional conditions such as CPU-detected errors (an instruction attempting to divide by zero, for example), hardware faults detected by error-checking circuits, and urgent service requests or *interrupts* generated by IO devices.

Architecture versus implementation. With the advent of the third generation, a distinction between a computer's overall design and its implementation details became apparent. As defined by System/360's designers [Prasad 1989], the *architecture* of a computer is its structure and behavior as seen by a programmer working at the assembly-language level. The architecture includes the computer's instruction set, data formats, and addressing modes, as well as the general design of its CPU, main memory, and IO subsystems. The architecture therefore defines a conceptual model of a computer at a particular level of abstraction. A computer's *implementation*, on the other hand, refers to the logical and physical design techniques used to realize the architecture in any specific instance. The term *computer organization* also refers to the logical aspects of the implementation, but the boundary between the terms *architecture* and *organization* is vague.

Hence we can say that the models of the IBM System/360 series have a common architecture but different implementations. These differences reflect the existence of physical circuit technologies with different cost/performance ratios for constructing processing circuits and memories. To achieve instruction-set compatibility across many models, the System/360 also used an implementation technique called *microprogramming*. Originally proposed in the early 1950s by Maurice V. Wilkes at Cambridge University, microprogramming allows a CPU's program control unit PCU to be designed in a systematic and flexible way [Wilkes and Stringer 1953]. Low-level control sequences known as *microprograms* are placed in a special control memory in the PCU so that an instruction from the CPU's main

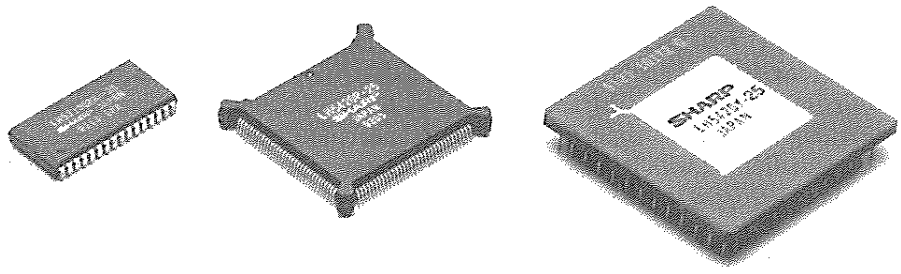
instruction set is executed by invoking and executing the corresponding microprogram. A CPU with no floating-point arithmetic circuits can execute floating-point instructions (albeit slowly) if microprograms are written to perform the desired floating-point operations by means of fixed-point arithmetic circuits. Microprogramming allowed the smaller System/360 models to implement the full System/360 instruction set with less hardware than the larger, faster models, some of which were not microprogrammed.

Other developments. The System/360 was typical of commercial computers aimed at both business and scientific applications. Efforts were also directed by various manufacturers towards the design of extremely powerful (and expensive) scientific computers, loosely termed *supercomputers*. Control Data Corp., for instance, produced a series of commercially successful supercomputers beginning with the CDC 6660 in 1964, and continuing into the 1980s with the subsequent CYBER series. These early supercomputers experimented with various types of parallel processing to improve their performance. One such technique called *pipelining* involves overlapping the execution of instructions from the same program within a specially designed CPU. Another technique, which allows instructions from different programs to be executed simultaneously, employs a computer with more than one CPU; such a computer is called a *multiprocessor*.

A contrasting development of this period was the mass production of small, low-cost computers called *minicomputers*. Their origins can be traced to the LINC (Laboratory Instrument Computer) developed at MIT in the early 1960s [Siewiorek, Bell, and Newell 1982]. This machine influenced the design of the PDP (Programmed Data Processor) series of small computers introduced by Digital Equipment Corp. (Digital) in 1965, which did much to establish the minicomputer market. Minicomputers are characterized by short word size—CPU word sizes of 8 and 16 bits were typical—limited hardware and software facilities, and small physical size. Most important, their low cost made them suitable for many new applications, such as the industrial process control where a computer is permanently assigned to one particular application. The Digital VAX series of minicomputers introduced in 1978 brought general-purpose computing to many small organizations that could not afford the high cost of a mainframe computer.

1.3 THE VLSI ERA

Since the 1960s the dominant technology for manufacturing computer logic and memory circuits has been the integrated circuit or IC. This technology has evolved steadily from ICs containing just a few transistors to those containing thousands or millions of transistors; the latter case is termed *very large-scale integration* or VLSI. The impact of VLSI on computer design and application has been profound. VLSI allows manufacturers to fabricate a CPU, main memory, or even all the electronic circuits of a computer, on a single IC that can be mass-produced at very low cost. This has resulted in new classes of machines ranging from portable personal computers to supercomputers that contain thousands of CPUs.



(*a*) (*b*) (*c*)

Figure 1.18
Some representative IC packages: (*a*) 32-pin small-outline J-lead (SOJ); (*b*) 132-pin plastic quad flatpack (PQFP); (*c*) 84-pin pin-grid array (PGA). [Courtesy of Sharp Electronics Corp.]

1.3.1 Integrated Circuits

The integrated circuit was invented in 1959 at Texas Instruments and Fairchild Corporations [Braun and McDonald 1982]. It quickly became the basic building block for computers of the third and subsequent generations. (The designation of computers by generation largely fell into disuse after the third generation.) An IC is an electronic circuit composed mainly of transistors that is manufactured in a tiny rectangle or *chip* of semiconductor material. The IC is mounted into a protective plastic or ceramic package, which provides electrical connection points called *pins* or *leads* that allow the IC to be connected to other ICs, to input-output devices like a keypad or screen, or to a power supply. Figure 1.18 depicts several representative IC packages. Typical chip dimensions are 10×10 mm, while a package like that of Figure 1.18*b* is approximately $30 \times 30 \times 4$ mm. The IC package is often considerably bigger than the chip it contains because of the space taken by the pins. The PGA package of Figure 1.18*c* has an array of pins (as many as 300 or more) projecting from its underside. A *multichip module* is a package containing several IC chips attached to a substrate that provides mechanical support, as well as electrical connections between the chips. Packaged ICs are often mounted on a *printed circuit board* that serves to support and interconnect the ICs. A contemporary computer consists of a set of ICs, a set of IO devices, and a power supply. The number of ICs can range from one IC to several thousand, depending on the computer's size and the IC types it uses.

IC density. An integrated circuit is roughly characterized by its *density,* defined as the number of transistors contained in the chip. As manufacturing techniques improved over the years, the size of the transistors in an IC and their interconnecting wires shrank, eventually reaching dimensions below a micron or 1 μm. (By comparison, the width of a human hair is about 75 μm.) Consequently, IC densities have increased steadily, while chip size has varied very little.

The earliest ICs—the first commercial IC appeared in 1961—contained fewer than 100 transistors and employed *small-scale integration* or SSI. The terms *medium-scale, large-scale,* and *very-large-scale integration* (MSI, LSI and VLSI,

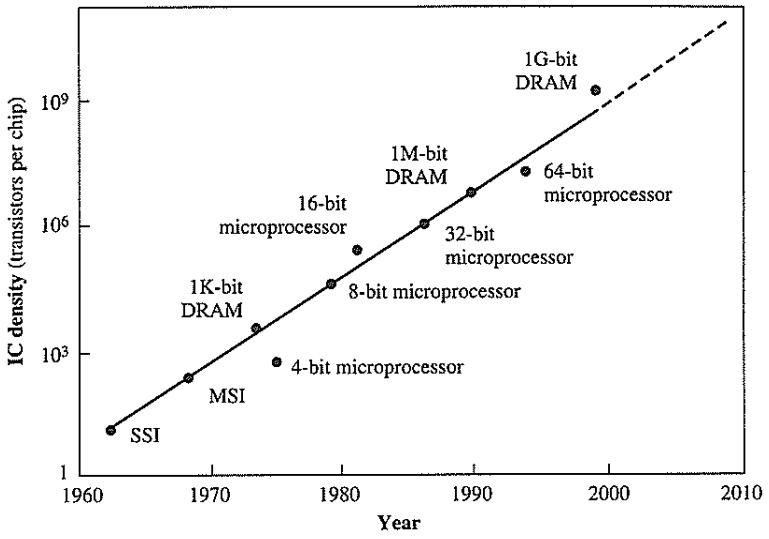


Figure 1.19
Evolution of the density of commercial ICs.

respectively) are applied to ICs containing hundreds, thousands, and millions of transistors, respectively. The boundaries between these IC classes are loose, and VLSI often serves as a catchall term for very dense circuits. Because their manufacture is highly automated—it resembles a printing process—ICs can be manufactured in high volume at low cost per circuit. Indeed, except for the latest and densest circuits, the cost of an IC has stayed fairly constant over the years, implying that newer generations of ICs deliver far greater value (measured by computing performance or storage capacity) per unit cost than their predecessors did.

Figure 1.19 shows the evolution of IC density as measured by two of the densest chip types: the dynamic random-access memory (DRAM), a basic component of main memories, and the single-chip CPU or *microprocessor*. Around 1970 it became possible to manufacture all the electronic circuits for a pocket calculator on a single IC chip. This development was quickly followed by single-chip DRAMs and microprocessors. As Figure 1.19 shows, the capacity of the largest available DRAM chip was $1K = 2^{10}$ bits in 1970 and has been growing steadily since then, reaching $1M = 2^{20}$ bits around 1985. A similar growth has occurred in the complexity of microprocessors. The first microprocessor, Intel's 4004, which was introduced in 1971, was designed to process 4-bit words. The Japanese calculator manufacturer Busicom commissioned the 4004 microprocessor, but after Busicom's early demise, Intel successfully marketed the 4004 as a programmable controller to replace standard, nonprogrammable logic circuits. As IC technology improved and chip density increased, the complexity and performance of one-chip microprocessors increased steadily, as reflected in the increase in CPU word size to 8 and then 16 bits by the mid-1980s. By 1990 manufacturers could fabricate the entire CPU of a System/360-class computer, along with part of its main memory, on a single IC. The combination of a CPU, memory, and IO circuits in one IC (or a small number of ICs) is called a *microcomputer*.

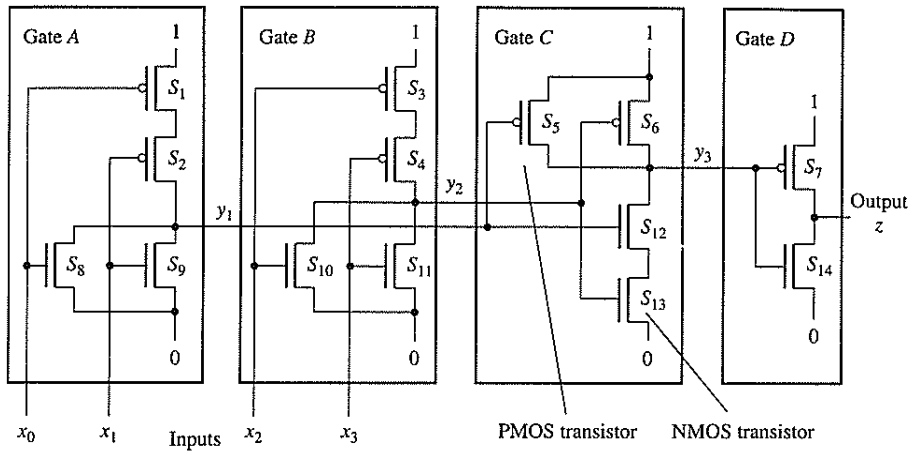
IC families. Within IC technology several subtechnologies exist that are distinguished by the transistor and circuit types they employ. Two of the most important of these technologies are bipolar and unipolar; the latter is normally referred to as MOS (metal-oxide-semiconductor) after its physical structure. Both bipolar and MOS circuits have transistors as their basic elements. They differ, however, in the polarities of the electric charges associated with the primary carriers of electrical signals within their transistors. *Bipolar circuits* use both negative carriers (electrons) and positive carriers (holes). *MOS circuits,* on the other hand, use only one type of charge carrier: positive in the case of P-type MOS (PMOS) and negative in the case of N-type MOS (NMOS). Various bipolar and MOS IC circuit types or *IC families* have been developed that provide trade-offs among density, operating speed, power consumption, and manufacturing cost. An MOS family that efficiently combines PMOS and NMOS transistors in the same IC is *complementary MOS* or *CMOS*. This technology came into widespread use in the 1980s and has been the technology of choice for microprocessors and other VLSI ICs since then because of its combination of high density, high speed, and very low power consumption [Weste and Eshragian 1992].

EXAMPLE 1.6 A ZERO-DETECTION CIRCUIT EMPLOYING CMOS TECHNOLOGY. To illustrate the role of transistors in computing, we examine a small CMOS circuit whose function is to detect when a 4-bit word $x_0x_1x_2x_3$ becomes zero. The circuit's output z should be 1 when $x_0x_1x_2x_3 = 0000$; it should be 0 for the other 15 combinations of input values. Zero detection is quite a common operation in data processing. For example, it is used to determine when a program loop terminates, as in the **if** statement (location 5R) appearing in the IAS program of Figure 1.15.

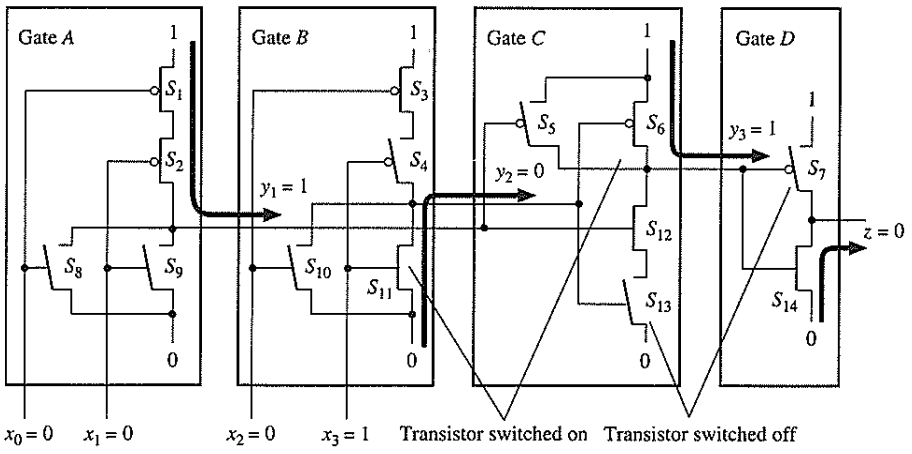
Figure 1.20 shows a particular implementation *ZD* of zero detection using a representative CMOS subfamily known as *static CMOS*. The circuit is shown in standard symbolic form in Figure 1.20*a*. It consists of equal numbers of PMOS transistors denoted S_1:S_7 and NMOS transistors denoted S_8:S_{14}. Each transistor acts as an on-off switch with three terminals, where the center terminal c controls the switch's state. When turned on, a signal propagation path is created between the transistor's upper and lower terminals; when turned off, that path is broken. An NMOS transistor is turned on by applying 1 to its control terminal c; it is turned off by applying 0 to c. A PMOS transistor, on the other hand, is turned on by $c = 0$ and turned off by $c = 1$.

Each set of input signals applied to *ZD* causes some transistors to switch on and others to switch off, which creates various signal paths through the circuit. In Figure 1.20 the constant signals 0 and 1 are applied at various points in *ZD*. (These signals are derived from *ZD*'s electrical power supply.) The 0/1 signals "flow" through the circuit along the paths created by the transistors and determine various internal signal values, as well as the value applied to the main output line z. Figure 1.20*b* shows the signals and signal transmission paths produced by $x_0x_1x_2x_3 = 0001$. The first input signal $x_0 = 0$ is applied to PMOS transistor S_1 and NMOS transistor S_8; hence S_1 is turned on and S_8 is turned off. Similarly, $x_1 = 0$ turns S_2 on and S_9 off. A path is created through S_1 and S_2, which applies 1 to the internal line y_1, as shown by the left-most heavy arrow in Figure 1.20*b*. In the same way the remaining input combinations make $y_2 = 0$ and $y_3 = 1$. The latter signal is applied to the two right-most transistors turning S_7 off and S_{14} on, which creates a path from the zero source to the primary output line via S_{14}, so $z = 0$ as required.

If we change input x_3 from 1 to 0 in Figure 1.20*b*, the following chain of events occurs: S_4 turns on and S_{11} turns off, changing y_2 to 1. Then S_{13} turns on and S_6 turns off, making $y_3 = 0$. Finally, the new value of y_3 turns S_7 on and S_{14} off, so z becomes 1.



(*a*)



(*b*)

Figure 1.20
(*a*) CMOS circuit *ZD* for zero detection; (*b*) state of *ZD* with input combination $x_0x_1x_2x_3 = 0001$ making $z = 0$.

Hence the zero input combination $x_0x_1x_2x_3 = 0000$ makes $z = 1$ as required. It can readily be verified that no other input combination does this.

A transistor circuit like that of Figure 1.20 models the behavior of a digital circuit at a low level of abstraction called the *switch level.* Because many of the ICs of interest contain huge numbers of transistors, it is rarely practical to analyze their computing functions at the switch level. Instead, we move to higher abstraction levels, two of which are illustrated in Figure 1.21. At the *gate* or *logic* level illustrated by Figure 1.21*a*, we represent certain common subcircuits by symbolic

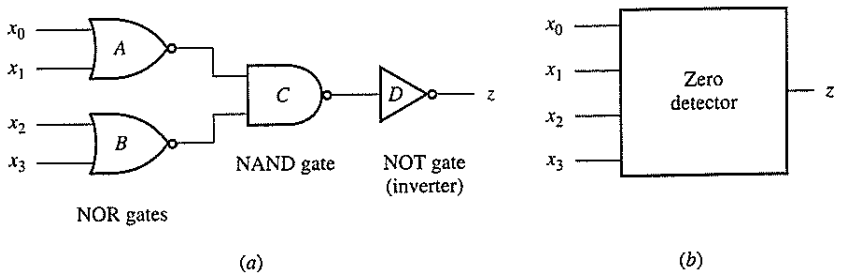


Figure 1.21
The zero-detection circuit of Figure 1.20 modeled at (*a*) the gate level and (*b*) the register level of abstraction.

components called (logic) *gates*. This particular logic circuit comprises four gates *A*, *B*, *C*, and *D* of three different types as indicated; note that each gate type has a distinct graphic symbol. In moving from the switch level, we collapse a multitransistor circuit into a single gate and discard all its internal details. A key advantage of the logic level is that it is *technology independent*, so it can be used equally well to describe the behavior of any IC family. In dealing with computer design, we also use an even higher level of abstraction known as the *register* or *register-transfer* level. It treats the entire zero-detection circuit as a primitive or indivisible component, as in Figure 1.21*b*. The register level is the level at which we describe the internal workings of a CPU or other processor as, for example, in Figures 1.2 and 1.17. Observe that the primitive components (represented by boxes) in these diagrams include registers, ALUs, and the like. When we treat an entire CPU, memory, or computer as a primitive component, we have moved to the highest level of abstraction, which is called the *processor* or *system* level.

1.3.2 Processor Architecture

By 1980 computers were classified into three main types: mainframe computers, minicomputers, and microcomputers. The term *mainframe* was applied to the traditional "large" computer system, often containing thousands of ICs and costing millions of dollars. It typically served as the central computing facility for an organization such as a university, a factory, or a bank. Mainframes were then room-sized machines placed in special computer centers and not directly accessible to the average user. The minicomputer was a smaller (desk size) and slower version of the mainframe, but its relatively low cost (hundreds of thousands of dollars) made it suitable as a "departmental" computer to be shared by a group of users—in a small business, for example. The microcomputer was even smaller, slower, and cheaper (a few thousand dollars), packing all the electronics of a computer into a handful of ICs, including microprocessor (CPU), memory, and IO chips.

Personal computers. Microcomputer technology gave rise to a new class of general-purpose machines called *personal computers* (PCs), which are intended for a single user. These small, inexpensive computers are designed to sit on an office desk or fold into a compact form to be carried. The more powerful desktop computers intended for scientific computing are referred to as *workstations*. A typical

PC has the von Neumann organization, with a microprocessor, a multimegabyte main memory, and an assortment of IO devices: a keyboard, a video monitor or screen, a magnetic or optical disk drive unit for high-capacity secondary memory, and interface circuits for connecting the PC to printers and to other computers. Personal computers have proliferated to the point that, in the more developed societies, they are present in most offices and many homes. Two of the main applications of PCs are word processing, where personal computers have assumed and greatly expanded all the functions of the typewriter, and data-processing tasks like financial record keeping. They are also used for entertainment, education, and increasingly, communication with other computers via the World Wide Web.

Personal computers were introduced in the mid-1970s by a small electronics kit maker, MITS Inc. [Augarten 1984]. The MITS Altair computer was built around the Intel 8008, an early 8-bit microprocessor, and cost only $395 in kit form. The most successful personal computer family was the IBM PC series introduced in 1981. Following the precedent set by earlier IBM computers, it quickly become the de facto standard for this class of machine. A new factor also aided the standardization process—namely, IBM's decision to give the PC what came to be called an *open architecture*, by making its design specifications available to other manufacturers of computer hardware and software. As a result, the IBM PC became very popular, and many versions of it—the so-called *PC clones*—were produced by others, including startup companies that made the manufacture of low-cost PC clones their main business. The PC's open architecture also provided an incentive for the development of a vast amount of application-specific software from many sources. Indeed a new software industry emerged aimed at the mass-production of low-cost, self-contained programs aimed at specific applications of the IBM PC and a few other widely used computer families.

The IBM PC series is based on Intel Corp.'s 80X86 family of microprocessors, which began with the 8086 microprocessor introduced in 1978 and was followed by the 80286 (1983), the 80386 (1986), the 80486 (1989), and the Pentium[2] (1993) [Albert and Avnon 1993]; the Pentium II appeared in 1997. The IBM PC series is also distinguished by its use of the MS/DOS operating system and the Windows graphical user interface, both developed by Microsoft Corp. Another popular personal computer series is Apple Computer's Macintosh, introduced in 1984 and built around the Motorola 680X0 microprocessor family, whose evolution from the 68000 microprocessor (1979) parallels that of the 80X86/Pentium [Farrell 1984]. In 1994 the Macintosh CPU was changed to a new microprocessor known as the PowerPC.

Figure 1.22 shows the organization of a typical personal computer from the mid-1990s. Its legacy from earlier von Neumann computers is apparent—compare Figure 1.22 to Figure 1.17. At the core of this computer is a single-chip microprocessor such as the Pentium or PowerPC. As we will see, the microprocessor's internal (micro) architecture usually contains a number of speedup features not found in its predecessors. A system bus connects the microprocessor to a main memory based on semiconductor DRAM technology and to an IO subsystem. A separate IO bus, such as the industry standard PCI (peripheral component interconnect) "local"

[2]A legal ruling that microprocessor names that are numbers cannot have trademark protection, resulted in the 80486 being followed by a microprocessor called the Pentium rather than the 80586.

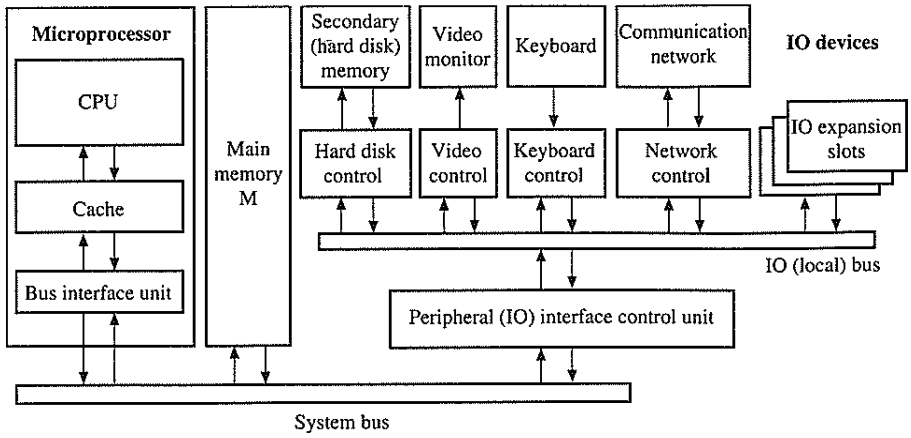


Figure 1.22
A typical personal computer system.

bus, connects directly to the IO devices and their individual controllers. The IO bus is linked to the system bus, to which the microprocessor and memory are attached via a special bus-to-bus control unit sometimes referred to as a *bridge*. The IO devices of a personal computer include the traditional keyboard, a CRT-based or flat-panel video monitor, and disk drive units for the hard and flexible (floppy) disk storage devices that constitute secondary memory. More recent additions to the IO device repertoire include drive units for CD-ROMs (compact disc read-only memories), which have extremely high capacity and allow sound and video images to be stored and retrieved efficiently. Other common audiovisual IO devices in personal computers are microphones, loudspeakers, video scanners, and the like, which are referred to as *multimedia* equipment.

Performance considerations. As processor hardware became much less expensive in the 1970s, thanks mainly to advances in VLSI technology (Figure 1.19), computer designers increased the use of complex, multistep instructions. This reduces N, the total number of instructions that must be executed for a given task, since a single complex instruction can replace several simpler ones. For example, a multiply instruction can replace a multiinstruction subroutine that implements multiplication by repeated execution of add instructions. Reducing N in this way tends to reduce overall program execution time T, as well as the time that the CPU spends fetching instructions and their operands from memory. The same advances in VLSI made it possible to add new features to old microprocessors, such as new instructions, data types, instruction sets, and addressing modes, while retaining the ability to execute programs written for the older machines.

The Intel 80X86/Pentium series illustrates the trend toward more complex instruction sets. The 1978-vintage 8086 microprocessor chip, which contained a mere 20,000 transistors, was designed to process 16-bit data words and had no instructions for operating on floating-point numbers [Morse et al. 1978]. Twenty-five years later, its direct descendant, the Pentium, contained over 3 million transistors, processed 32-bit and 64-bit words directly, and executed a comprehensive set of floating-point instructions [Albert and Avnon 1993]. The Pentium accumulated

most of the architectural features of its various predecessors in order to enable it to execute, with little or no modification, programs written for earlier 80X86-series machines. Reflecting these characteristics, the 80X86, 680X0, and most older computer series have been called *complex instruction set computers* (CISCs).[3]

By the 1980s it became apparent that complex instructions have certain disadvantages and that execution of even a small percentage of such instructions can sometimes *reduce* a computer's overall performance. To illustrate this condition, suppose that a particular microprocessor has only fast, simple instructions, each of which requires k time units, to execute. Thus the microprocessor can execute 100 instructions in $100k$ time units. Now suppose that 5 percent of the instructions are slow, complex instructions requiring $21k$ time units each. To execute an average set of 100 instructions therefore requires $(5 \times 21 + 95)k = 200k$ time units, assuming no other factors are involved. Consequently, the 5 percent of complex instructions can, as in this particular example, double the overall program execution time.

Thus while complex instructions reduce program size, this technology does not necessarily translate into faster program execution. Moreover, complex instructions require relatively complex processing circuits, which tend to put CISCs in the largest and most expensive IC category. These drawbacks were first recognized by John Cocke and his colleagues at IBM in the mid-1970s, who developed an experimental computer called 801 that aimed to achieve very fast overall performance via a streamlined instruction set that could be executed extremely fast [Cocke and Markstein 1990]. The 801 and subsequent machines with a similar design philosophy have been called *reduced instruction set computers* (RISCs). A number of commercially successful RISC microprocessors were introduced in the 1980s, including the IBM RISC System/6000 and SPARC, an "open" microprocessor developed by Sun Microsystems and based on RISC research at the University of California, Berkeley [Patterson 1985]. Many of the speedup features of RISC machines have found their way into other new computers, including such CISC microprocessors as the Pentium. Indeed, the term *RISC* is often used to refer to *any* computer with an instruction set and an associated CPU organization designed for very high performance; the actual size of the instruction set is relatively unimportant.

A computer's performance is also strongly affected by other factors besides its instruction set, especially the time required to move instructions and data between the CPU and main memory M and, to a lesser extent, the time required to move information between M and IO devices. It typically takes the CPU about five times longer to obtain a word from M than from one of its internal registers. This difference in speed has existed since the first electronic computers, despite strenuous efforts by circuit designers to develop memory devices and processor–memory interface circuits that are fast enough to keep up with the fastest microprocessors. Indeed the CPU–M speed disparity has become such a feature of standard (von Neumann) computers that is sometimes referred to as the *von Neumann bottleneck.* RISC computers usually limit access to main memory to a few load and store instructions; other instructions, including all data-processing and program-control instructions, must have their operands in CPU registers. This so-

[3]The public became aware of CISC complexity when a design flaw affecting the floating-point division instruction of the Pentium was discovered in 1994. The cost to Intel of this bug, including the replacement cost of Pentium chips already installed in PCs, was about $475 million.

called *load-store architecture* is intended to reduce the impact of the von Neumann bottleneck by reducing the total number of the memory accesses made by the CPU.

Performance measures. A rough indication of CPU speed is the number of "basic" operations that it can perform per unit of time. A typical basic operation is the fixed-point addition of the contents of two registers R1 and R2, as in the symbolic instruction

$$R1 := R1 + R2$$

Such operations are timed by a regular stream of signals (ticks or beats) issued by a central timing signal, the system *clock*. The speed of the clock is its frequency f measured in millions of ticks per second; the units for this are megahertz (MHz). Each tick of the clock triggers a basic operation; hence the time required to execute the operation is $1/f$ microseconds (μs). This value is called the *clock cycle* or *clock period* T_{clock}. For example, a computer clocked at 250 MHz can perform one basic operation in the clock period $T_{clock} = 1/250 = 0.004$ μs. Complicated operations such as division or operations on floating-point numbers can require more than one clock cycle to complete their execution.

Generally speaking, smaller electronic devices operate faster than larger ones, so the increase in IC chip density discussed above has been accompanied by a steady, but less dramatic, increase in clock speed. For example, from 1981 to 1995 microprocessor clock speeds increased from about 10 MHz to 100 MHz. Clock speeds of 1 gigahertz (1 GHz or 1000 MHz) and beyond are feasible using faster versions of current CMOS technology. It might therefore seem possible to achieve any desired processor speed simply by increasing the CPU clock frequency. However, the rate at which clock frequency is increasing due to IC technology improvements is relatively slow and may be approaching limits determined by the speed of light, power dissipation, and similar physical considerations. Extremely fast circuits also tend to be very expensive to manufacture.

The CPU's processing of an instruction involves several steps, each of which requires at least one clock cycle:

1. Fetch the instruction from main memory M.
2. Decode the instruction's opcode.
3. Load (read) from M any operands needed unless they are already in CPU registers.
4. Execute the instruction via a register-to-register operation using an appropriate functional unit of the CPU, such as a fixed-point adder.
5. Store (write) the results in M unless they are to be retained in CPU registers.

The fastest instructions have all their operands in CPU registers and can be executed by the CPU in a single clock cycle, so steps 1 to 3 all take one clock cycle. The slowest instructions require multiple memory accesses and multiple register-to-register operations to complete their execution. Consequently, measures of instruction execution performance are based on average figures, which are usually determined experimentally by measuring the run times of representative or *benchmark programs*. The more representative the programs are, that is, the more accurately they reflect real applications, the better the performance figures they provide.

Suppose that execution of a particular benchmark program or set (suite) of such programs Q on a given CPU takes T seconds and involves the execution of a total of N machine (object) instructions. Here N is the actual number of instructions executed, including repeated executions of the same instruction; it is not the number of instructions appearing in Q. As far as the typical computer user is concerned, the key performance goal is to minimize the total *program execution time* T. While T can be determined accurately only by measurement of Q's run time in actual or simulated execution, we can relate T to some basic parameters of the computer's architecture and implementation. One such parameter is the (average) number of *instructions executed per second*, which we denote by *IPS*. Clearly, $T = N/IPS$ s. Another common measure of the performance of a CPU is the average number of *cycles per instruction* or *CPI* needed to execute Q. Now $CPI = (f \times 10^6)/IPS$, where f is the CPU's clock frequency in MHz. Hence, the program execution time T is given by

$$T = \frac{N \times CPI}{f \times 10^6} \text{ s} \tag{1.12}$$

It is also common to measure CPU performance in terms of *millions of instructions executed per second*, denoted *MIPS*, where $MIPS = IPS \times 10^6$. Clearly $MIPS = f/CPI$.

Equation (1.12) indicates how the three separate factors software, architecture, and hardware technology jointly determine a computer's performance.

1. *Software*: The efficiency with which the programs are written and compiled into object code influences N, the number of instructions executed. Other factors being equal, reducing N tends to reduce the overall execution time T.
2. *Architecture*: The efficiency with which individual instructions are processed directly affects *CPI*, the number of cycles per instruction executed. Reducing *CPI* also tends to reduce T.
3. *Hardware*: The raw speed of the processor circuits determines f, the clock frequency. Increasing f tends to reduce T.

In general, the complex instruction sets of CISC processors aim to reduce N at the expense of *CPI*, whereas RISC processors aim to reduce *CPI* at the expense of N. Advances in VLSI technology affecting all types of computers tend to increase f.

Speedup techniques. A number of speed-enhancing features have been incorporated into the design of computers in recent years [Hwang 1993]; they are summarized in Figure 1.23. These methods were defined as far back as the 1960s and 1970s for use in mainframe computers. A *cache* is a memory unit placed between the CPU and main memory M and used to store instructions, data, or both. It has much smaller storage capacity than M, but it can be accessed (read from or written into) more rapidly and is often placed (at least partly) on the same chip as the CPU. The cache's effect is to reduce the average time required to access an instruction or data word, typically to just a single clock cycle. Special hardware and software techniques support the complex flow of information among M, the cache, and the registers of the CPU.

Another important speedup technique known as *pipelining* allows the processing of several instructions to be partially overlapped. Pipelining is most easily done

Feature	Objective	Description
Cache memory	To provide the CPU with faster access to instructions and data.	A cache is a memory unit inserted between the CPU and main memory M. It is faster than M but has less storage capacity.
Pipelined processing	To increase performance by allowing the processing of several instructions to be partially overlapped.	The CPU is constructed from independent subunits (stages), which can hold several instructions in different stages of execution.
Superscalar processing	To increase performance by allowing several instructions to be processed in parallel (full overlapping).	Multiple (pipelined) units are provided for instruction processing. Instructions can be issued simultaneously to each unit.

Figure 1.23
Some important speedup features of modern computers.

for a sequence of instructions of the same or similar types that employ a single E-unit, such as a floating-point processor. However, all the common steps involved in instruction processing by the CPU can be pipelined: instruction fetching (IF), instruction decoding (ID), operand loading (OL), execution (EX), and operand storing (OS). A pipelined system is often compared to an assembly line on which many products are in various stages of manufacture at the same time. In a nonpipelined CPU, instructions are executed in strict sequence, as depicted in Figure 1.24*a*. Pipelining permits the situation shown in Figure 1.24*b*, where each major step of

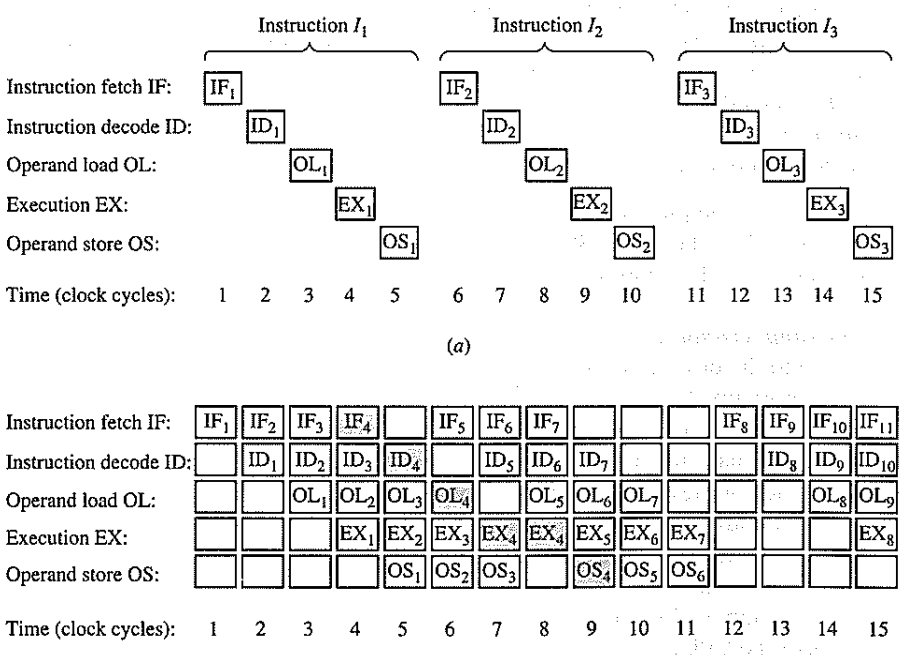


Figure 1.24
Instruction processing: (*a*) sequential or nonpipelined and (*b*) pipelined.

instruction processing is assigned to, and handled independently by, a separate subunit (stage) of the CPU pipeline. In this example, up to five instructions can be overlapped, provided the necessary pipeline stages are available. Note that performance-reducing delays occur, as in the case of instruction I_4 (shaded), which must use the EX stage for two consecutive cycles. A similar problem occurs in the case of branch instructions like I_7 in Figure 1.24*b*, where the outcome of I_7's EX step must be known before the location of the next instruction (I_8) to be processed can be identified.

A microprocessor's effective *MIPS* rate can also be increased by replicating various instruction-processing circuits so that several instructions can be in the same processing phase at the same time. This makes it possible to start the processing of, or *issue*, two or more instructions simultaneously or in parallel; in other words, the instructions can be completely overlapped. CPUs with this capability are said to be *superscalar.* (Note that two instructions in the same pipeline must be issued sequentially rather than in parallel.) For example, if the logic needed for the IF, ID, OL, EX, and OS steps is duplicated (with or without pipelining), then two instructions can be issued simultaneously. However, if the instructions are not independent, for example, if they share the same operands or one takes as input a result computed by the other, then delays not unlike those illustrated in Figure 1.24*b* can occur. Pipelining and superscalar design are both instances of *instruction-level parallelism.* The logic circuits needed to deal with parallelism of this kind add considerable complexity to the CPU's program control and execution units.

EXAMPLE 1.7 THE POWERPC MICROPROCESSOR SERIES [MOTOROLA 1993]. In the early 1990s Apple, IBM, and Motorola jointly developed the PowerPC. It is a family of single-chip microprocessors, including the 601, 603, and other models, which share a common architecture derived from the POWER architecture used in IBM's RISC System/6000 [Diefendorf, Oehler, and Hochsprung 1994; Weiss and Smith 1994]. Although it is also designated a RISC, the PowerPC has a large number of instructions—more than 200 distinct types, in fact—and its design is far from simple. Nevertheless, it exhibits the following features that are typical of contemporary RISC-style designs:

1. Instructions have a fixed length (32 bits or one word) and employ just a few opcode formats and addressing modes.
2. Only load and store instructions can access main memory; all other instructions must have their operands in CPU registers. This load/store architecture reduces the time devoted to accessing memory. This time is further reduced by the use of one or more levels of cache memory.
3. Instruction processing is heavily pipelined. For example, the PowerPC has an E-unit for integer (fixed-point) operations that has the four pipeline stages: fetch, decode, execute, and write results. Hence if an E-unit's pipeline can be kept full, a new result emerges from it every clock cycle, thus achieving the ideal performance level of one fully executed instruction per clock cycle.
4. The CPU contains several E-units—the number depends on the model—which allow it to issue several instructions simultaneously and puts the PowerPC in the superscalar category.

The organization shown in Figure 1.25 is typical of the early PowerPC models, such as the 601 and 603, which have three E-units: an integer execution unit, a floating-point unit, and a branch processing unit, allowing up to three instructions to be

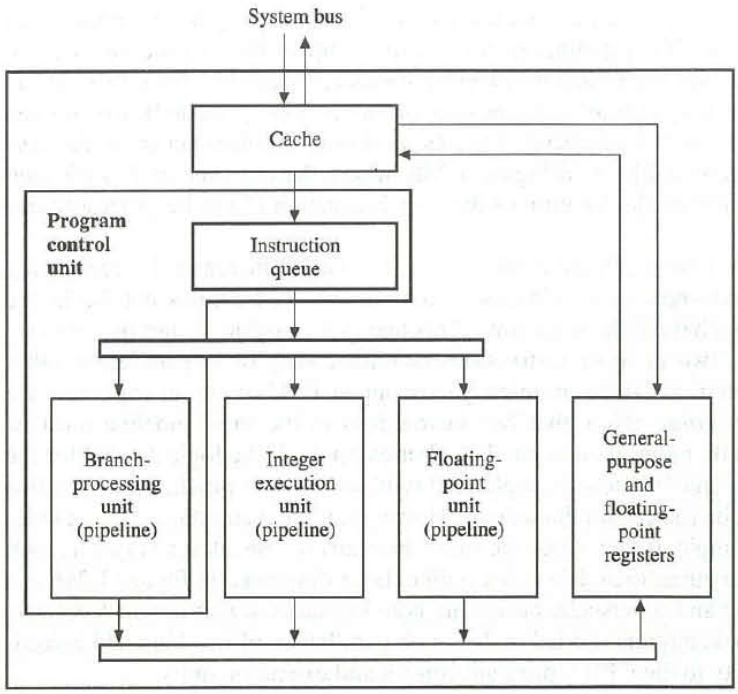


Figure 1.25
Overall organization of the PowerPC.

issued in the same clock cycle. The integer unit executes all fixed-point numerical and logic operations, including those associated with load-store instructions. Although part of the CPU's program control unit, the branch processing unit is considered an E-unit for branch instructions. Each PowerPC chip also contains a cache memory, whose size and organization vary with the model. For example, the PowerPC 603, which was introduced in 1995 and is aimed at low-power applications like laptop computers, has a 16 KB cache, half of which stores data while the other half stores instructions. A hint of the complexity of the 603 can be seen from Figure 1.26. It contains 1.6 million transistors in an IC chip of area 7.4×11.5 mm (in its earliest versions) and consumes less than 3 watts of power.

To illustrate the PowerPC's instruction set, consider the vector addition discussed earlier and expressed by the FORTRAN90 statement

$$C(1:1000) = A(1:1000) + B(1:1000)$$

Assume that each vector consists of 1000 double-precision (64-bit), floating-point numbers. An assembly-language program for the PowerPC that carries out this vector operation appears in Figure 1.27. (We have slightly simplified the language syntax here.) The last five instructions form the program's main loop and are executed 1000 times. The key data-processing instruction in this loop has the opcode fadd, and performs a double-precision, floating-point addition. All fadd's operands are in 64-bit floating-point registers, of which the PowerPC has 32, denoted fr0:fr31 here. The program communicates with memory via the instructions lw (load word), lfdu (load floating-point double-precision with update), and stfdu (store floating-point double-precision with update); these are just a few of the PowerPC's many types of load-store

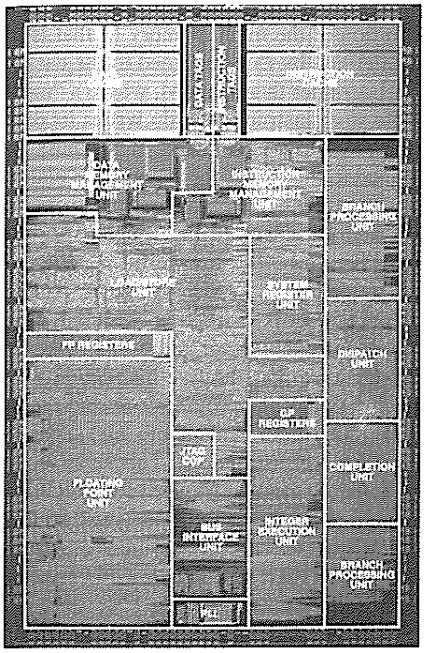


Figure 1.26
Photomicrograph of the PowerPC 603 microprocessor chip. [Courtesy of Motorola Inc.]

instructions. The PowerPC has 32 general-purpose registers r0:r31, several of which serve as memory address registers in our program. The update option, indicated by the *u* suffix on lfdu and stfdu invokes a kind of automatic indexing, which causes the contents of the memory address register to be initially incremented. For example, the instruction

lfdu fr1, 1(r5)

invokes the following two operations: increment the address register r5 and then load the data register fr1. In other words

r5 := r5 + 1; fr1 := mem(r5); (1.13)

Location	Instruction		Comment
	mtspr	CTR, #1000	Move vector length N = 1000 to special register CTR.
	lw	r5, #A	Load start address of vector A into general register r5.
	lw	r6, #B	Load start address of vector B into general register r6.
	lw	r7, #C	Load start address of vector C into general register r7.
LOOP	lfdu	fr1, 1(r5)	Load A(i + 1) into floating-point register fr1; update r5.
	lfdu	fr2, 1(r6)	Load B(i + 1) into floating-point register fr2; update r6.
	fadd	fr1, fr2, fr1	Perform floating-point addition fr1 := fr1 + fr2.
	stfdu	fr1, 1(r7)	Store fr1 as C(i + 1); update r7.
	bne	LOOP	Decrement CTR, then branch to LOOP if CTR ≠ 0.

Figure 1.27
A PowerPC program for vector addition.

The memory data denoted by mem(r5) in (1.13) is normally in the PowerPC's cache memory which, at any time, mimics a portion of the main memory M that is in active use. Thus if the current memory address defined by r5 is assigned to the cache, the data required by lfdu is fetched from the cache, rather than from M, where a "master" copy of the same data resides. Similarly, the store instruction stfdu writes its data into a cache location, although (eventually) the corresponding data in M must be updated. Should mem(r7) not be currently assigned to the cache, the PowerPC's elaborate memory access control automatically transfers data between M and the cache to assign the relevant portion of the processor's address space to the cache. The last instruction bne (branch if not equal) appearing in Figure 1.27 is a powerful conditional branch instruction. First bne automatically decrements the "special" register called CTR (counter) and tests it for zero. If CTR $\neq 0$, then the next instruction executed is the one stored in location LOOP. When CTR reaches zero, the vector addition terminates and the instruction following bne is executed. Observe that the five-instruction program loop typically resides in the cache for the duration of the program's execution.

As Figure 1.25 indicates, the Power PC has three (more in some models) separate E-units for executing integer, floating-point, and branch instructions. This superscalar design allows up to three separate instructions to be dispatched (issued) for execution in every clock cycle. Moreover, these E-units are pipelined to varying degrees, so that an active E-unit can contain several consecutive instructions in various stages of execution. Hence, for our vector addition task, we would expect to find the CPU concurrently executing several operations of the form

$$C(j) := A(j) + B(j),\ C(j+1) := A(j+1) + B(j+1),\ C(j+2) := A(j+2) + B(j+2), \ldots$$

The concurrency achieved, and therefore the execution time of the program, depend on various implementation details and cannot be determined from inspection of the program code alone.

The vector addition programs for the IAS (Figure 1.15) and the PowerPC (Figure 1.27) reflect the evolution of computer architecture over a 50-year period. The two programs are fundamentally similar in that each program is designed to loop N times through the three basic steps: load data from M, add data in CPU registers, and store results in M. The computers share the same basic features of the von Neumann architecture. However, the IAS machine has far fewer data types, a much weaker instruction set (especially in the area of program control), and essentially no instruction-level parallelism. The IAS lacks floating-point data formats and instructions, so a much more complicated IAS program would be required to handle double-precision, floating-point numbers comparable to those assumed in Figure 1.27. The IAS also lacks the following features of the PowerPC's instruction set: indexed addressing modes; conditional branch instructions that can decrement and test a variable; and powerful arithmetic instructions such as multiply, divide, and multiply-and-add. Note also the vast differences in physical size, performance, and cost between the IAS and PowerPC.

1.3.3 System Architecture

We next review the overall organization of contemporary computer systems, including those formed by linking computers together into large networks.

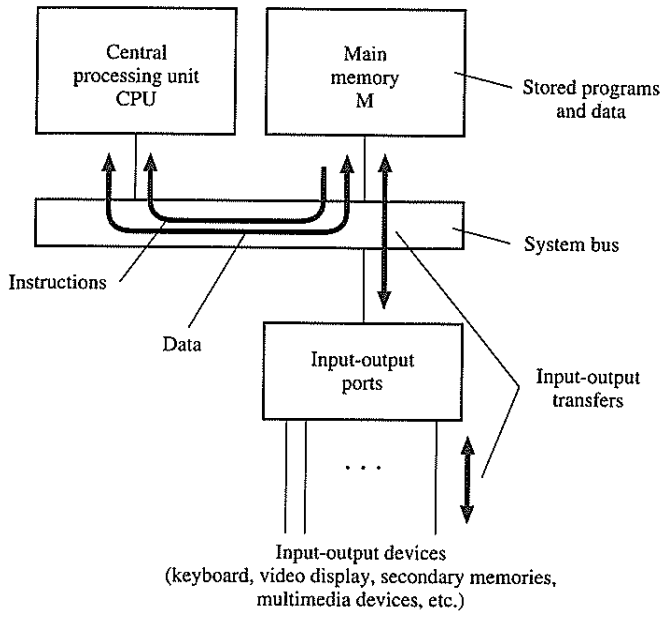


Figure 1.28
Overview of computer system operation.

Basic organization. A stand-alone computer system, which is most commonly seen as a desktop machine (a PC or workstation) intended for a single user, has the basic organization illustrated by Figure 1.28; see also Figure 1.22. This organization has changed little from that found in earlier generations, despite the massive improvements in implementation technologies that have occurred in recent years. The computer's main hardware components continue to be a CPU, a main memory, and an IO subsystem, which communicate with one another over a system bus. Its main software component is an operating system that performs most system management functions.

The key hardware element is a single-chip microprocessor, embodying a modern version of the von Neumann architecture. The microprocessor serves as the computer's CPU and is responsible for fetching, decoding, and executing instructions. Data and instructions are typically composed of 32-bit words, which constitute the basic information units processed by the computer. The CPU is characterized by an instruction set containing up to 200 or so instruction types, which perform data transfer, data processing, and program control operations that have changed little over the years. The CPU may be augmented by on-chip or off-chip *coprocessors* that implement such specialized functions as managing the graphical user interface (GUI).

The role of the computer's main or primary memory M is to store programs and data as they are being processed by the CPU. M is a random-access memory (RAM) comprising a linear store of items (usually 8-bit bytes), each of which is assigned a unique address that permits the CPU to read or change (write) its contents via load or store instructions, respectively. M is backed up by a much larger but slower secondary memory, typically implemented by hard disks employing

magnetic or optical storage technology and forming part of the IO subsystem. As in the PowerPC (Figure 1.25), an intermediate memory called a cache may also be inserted between the CPU and M. Thus we find a hierarchy of memory devices composed of the CPU's registers, the cache, the main memory, and the secondary memory. This complex structure results from the fact that the fastest memory devices are also the most costly. The memory hierarchy is intended to provide the CPU with fast access to large amounts of data at a fairly low cost.

The purpose of the IO system is to enable a user to communicate with the computer. IO devices are attached to the host computer by means of IO *ports*, whose function is to control data transfers between IO devices and main memory. Active programs communicate with IO ports in much the same way as they communicate with M. An IO device is assigned a set of memory-like addresses, which allow input and output instructions to be implemented in essentially the same way as load and store instructions, respectively. However, the CPU usually takes much longer to access a word stored in the IO system than to access a word stored in M—most IO operations are quite slow.

The traditional input and output devices are a keyboard and screen (provided by a CRT or a flat-panel display), respectively, which are convenient for handling textual information. Adding a pointing device like a mouse makes a display screen into an input device, permitting communication between the user and the computer via graphical images. Special software, such as the Windows interface found in personal computers, supports GUIs. Audio interfaces for speech generation and recognition extend the computer into a multimedia system. A major component of most IO systems is a set of secondary memory devices that provide bulk storage of programs and data. Rapid transfer of information between primary and secondary memories is often a key factor in a system's overall performance.

Microcontrollers. Their small size and low cost have made it feasible to use miniature general-purpose computers, referred to as *microcontrollers,* for tasks that previously employed either special-purpose control circuits or had no control logic at all, for example, controlling a home washing machine or the ignition system of a car. Programs stored in a read-only memory (ROM) that forms a part of the main memory tailor a microcontroller to a particular application. The microcontroller is built into, or *embedded* in, the controlled device, often in a way that is invisible to the end user. Hence an embedded microcontroller that has been programmed to handle the application in question can replace application-specific control circuits, often at substantial cost savings. Furthermore, by bringing the power of a computer to bear on relatively mundane applications, manufacturers can readily introduce many new features to improve flexibility, performance, or ease of use. As a result, most computers in operation today are microcontrollers in embedded systems.

Figure 1.29 shows one of the first applications of a microcontroller: a point-of-sale (POS) terminal that has replaced cash registers in retail stores. The microcontroller has a conventional computer organization built around a system bus to which are attached a microprocessor (the CPU), one or more ROM chips for program storage, and one or more RAM chips for data and working storage. All IO devices are also connected to the system bus using IO ports with standard interfaces. The IO devices in a typical POS terminal are a keyboard, a receipt printer, a visual display, a product-code scanner, and a credit-card reader. The latter is used

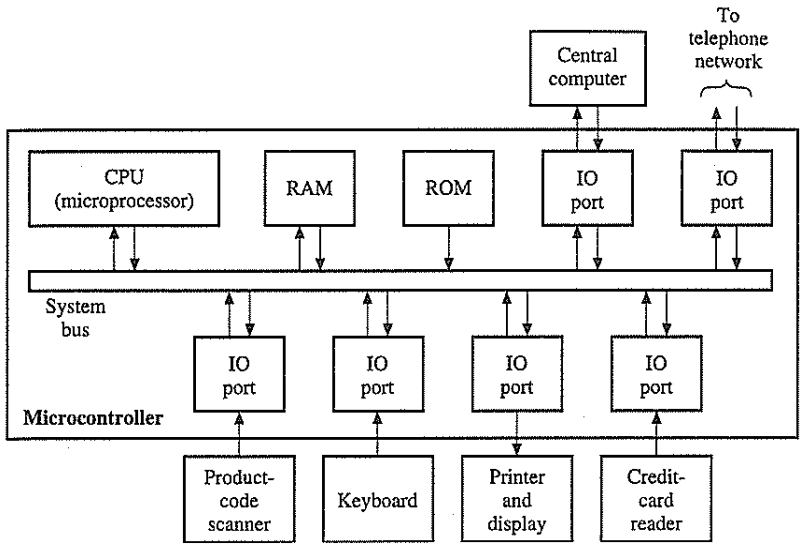


Figure 1.29
A microcontroller-based point-of-sale terminal.

for credit authorization and requires a connection to the telephone system. The final component is a link to a central computer used to provide pricing information, perform inventory control, and so forth.

Computer networks. The computer in Figure 1.29 is linked directly to a central computer and indirectly to a potentially huge number of computers via the telephone network. The linking of computers to form networks of various types has become an increasingly important feature of modern computing; see Figure 1.30. A

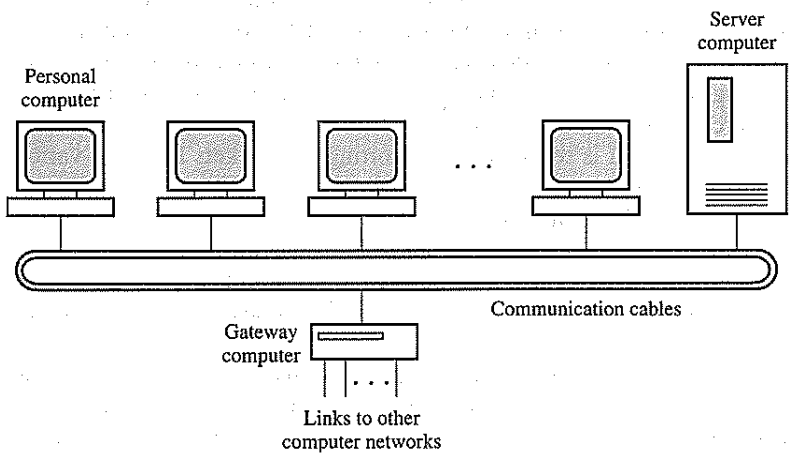


Figure 1.30
A local-area computer network.

computer in an office or industrial environment is typically linked to other computers in the same organization via communication links that can be thought of as an extension to the system bus. The linked computers then form a small, closed computer network known as a *local-area network* (LAN) or *intranet*. The physical links between the computers can be built in various ways, including electrical cables, optical fibers, and radio (wireless) links. Special IO programs (communication software) enable the computers on the network to exchange information and access common computing resources called *servers*.

Computer networks have several advantages over the large, centralized (mainframe) computers that they have come to replace. The individual user has direct access to a computer (his or her personal computer) that can quickly and conveniently handle many routine computing tasks. Users can also access computing facilities that they need less frequently, for example a high-performance supercomputer or costly IO equipment, via the computer network. Many widely dispersed users can share such specialized equipment via the network, thus lowering its cost to individual users. Furthermore, a computer network provides useful new services such as electronic mail, remote library services, and on-line shopping.

Several LANs can be linked together by various means including the telephone networks, which increasingly are designed to accommodate digital data transmission, including video data, as well as the traditional (digitized) voice communication. In Figure 1.30, one computer serves as a *gateway* device that manages communication between the LAN and other computer networks. A collection of linked LANs forms a large computer network that can be worldwide in scope. In the early 1990s a network of this sort known as the *Internet* emerged, which because of its huge size and global reach—an estimated 16 million server sites in 180 countries with 72 million users in 1997—has had a profound impact on the way people compute and communicate.

The Internet had its origins in a computer network called the ARPANET sponsored by the Advanced Research Projects Agency of the U.S. Department of Defense around 1970. This experimental network was originally designed to connect research institutions in the United States via leased lines; Figure 1.31 shows the structure of the ARPANET at an early stage in its evolution (1972) when it linked 26 research organizations in the United States. The ARPANET pioneered an information-transmission technique called *packet switching*, which divides both long and short messages into *packets of* fixed length that can be transmitted independently from source to destination via variable numbers of intermediate nodes. Each node contains a server that is responsible for sorting the packets from the various messages and forwarding them to the appropriate next destinations. Different packages can be sent by different routes determined by the network traffic conditions. At the final destination, a message is reassembled from its constituent packets. The communication software designed for the ARPANET and known as *TCP/IP* (*Transmission Control Protocol/Internet Protocol*) defines the communication standards for the Internet.

In the early years the Internet was used almost exclusively to transfer text files such as electronic mail (e-mail) messages. This situation changed fundamentally in 1989 when scientists at CERN (Centre Européen pour la Recherche Nucléaire) in Geneva overlaid on TCP/IP a new, high-level protocol called *http* (*hypertext transport protocol*) and an associated programming language *html* (*hypertext markup*

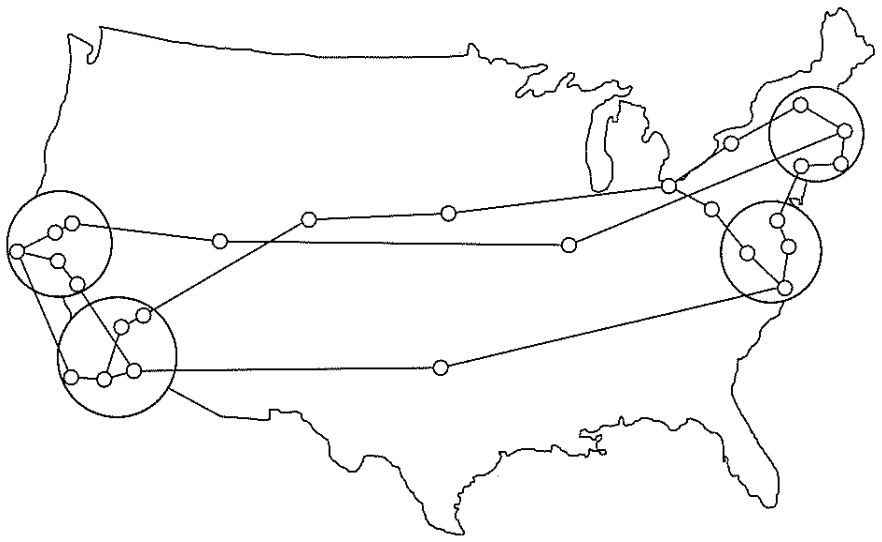


Figure 1.31
The ARPANET in 1972.

language) to permit the linking of diverse file types—text, still pictures, movies, sound, etc.—in an simple way. This combination enabled users to create multimedia files easily and transmit them rapidly over the Internet. For example, using html, a text file can be tagged with commands that tell a computer where to find and insert visual images into the text file; the required image files can be located anywhere on the Internet. The human end user can access the information from a remote host via a simple point-and-click operation on a PC or workstation. The result is an enormously rich collection of easily accessible data that has come to be known as the *World Wide Web*.

Parallel processing. So-called supercomputers capable of executing many instructions in parallel have existed since the 1950s. Early commercial supercomputers relied heavily on pipeline processing and had a single CPU organized around one or more multistage pipelines. This organization allows several instructions to be in process simultaneously in each pipeline, resulting in a potential increase in performance of a factor of n per n-stage pipeline. The Cray-1 supercomputer, first marketed by Cray Research Inc. in 1976, contained 12 pipeline processors for arithmetic-logic operations, several of which could operate in parallel [Russell 1978]. The Cray-1 could execute up to 160 million operations such as floating-point addition per second. Computers of this type have been most successfully applied to scientific computations involving large amounts of vector and matrix calculations; consequently they are sometimes called *vector processors.* The degree of parallelism n possible with a pipeline is small, typically less than 10. As the PowerPC demonstrates (Example 1.7), pipeline processing of instructions is now a standard feature of microprocessors. Indeed, single-chip microprocessors reached the Cray-1's level of performance in scientific computation in the mid-1990s.

An alternative approach to parallel processing with the potential of achieving unlimited degrees of parallelism is to use many independent processors operating in unison. For example, a network of computers can be programmed to work concurrently on different parts of the same task. Such a *loosely coupled* or *distributed* system is useful for computing tasks that can easily be partitioned into independent subtasks, with infrequent communication of results among the subtasks. However, many large-scale scientific computations permit a task to be partitioned into subtasks but require frequent and rapid exchange of results between the subtasks. The time required for such exchanges—they are essentially slow IO transfers—limits the usefulness of a computer network as a supercomputer. To address the interprocessor communication problem, computers have been built that employ n separate CPUs that are tightly coupled, both physically and logically. Processors in these machines can access one another's data rapidly and are called *multiprocessors*. The task of writing parallel programs and optimizing compilers for multiprocessors is far less well understood than the corresponding problem for a single (pipelined or nonpipelined) processor. Nevertheless, machines of this type have been studied for many years, and in the 1980s powerful multiprocessors employing many low-cost microprocessors as their CPUs began to be manufactured commercially, mainly as scientific computers.

Two types of multiprocessors are shared-memory and distributed-memory machines. In *shared-memory* machines all the processors have access to a common main memory through which they communicate to share programs and data. In *distributed-memory* machines each processor has only a private or local main memory and communicates with other processors by sending them messages through an IO subsystem linking the processors. In each case a key issue is to design processor-to-memory or processor-to-processor interconnection networks that are of high-speed and reasonable cost. For small multiprocessors containing up to 30 or so processors, a fast bus can serve as an interconnection network. In effect, the basic organization of Figure 1.30 is used with multiple CPUs attached to a high-speed system bus. To construct *massively parallel* multiprocessors, that is, computers with hundreds or thousands of CPUs, various specialized interconnection networks have been developed, which we will examine in Chapter 7. Massively parallel multiprocessors are difficult to program and cannot run conventional (uniprocessor) programs efficiently. As a result, these machines have so far had a limited impact on the commercial computer marketplace.

1.4 SUMMARY

Humans have struggled with difficult computations since ancient times. Some of these problems are inherently unsolvable—they cannot be solved even in principle by a Turing machine, which is a simple, abstract, but completely general digital computer. Some theoretically solvable problems are intractable in that they cannot be solved within a reasonable amount of time by practical computers. However, given a suitable algorithm or solution method as well as a computer of sufficient power, many important problems can be satisfactorily solved. Designing practical computers that provide the highest possible performance at acceptable cost is the basic job of the computer architect.

The design of computing machines has evolved over a long period of time. Charles Babbage conceived the concept of a general-purpose, program-controlled computer in the mid-19th century. Such a machine was not completed until the 1940s, however, when the first electronic computers were successfully constructed. Since then, progress has been dramatic, mainly driven by advances in computer hardware technology.

John von Neumann and others defined the basic organization of the modern computer. It comprises the following major components: a CPU responsible for fetching and executing instructions; a main memory used for instruction and data storage; and a set of input-output devices, such as user terminals, printers, and secondary memory devices. Three main instruction types are found in every computer: data-transfer, data-processing, and program-control instructions. The instruction set and the way the instructions are processed define the power of a computer. The computer is typically programmed in a high-level language such as C++ or Java, which is automatically compiled into executable code (object programs) built from its instruction set.

Integrated circuit technology has had a profound impact on computer design via the single-chip microprocessor and the high-capacity RAM chip. IC technology has enabled manufacturers to build very small, low-cost computers for general use (personal computers and workstations) as well as for special applications (embedded microcontrollers). IC technology has also been the driving force in the proliferation of large-scale computer networks—the Internet, for example—and high-performance multiprocessors.

As the computer industry has matured, a few computer series have tended to become de facto architectural standards, notably IBM's System/360 mainframe family introduced in the 1960s and its PC personal computer family introduced in the 1980s. Recent computer families are distinguished by powerful RISC-style instruction sets and such performance-enhancing features as pipelining, instruction-level parallelism, and cache memories. Continuing advances in hardware and software technology, such as the introduction of multimedia computing and the World Wide Web, suggest that major advances in computer design will continue into the foreseeable future.

1.5 PROBLEMS

1.1. To what extent does each of the following items play the role of processor and/or memory when used in numerical computations: an abacus; a slide rule; an electronic pocket calculator?

1.2. Consider the Turing machine program of Figure 1.4, which adds two unary numbers n_1 and n_2. A unary zero is represented by one or more blanks, which is an undesirable feature of the unary system. Determine how the given Turing machine behaves (*a*) if $n_1 = n_2 = 0$, that is, the initial tape is entirely blank; and (*b*) if $n_1 \neq 0$ but $n_2 = 0$. In each case specify the final contents of the tape.

1.3. Design a a Turing machine that subtracts a unary number n_2 from another unary number $n_1 > n_2$. Assume that n_1, n_2, and the result $n_1 - n_2$ are stored in the formats described

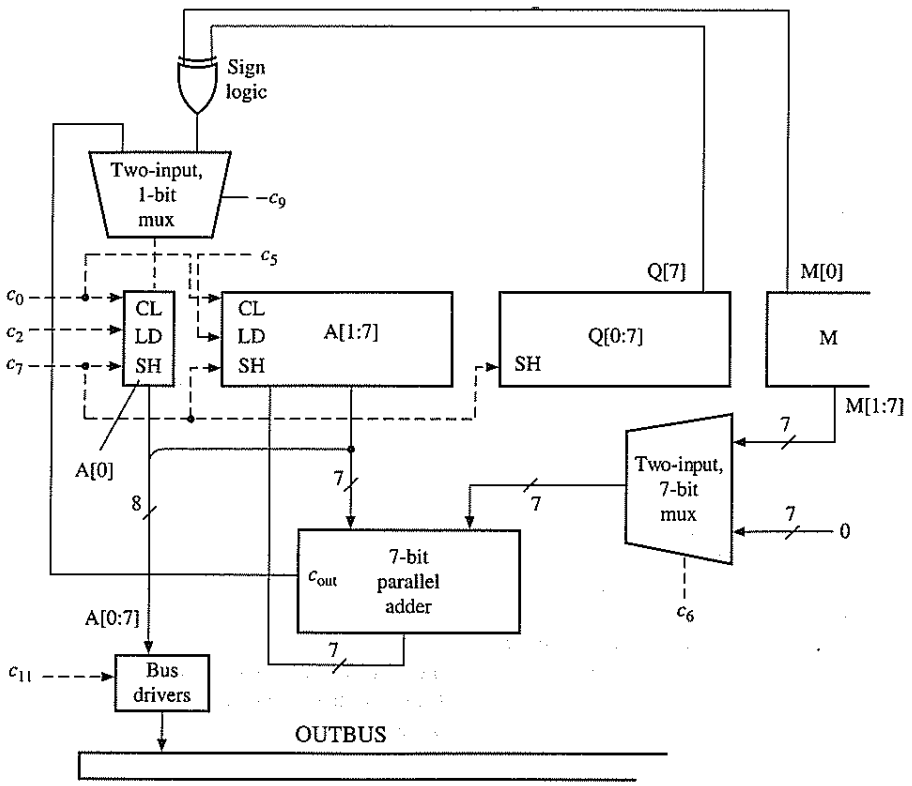


Figure 2.44
Implementation of some control points of *multiplier8*.

signals for the multiplier. In some cases several control signals implement a particular operation. For instance, the add operation employs c_6 to select the adder's right input operand, c_9 to select c_{OUT} for loading into $A[0]$, and c_2 and c_5 to actually load the 8-bit sum into $A[0:7]$. The number of distinguished control signals will vary with the details of the logic used to implement the control unit. Figure 2.44 shows a straightforward implementation of the control logic associated with the accumulator and adder subcircuits using the control signals defined in Figure 2.43.

2.3 THE PROCESSOR LEVEL

The processor or system level is the highest in the computer design hierarchy. It is concerned with the storage and processing of blocks of information such as programs and data files. The components at this level are complex, usually sequential, circuits that are based on VLSI technology. Processor-level design is very much a heuristic process, as there is little design theory at this level of abstraction.

2.3.1 Processor-Level Components

The component types recognized at the processor level fall into four main groups: processors, memories, IO devices, and interconnection networks; see Figure 2.45. In this section we give only a brief summary of the characteristics of processor-level components; they are examined individually and in much greater depth in later chapters.

Central processing unit. We define a CPU to be a general-purpose, instruction-set processor that has overall responsibility for program interpretation and execution in a computer system. The qualifier *general-purpose* distinguishes CPUs from other, more specialized processors, such as IO processors (IOPs), whose functions are restricted. An instruction-set processor is characterized by the fact that it operates on word-organized instructions and data, which the processor obtains from an external memory that also stores results computed by the processor. Most contemporary CPUs are microprocessors, implying that their physical implementation is a single VLSI chip.

Figure 2.46 shows the essential internal organization of a CPU at the register level. The CPU contains the logic needed to execute its particular instruction set and is divided into datapath and control units. The control part (the I-unit) generates the addresses of instructions and data stored in external memory. In this particular system a cache memory is interposed between the main memory M and the CPU. The cache is a fast buffer memory designed to hold an active portion of the system's address space; it is often placed, wholly or in part, on the same IC as the CPU. Each memory request generated by the CPU is first directed to the cache. If the required information is not currently assigned to the cache, the request is redirected to M and the cache is automatically updated from M. The I-unit fetches instructions from the cache or M and decodes them to derive the control signals needed for their execution. The CPU's datapath (E-unit) has the arithmetic-logic circuits that execute most instructions; it also has a set of registers for temporary data storage. The CPU manages a system bus, which is the main communication link among the CPU-cache subsystem, main memory, and the IO devices.

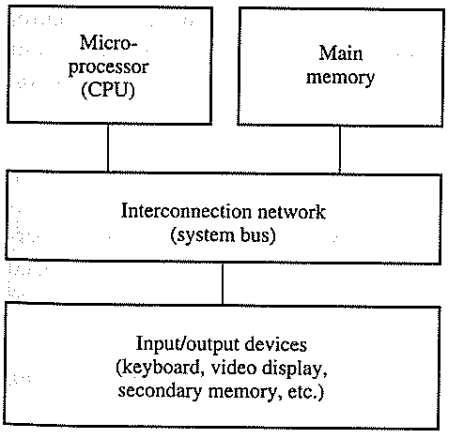


Figure 2.45
Major components of a computer system.

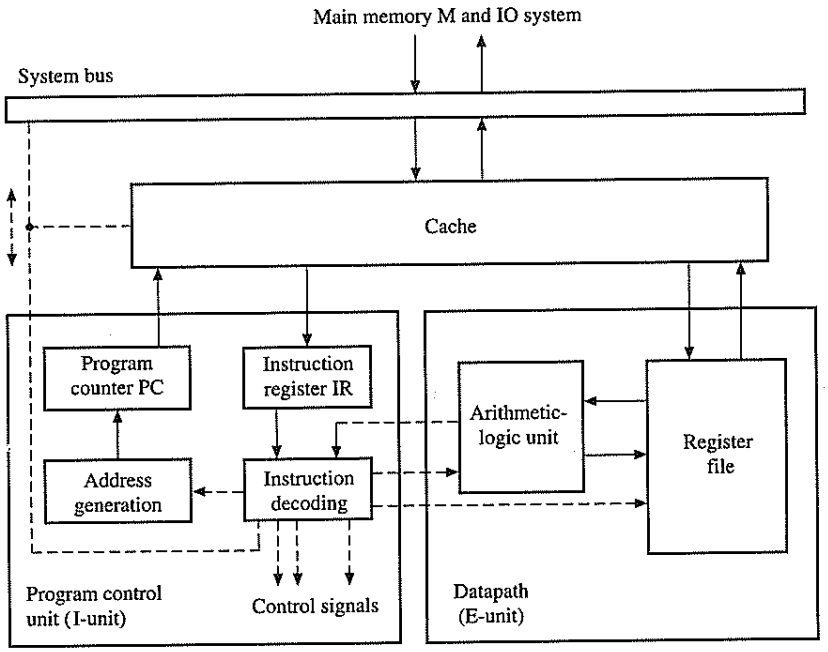


Figure 2.46
Internal organization of a CPU and cache memory.

The CPU is a synchronous sequential circuit whose clock period is the computer's basic unit of time. In one clock cycle the CPU can perform a register-transfer operation, such as fetching an instruction word from M via the system bus and loading it into the instruction register IR. This operation can be expressed formally by

$$IR := M(PC);$$

where PC is the program counter the CPU uses to hold the expected address of the next instruction word. Once in the I-unit, an instruction is decoded to determine the actions needed for its execution; for example, perform an arithmetic operation on data words stored in CPU registers. The I-unit then issues the sequence of control signals that enables execution of the instruction in question. The entire process of fetching, decoding, and executing an instruction constitutes the CPU's *instruction cycle*.

Memories. CPUs and other instruction-set processors operate in conjunction with external memories that store the programs and data required by the processors. Numerous memory technologies exist, and they vary greatly in cost and performance. The cost of a memory device generally increases rapidly with its speed of operation. The memory part of a computer can be divided into several major subsystems:

1. Main memory M, consisting of relatively fast storage ICs connected directly to, and controlled by, the CPU.

2. Secondary memory, consisting of less expensive devices that have very high storage capacity. These devices often involve mechanical motion and so are much slower than M. They are generally connected indirectly (via M) to the CPU and form part of the computer's IO system.
3. Many computers have a third type of memory called a cache, which is positioned between the CPU and main memory. The cache is intended to further reduce the average time taken by the CPU to access the memory system. Some or all of the cache may be integrated on the same IC chip as the CPU itself.

Main memory M is a word-organized addressable *random-access memory* (RAM). The term *random access* stems from the fact that the access time for every location in M is the same. Random access is contrasted with *serial access,* where memory access times vary with the location being accessed. Serial access memories are slower and less expensive than RAMs; most secondary-memory devices use some form of serial access. Because of their lower operating speeds and serial-access mode, the manner in which the stored information is organized in secondary memories is more complex than the simple word organization of main memory. Caches also use random access or an even faster memory-accessing method called associative or content addressing. Memory technologies and the organization of stored information are covered in Chapter 6.

IO devices. Input-output devices are the means by which a computer communicates with the outside world. A primary function of IO devices is to act as data transducers, that is, to convert information from one physical representation to another. Unlike processors, IO devices do not alter the information content or meaning of the data on which they act. Since data is transferred and processed within a computer system in the form of digital electrical signals, input (output) devices transform other forms of information to (from) digital electrical signals. Figure 2.47 lists some widely used IO devices and the information media they involve. Many of these devices use electromechanical technologies; hence their speed of operation is slow compared with processor and main-memory speeds. Although the CPU can take direct control of an IO device it is often under the immediate control of a special-purpose processor or control unit that directs the flow of information between the IO device and main memory. The design of IO systems is considered in Chapter 7.

Interconnection networks. Processor-level components communicate by word-oriented buses. In systems with many components, communication may be controlled by a subsystem called an *interconnection network;* terms such as *switching network, communications controller,* and *bus controller* are also used in this context. The function of the interconnection network is to establish dynamic communication paths among the components via the buses under its control. For cost reasons, these paths are usually shared. Only two communicating devices can access and use a shared bus at any time, so contention results when several system components request use of the bus. The interconnection network resolves such contention by selecting one of the requesting devices on some priority basis and connecting it to the bus. The interconnection network may place the other requesting devices in a queue.

CHAPTER 3

Processor Basics

This chapter considers the overall design of instruction-set processors as exemplified by the central processing unit (CPU) of a computer. The fundamentals of CPU organization and operation are examined, along with the selection and formats of instruction and data types. Various representative microprocessors of both the RISC and CISC types are presented and discussed.

3.1 CPU ORGANIZATION

We begin by considering the organization of the central processor (microprocessor) of a computer and the methods used to represent the information it is intended to process.

3.1.1 Fundamentals

The primary function of the CPU and other instruction-set processors is to execute sequences of instructions, that is, programs, which are stored in an external main memory. Program execution is therefore carried out as follows:

1. The CPU transfers instructions and, when necessary, their input data (operands) from main memory to registers in the CPU.
2. The CPU executes the instructions in their stored sequence except when the execution sequence is explicitly altered by a branch instruction.
3. When necessary, the CPU transfers output data (results) from the CPU registers to main memory.

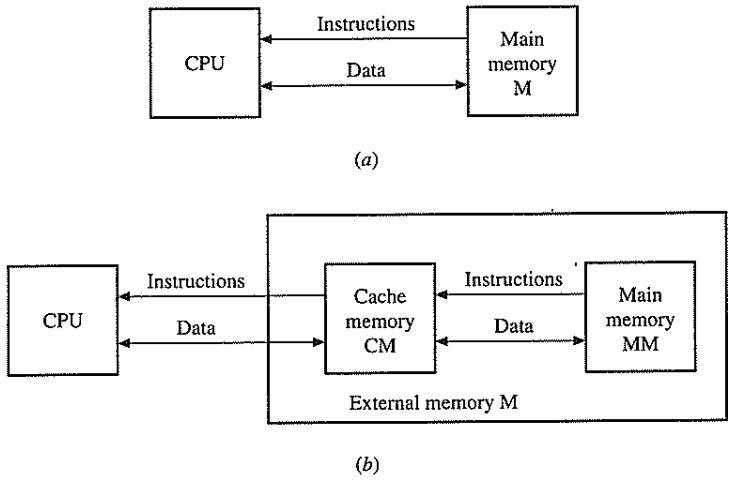


Figure 3.1
Processor-memory communication: (*a*) without a cache and (*b*) with a cache.

Consequently, streams of instructions and data flow between the external memory and the set of registers that forms the CPU's internal memory. The efficient management of these instruction and data streams is a basic function of the CPU.

External communication. If, as in Figure 3.1*a*, no cache memory is present, the CPU communicates directly with the main memory M, which is typically a high-capacity multichip random-access memory (RAM). The CPU is significantly faster than M; that is, it can read from or write to the CPU's registers perhaps 5 to 10 times faster than it can read from or write to M. VLSI technology, especially the single-chip microprocessor, has tended to increase the processor/main-memory speed disparity.

To remedy this situation, many computers have a cache memory CM positioned between the CPU and main memory. The cache CM is smaller and faster than main memory and may reside, wholly or in part, on the same chip as the CPU. It typically permits the CPU to perform a memory load or store operation in a single clock cycle, whereas a memory access that bypasses the cache and is handled by main memory takes many clock cycles. The cache is designed to be transparent to the CPU's instructions, which "see" the cache and main memory as forming a single, seamless memory space consisting of 2^m addressable storage locations M(0), M(1), ..., M(2^m–1). In this chapter we will take this viewpoint and use M to refer to the *external memory*, whether or not a cache is present. A specific memory location in M with address *adr* is referred to as M(*adr*) or simply as *adr*. When necessary, we will use MM to distinguish the main memory from the cache memory CM, as in Figure 3.1*b*. The structure of caches and their interactions with main memory are further studied in Chapter 6.

The CPU communicates with IO devices in much the same way as it communicates with external memory. The IO devices are associated with addressable registers called *IO ports* to which the CPU can store a word (an output operation) or from which it can load a word (an input operation). In some computers there are no IO

instructions per se; all IO data transfers are implemented by memory-referencing instructions, an approach called *memory-mapped IO*. This approach requires that memory locations and IO ports share the same set of addresses, so an address bit pattern that is assigned to memory cannot also be assigned to an IO port, and vice versa. Other computers employ IO instructions that are distinct from memory-referencing instructions. These instructions produce control signals to which IO ports, but not memory locations, respond. This second approach is sometimes called *IO-mapped IO*.

User and supervisor modes. The programs executed by a general-purpose computer fall into two broad groups: user programs and supervisor programs. A *user* or *application program* handles a specific application, such as word processing, of interest to the computer's users. A *supervisor program*, on the other hand, manages various routine aspects of the computer system on behalf of its users; it is typically part of the computer's operating system. Examples of supervisory functions are controlling a graphics interface and transferring data between secondary and main memory. In normal operation the CPU continually switches back and forth between user and supervisor programs. For example, while executing a user program, the need often arises for information that is available only on some hard disk unit in the computer's IO system. This condition causes the supervisor to temporarily suspend execution of the user program, execute a routine that initiates the required IO data-transfer operation, and then resume execution of the user program.

It is generally useful to design a CPU so that it can receive requests for supervisor services directly from secondary memory units and other IO devices. Such a request is called an *interrupt*. In the event of an interrupt, the CPU suspends execution of the program that it is currently executing and transfers to an appropriate interrupt-handling program. As interrupts, particularly from IO devices, require a rapid response from the CPU, it checks frequently for the presence of interrupt requests.

CPU operation. The flowchart in Figure 3.2 summarizes the main functions of a CPU. The sequence of operations performed by the CPU in processing an instruction constitutes an *instruction cycle*. While the details of the instruction cycle vary with the type of instruction, all instructions require two major steps: a *fetch step* during which a new instruction is read from the external memory M and an *execute step* during which the operations specified by the instruction are executed. A check for pending interrupt requests is also usually included in the instruction cycle, as shown in Figure 3.2.

The actions of the CPU during an instruction cycle are defined by a sequence of microoperations, each of which typically involves a register-transfer operation. The time required for the shortest well-defined CPU microoperation is the *CPU cycle time* or *clock period* T_{clock} and is a basic unit of time for measuring CPU actions. Recall that f, the CPU's clock frequency (in MHz) is related to T_{clock} (in μs) by $T_{clock} = 1/f$. As we will see, the number of CPU cycles required to process an instruction varies with the instruction type and the extent to which the processing of individual instructions can be overlapped. For the moment we will assume that each instruction is fetched from M in one CPU clock cycle (this is usually true when M is a cache) and can be executed in another CPU cycle.

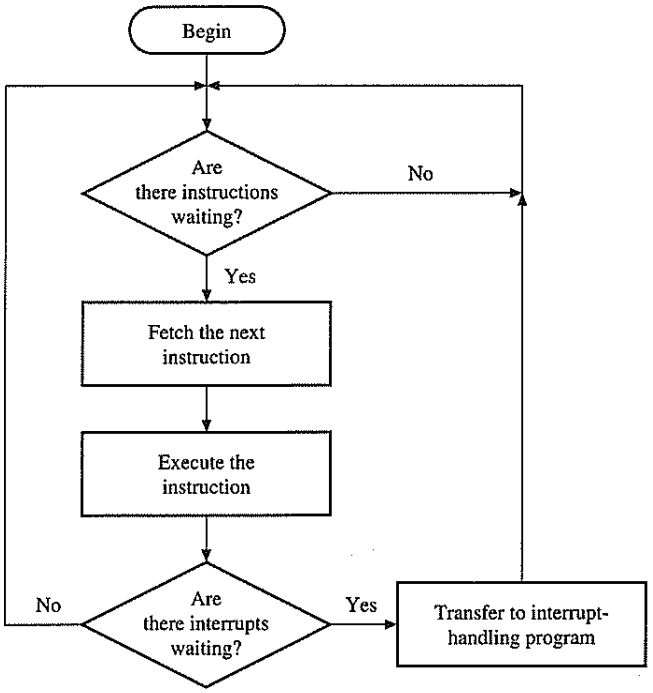


Figure 3.2
Overview of CPU behavior.

Accumulator-based CPU. Despite the improvements in IC technology over the years, CPU design continues to be based on the premise that the CPU should be as fast as the available technology and overall design requirements allow. Since cost generally increases with circuit complexity, the number of components in the CPU must be kept relatively small. The CPU organization proposed by von Neumann and his colleagues for the IAS computer (section 1.2.2) is the basis for most subsequent designs. It comprises a small set of registers and the circuits needed to execute a functionally complete set of instructions. In many early designs, one of the CPU registers, the *accumulator,*[1] played a central role, being used to store an input or output operand (result) in the execution of many instructions.

Figure 3.3 shows at the register level the essential structure of a small accumulator-oriented CPU. This organization is typical of first-generation computers (compare Figure 1.12) and low-cost microcontrollers. Assume for simplicity that instructions and data have some fixed word size n bits and that instructions can be adequately expressed by means of register-transfer operations in our HDL. Instructions are fetched by the program control unit PCU, whose main register is the pro-

[1]The term *accumulator* originally meant a device that combined the functions of number storage and addition. Any quantity transferred to an accumulator was automatically added to its previous contents. *Accumulator* is still often used in this restricted sense.

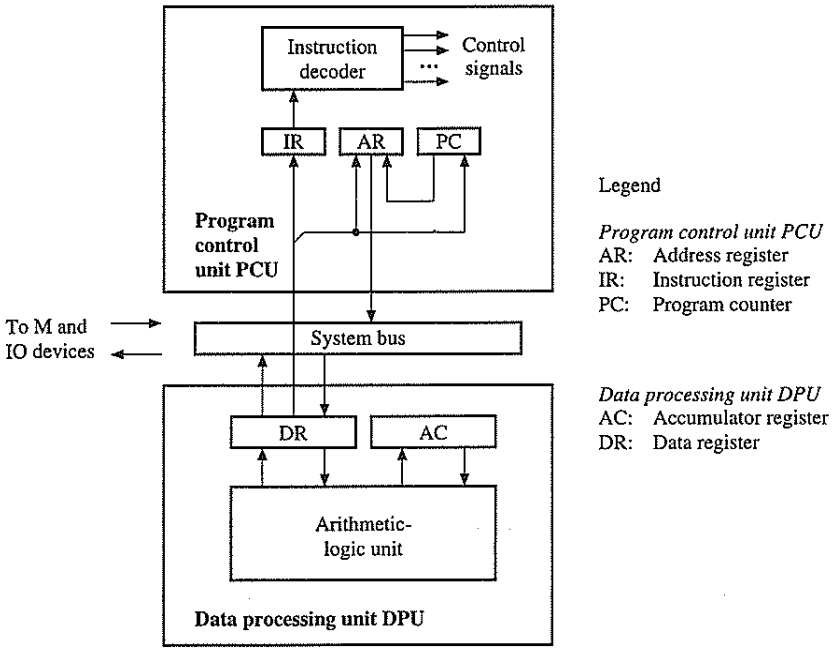


Figure 3.3
A small accumulator-based CPU.

gram counter PC. They are executed in the data processing unit DPU, which contains an n-bit arithmetic-logic unit (ALU) and two data registers AC and DR. Most instructions perform operations of the form

$$X1 := f_i(X1, X2)$$

where X1 and X2 denote a CPU register (AC, DR, or PC) or an external memory location M(*adr*). The operations f_i performed by the ALU are limited to fixed-point (integer) addition and subtraction, shifting, and logical (word-gate) operations.

Some instructions have an operand in an external memory location M(*adr*), and must therefore include the address part *adr*. Memory addresses are stored in two address registers in the PCU: the program counter PC, which stores instruction addresses only, and the general-purpose (data) address register AR. An instruction *I* that refers to a data word in M contains two parts, an opcode *op* and a memory address *adr*, and may be written as *I* = *op.adr*. Each instruction cycle begins with the instruction fetch operation

$$IR.AR := M(PC); \qquad (3.1)$$

which transfers the instruction word *I* from M to the CPU. The opcode *op* is loaded into the PCU's instruction register IR, and the address *adr* is loaded into address register AR. Hence (3.1) is equivalent to

$$IR := op, AR := adr;$$

Instructions that do not reference M do not use AR; their opcode part specifies the CPU registers to use, as well as the operation f_i to be carried out. Once it has placed the opcode of I in IR, the CPU proceeds to decode and execute it. Note that, at this point, the CPU can increment PC in order to obtain the address of the next instruction.

The two essential memory-addressing instructions are called load and store. The *load* instruction for our sample CPU is

$$AC := M(adr);$$

which transfers a word from the memory location with address *adr* to the accumulator. It is often written in assembly-language programs as LD *adr*. The corresponding *store* instruction is

$$M(adr) := AC;$$

which transfers a word from AC to M and may be written as ST *adr*. Note how the accumulator AC serves as an implicit source or destination register for data words.

Programming considerations. Data-processing operations normally require up to three operands. For example, the addition

$$Z := X + Y \tag{3.2}$$

has three distinct operands *X, Y,* and *Z*. The accumulator-based CPU of Figure 3.3 supports only *single-address* instructions, that is, instructions with one explicit memory address. However, AC and DR can serve as *implicit* operand locations so that multioperand operations can be implemented by executing several instructions in sequence. For example, a program to implement (3.2), assuming that *X, Y,* and *Z* all refer to data words in M, can take the following form:

HDL format	Assembly-language format	Narrative format (comment)
AC := M(*X*);	LD *X*	Load *X* from M into accumulator AC.
DR := AC;	MOV DR, AC	Move contents of AC to DR.
AC := M(*Y*);	LD *Y*	Load *Y* into accumulator AC.
AC := AC + DR;	ADD	Add DR to AC.
M(*Z*) := AC;	ST *Z*	Store contents of AC in M.

The preceding program fragment uses only the load and store instructions to access memory, a feature called *load/store architecture*. It is common (but as we will see, not always desirable) to allow other instructions to specify operands in memory. A CPU like that of Figure 3.3 can be designed to implement memory-referencing instructions of the form

$$AC := f_i(AC, M(adr))$$

whose execution requires two steps: one to move M(*adr*) to or from DR and one to perform the designated operation f_i. With an add instruction of this form, we can reduce the foregoing program from five to three instructions.

HDL Format	Assembly-language format	Narrative format (comment)
AC := M(*X*);	LD *X*	Load *X* from M into accumulator AC.
AC := AC + M(*Y*);	ADD *Y*	Load *Y* into DR and add to AC.
M(*Z*) := AC;	ST *Z*	Store contents of AC in M.

The memory-referencing ADD *Y* instruction can be expected to take longer to execute than the original ADD instruction that references only CPU registers. Memory references also complicate the instruction-decoding logic in the PCU. However, overall execution time should be reduced because we have eliminated an LD and a MOV instruction completely. As we will see later, the cost-performance impact of replacing a simple instruction with a more complex one has subtle implications that lie at the heart of the RISC–CISC debate.

Instruction set. Figure 3.4 gives a possible instruction set for our simple accumulator-based CPU, assuming a load/store architecture. These 10 instructions have the flavor of the instruction sets of some recent RISC machines, which demonstrate that small instruction sets can be both complete and efficient. We are, however, ignoring some important practical implementation issues in the interest of simplicity. We have not, for instance, specified the precise instruction or data formats to be used, and we do not consider such problems as numerical overflow—this condition occurs when an arithmetic instruction produces a result that is too big to fit in its destination register.

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Load	AC := M(*X*)	LD X	Load *X* from M into AC.
	Store	M(*X*) := AC	ST X	Store contents of AC in M as *X*.
	Move register	DR := AC	MOV DR, AC	Copy contents of AC to DR.
	Move register	AC := DR	MOV AC, DR	Copy contents of DR to AC.
Data processing	Add	AC := AC + DR	ADD	Add DR to AC.
	Subtract	AC := AC – DR	SUB	Subtract DR from AC.
	And	AC := AC ***and*** DR	AND	And bitwise DR to AC.
	Not	AC := ***not*** AC	NOT	Complement contents of AC
Program control	Branch	PC := M(*adr*)	BRA adr	Jump to instruction with address *adr*.
	Branch zero	**if** AC = 0 **then** PC := M(*adr*)	BZ adr	Jump to instruction *adr* if AC = 0.

Figure 3.4
Instruction set for the CPU of Figure 3.3.

The load and store instructions obviously suffice for transferring data between the CPU and main memory. We know from Boolean algebra that the AND and NOT operations are functionally complete, implying that the instruction set enables any logical operation to be programmed. We also know that addition and subtraction suffice for implementing most arithmetic operations. Consider, for example, the arithmetic operation negation, for which many CPUs have a single instruction of the type AC := –AC. We can easily implement negation by a three-instruction sequence as follows:

HDL format	Assembly-language format	Narrative format (comment)
DR := AC;	MOV DR, AC	Copy contents X of AC to DR.
AC := AC – DR;	SUB	Compute AC = $X - X = 0$.
AC := AC – DR;	SUB	Compute AC = $0 - X = -X$.

Figure 3.4 also gives a small set of program control instructions: an unconditional branch instruction BRA and a conditional branch-on-zero instruction BZ that tests the contents of AC. Observe that these instructions load a new address into the program counter PC, thus altering the instruction execution sequence. The BZ instruction allows more powerful program control operations such as procedure call and return to be implemented; it also facilitates complex operations such as multiplication, as we demonstrate in Example 3.1.

EXAMPLE 3.1 A MULTIPLICATION PROGRAM. Suppose we want to use the tiny instruction set of Figure 3.4 to program the multiplication operation

$$AC := AC \times N$$

where the multiplicand is the initial contents of the accumulator AC and the multiplier N is a variable stored in memory. We will assume that the multiplier and multiplicand are both unsigned numbers and that they are sufficiently small that the product will fit in a single word. We can construct the desired program along the following lines. We will execute the basic ADD instruction N times to implement $AC \times N$ in the form AC + AC + ... + AC. We will treat the memory location storing N as a count register and, after each addition step, decrement it by one until it reaches zero. We will test for $N = 0$ by means of the BZ instruction, and so we will have to transfer N to AC in order to perform this test. We will also have to use some memory locations as temporary registers for storing intermediate results and some other quantities, such as the initial value Y of AC. In particular, we will use memory locations *one*, *mult*, *ac*, and *prod* to store the constant 1, N, Y, and the partial product P, respectively. Here *one*, *mult*, *ac*, and *prod* are symbolic names for certain memory addresses that we have arbitrarily assigned. They are translated into numerical memory addresses by an assembler program prior to execution.

An assembly-language program implementing this plan appears in Figure 3.5. Its main body (lines 5 to 17) is traversed N times in the course of a multiplication. At the end the result P is in memory location *prod*. The first two instructions (lines 5 and 6) of the program check the value of N by reading it into AC and testing it with the BZ instruction. If the initial value of N is zero, the program exits immediately with the correct result $P = 0$. If N is nonzero, the instructions in lines 7 to 11 load it from *mult* into AC, subtract one from it, and then return the new, decremented value of N to *mult*. The

Line	Location	Instruction or data	Comment
0	*one*	00 . . . 001	The constant one.
1	*mult*	*N*	The multiplier.
2	*ac*	00 . . . 000	Location for initial value *Y* of AC.
3	*prod*	00 . . . 000	Location for (partial) product *P*.
4		ST *ac*	Save initial value *Y* of AC.
5	*loop*	LD *mult*	Load *N* into AC to test for termination.
6		BZ *exit*	Exit if $N = 0$; otherwise continue.
7		LD *one*	Load 1 into AC.
8		MOV DR, AC	Move 1 from AC to DR.
9		LD *mult*	Load *N* into AC to decrement it.
10		SUB	Subtract 1 from *N*.
11		ST *mult*	Store decremented *N*.
12		LD *ac*	Load initial value *Y* of AC.
13		MOV DR, AC	Move *Y* from AC to DR.
14		LD *prod*	Load current partial product *P*.
15		ADD	Add *Y* to *P*.
16		ST *prod*	Store the new partial product *P*.
17		BRA *loop*	Branch to *loop*.
18	*exit*	. . .	

Figure 3.5
A program for the multiplication operation AC := AC × *N*.

main step of adding *Y* to the accumulating partial product, that is, $P := P + Y$, is implemented in straightforward fashion by lines 12 to 16 of the program. Finally, a return is made to *loop* via the unconditional branch BRA (line 17).

This program uses most of the available instruction types and illustrates several weaknesses of an accumulator-based CPU. Because there are only a few data registers in the CPU, a considerable amount of time is spent shuttling the same information back and forth between the CPU and memory. Indeed, most of the instructions in this program are of the data-transfer type (ST, LD, and MOV), which do bookkeeping for the few instructions that actually compute the product *P*. It would both shorten the program and speed up its execution if we could store the quantities 1, *N*, *Y*, and *P* in their own CPU registers, as they are repeatedly required by the CPU.

Program execution. We now examine the execution process for the multiplication program of Figure 3.5. Of course, the program must be translated into executable object code prior to execution, but we can treat the assembly-language program as a symbolic representation of the object code. Recall that we are assuming that every instruction is one word long and can be fetched from M in a single CPU clock cycle. We further assume that every instruction is also executed in a single clock cycle. Hence each instruction requires two CPU clock cycles—one to fetch the instruction from M and one to execute it. At the end of

Clock cycle	Instruction cycle	PC	AR	PCU actions	DPU actions
1	ST *ac*	1004		IR.AR := M(PC), PC := PC + 1	
2			1002		M(AR) := AC
3	LD *mult*	1005		IR.AR := M(PC), PC := PC + 1	
4			1001		AC := M(AR)
5	BZ *exit*	1006		IR.AR := M(PC), PC := PC + 1	
6			1001	Test A; no further action if A ≠ 0	None
7	LD *one*	1007		IR.AR := M(PC), PC := PC + 1	
8			1000		AC := M(AR)
9	MOV DR, AC	1008		IR.AR := M(PC), PC := PC + 1	
10			*dddd*		DR := AC
11	LD *mult*	1009		IR.AR := M(PC), PC := PC + 1	
12			1001		AC := M(AR)
13	SUB	1010		IR.AR := M(PC), PC := PC + 1	
14			*dddd*		AC := AC – DR
15	ST *mult*	1011		IR.AR := M(PC), PC := PC + 1	
16			1001		M(AR) := AC
17	LD *ac*	1012		IR.AR := M(PC), PC := PC + 1	
18			1002		AC := M(AR)
19	MOV DR, AC	1013		IR.AR := M(PC), PC := PC + 1	
20			*dddd*		DR := AC
21	LD *prod*	1014		IR.AR := M(PC), PC := PC + 1	
22			1003		AC := M(AR)
23	ADD	1015		IR.AR := M(PC), PC := PC + 1	
24			*dddd*		AC := AC + DR
25	ST *prod*	1016		IR.AR := M(PC), PC := PC + 1	
26			1003		M(AR) := AC
27	BRA *loop*	1017		IR.AR := M(PC), PC := PC + 1	
28			1005	PC := AR	None
29	LD *mult*	1005		IR.AR := M(PC), PC := PC + 1	
30			1001		AC := M(AR)
31	BZ *exit*	1006		IR.AR := M(PC), PC := PC + 1	
32			1018	Test A: PC := AR if A = 0	None
33		1018		. . .	

Figure 3.6
Cycle-by-cycle execution trace of the multiplication program of Figure 3.5.

the fetch step, the PCU decodes the instruction's opcode to determine what operation to perform during the execution stage. It can also increment PC in preparation for the next instruction fetch. Recall that an edge-triggered register can be both read from and written into in the same clock cycle so that the new data is ready for use at the beginning of the next clock cycle. Hence every fetch cycle includes the following pair of register-transfer operations:

$$IR.AR := M(PC),\ PC := PC + 1 \qquad (3.3)$$

The subsequent execution cycle depends on the instruction opcode placed in IR.

Figure 3.6 depicts all the main actions taken by the CPU, including the memory addresses it generates, during execution of the program of Figure 3.5. Data of this type is referred to as an *execution trace* and is often obtained by simulation of the target CPU. (In effect, Figure 3.6 is a hand simulation of the multiplication program.) Execution traces are useful for analyzing program behavior and execution speed. In this example the program's data and instructions have been assigned to a consecutive sequence of memory locations 1000, 1001, 1002, . . . , where 1001 is the location named *one* in Figure 3.5. The first executable instruction is ST *ac*, which is in location 1004, so execution begins when PC is set to 1004. Observe how the contents of the program counter PC are incremented steadily until a branch instruction is encountered, at which point the branch address contained in the branch instruction may replace the incremented contents of PC.

3.1.2 Additional Features

Next we examine some more advanced features of CPUs and look at representative commercial microprocessors of the RISC and CISC types.

Architecture extensions. There are many ways in which the basic design of Figure 3.3 can be improved. Most recent CPUs contain the following extensions, which significantly improve their performance and ease of programming.

- *Multipurpose register set for storing data and addresses*: These replace the accumulator AC and the auxiliary registers DR and AR of our basic CPU. The resulting CPU is sometimes said to have the *general register organization* exemplified by the third-generation IBM System/360-370 (Figure 1.17), which has 32 such registers. The set of general registers is now usually referred to as a *register file*.
- *Additional data, instruction, and address types*: Most CPUs have instructions to handle data and addresses with several different word sizes and formats. Although some microprocessors have only add and subtract instructions in the arithmetic category, relatively little extra circuitry is required for (fixed-point) multiply and divide instructions, which simplify many programming tasks. Call and return instructions also simplify program design.
- *Register to indicate computation status*: A *status register* (also called a *condition code* or *flag* register) indicates infrequent or *exceptional conditions* resulting from the instruction execution. Examples are the appearance of an all-zero result or an invalid instruction like divide by zero. A status register can also indicate the user and supervisor states. Conditional branch instructions can test the status register, which simplifies the programming of conditional actions.

- *Program control stack*: Various special registers and instructions facilitate the transfer of control among programs due to procedure calling or external interrupts. Many CPUs use a flexible scheme for program-control transfer, which employs part of the external memory M as a push-down stack (see also Example 1.5). The stack memory is intended for saving key information about an interrupted program via push operations so that the saved information can be retrieved later via pop operations. A CPU address register called a *stack pointer* automatically keeps track of the stack's entry point.

Figure 3.7 shows the organization of a processor with the foregoing features. It has a register file in the DPU for data and/or address storage. The ALU obtains most of its operands from the register file and also stores most of its results there. A status register monitors the output of the ALU and other key points. The principal special-purpose address registers are the program counter and the stack pointer. Special circuits are included for address computation, although the main ALU can also be used for this purpose. The control circuits in the PCU derive their inputs from the instruction register, which stores the opcode of the current instruction, and

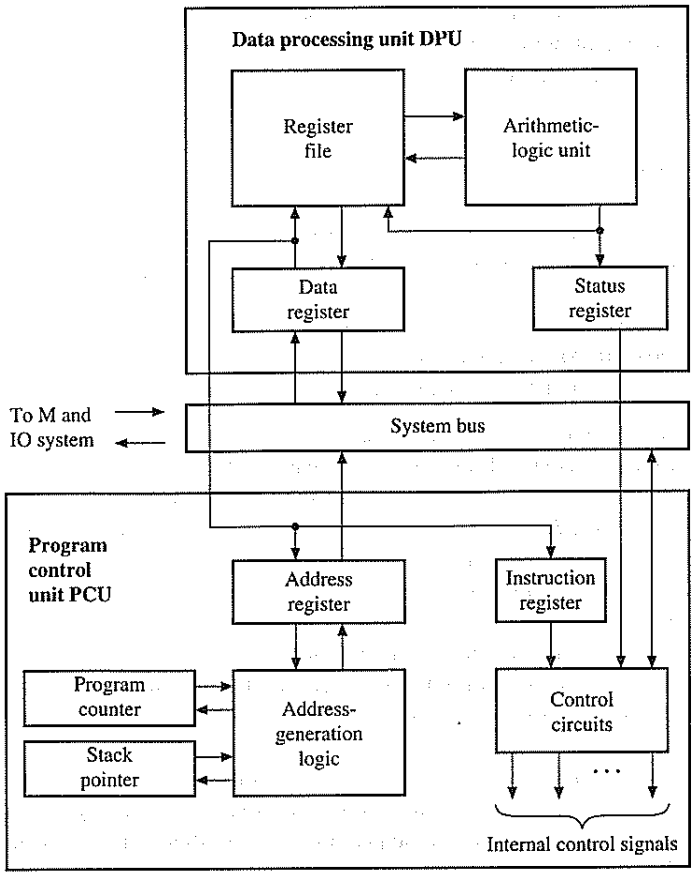


Figure 3.7
A typical CPU with the general register organization.

the status register. Communication with the outside world is via a system bus that transmits address, data, and control information among the CPU, M, and the IO system. Various nonprogrammable "buffer" registers serve as temporary storage points between the system bus and the CPU.

Pipelining. As discussed in Chapter 1, modern CPUs employ a variety of speedup techniques, including cache memories, and several forms of instruction-level parallelism. Such parallelism may be present in the internal organization of the DPU or in the overlapping of the operations carried out by the DPU and PCU. These features add to the CPU's complexity and will be explored in depth later in this book.

The considerable potential for parallel processing at the instruction level is evident even in the simple CPU of Figure 3.3. We see from the execution trace of Figure 3.6 that the main PCU and DPU activities take place in different clock cycles. If these activities do not share a resource such as the system bus, they can be carried out at the same time. In other words, while the current instruction is being executed in the DPU, the next instruction can be fetched by the PCU. For example, the three-instruction negation routine we gave earlier to change AC to –AC would be executed as follows in the style of Figure 3.6:

Clock cycle	Instruction cycle	PC	PCU actions	DPU actions
1	MOV DR, AC	2000	IR.AR := M(PC), PC := PC + 1	
2		2001		DR := AC
3	SUB	2001	IR.AR := M(PC), PC := PC + 1	
4		2002		AC := AC – DR
5	SUB	2002	IR.AR := M(PC), PC := PC + 1	
6		2003		AC := AC – DR

By merging the execution part of each instruction cycle with the fetch part of the following instruction cycle, we can reduce the overall execution time from six clock cycles to four, as shown below. (We use subscripts to distinguish the first and second SUB instructions.)

Clock cycle	Instruction cycle	PC	PCU actions	DPU actions
1	MOV	2000	IR.AR := M(PC), PC := PC + 1	
2	MOV/SUB_1	2001	IR.AR := M(PC), PC := PC + 1	DR := AC
3	SUB_1/SUB_2	2002	IR.AR := M(PC), PC := PC + 1	AC := AC – DR
4	SUB_2	2003		AC := AC – DR

This overlapping of instruction fetching and execution is an example of *instruction pipelining*, which is an important speedup feature of RISC processors. Figure 3.8 illustrates graphically the type of *two-stage* pipelining discussed above. Each instruction can be thought of as passing through two consecutive stages of

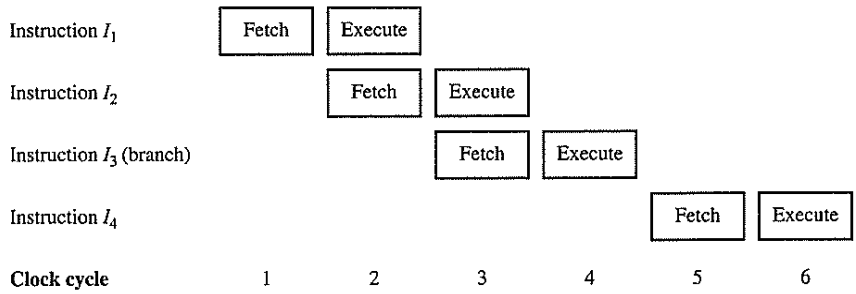


Figure 3.8
Overlapping instructions in a two-stage instruction pipeline.

processing: a fetch stage implemented mainly by the PCU and an execution stage implemented mainly by the DPU. Hence two instructions can be processed simultaneously in every CPU clock cycle, with one completing its fetch phase and the other completing its execute phase. A two-stage pipeline can therefore double the CPU's performance from one instruction every two clock cycles to one instruction every clock cycle.

A problem arises when a branch instruction is encountered, such as the BRA *loop* instruction stored in address (line) 17 of the multiplication program (Figure 3.5). Immediately before this instruction is fetched in some clock cycle i the program counter PC stores the address 17. PC is then incremented to 18 in preparation for clock cycle $i + 1$. Clearly in clock cycle $i + 1$, the CPU should *not* fetch the instruction stored at address 18—that instruction is not even in the multiplication program. In clock cycle $i + 1$, BRA is executed, which causes *loop* = 5 to be loaded into PC, implying that the next instruction should be taken from location 5. The fetching of this instruction can't begin until cycle $i + 2$, however, as illustrated in Figure 3.8 with $i = 4$. It follows that we cannot overlap the branch instruction and the instruction that follows it (I_3 and I_4 in the case of Figure 3.8).

Thus we see that branch instructions reduce the efficiency of instruction pipelining, although we will see later that steps can be taken to reduce this problem. We will also see that instruction processing is usually broken into more than two stages to increase the level of the parallelism attainable.

EXAMPLE 3.2 THE ARM6 MICROPROCESSOR [VAN SOMEREN AND ATACK 1994]. We now examine in some detail the architecture of a microprocessor family that embodies the RISC design philosophy in a relatively direct and elegant form. The ARM has its origins in the Acorn RISC Machine, a microprocessor developed in the United Kingdom in the 1980s to serve as the CPU of a personal computer. Subsequently, the family name was changed—without changing its acronym, however—to Advanced RISC Machine. The ARM family is primarily aimed at low-cost, low-power applications such as portable computers and games. For example, the Newton, a handheld "personal digital assistant" introduced by Apple Corp. in 1993 employs the ARM6 microprocessor, whose main features are described below.

The ARM6 is a 32-bit processor in that both its data words and its address words are 32 bits (4 bytes) long. It has a load/store architecture, so only its load and store instructions can address external memory M. As in most computers since the IBM System/360, main memory is organized as an array of individually addressable bytes. Thus

the maximum memory size of an ARM6 computer is 2^{32} bytes, also referred to as 4 gigabytes (4G bytes). The ARM6 employs an instruction pipeline to meet the goal of one instruction executed per CPU clock cycle. Note that it shares all these features with a more powerful (and more expensive) RISC microprocessor, the PowerPC (Example 1.7). The ARM6's instruction set is much smaller than the PowerPC's, however—it has no floating-point instructions, for example.

The internal organization of the ARM's CPU is shown in Figure 3.9. It has a 32-bit ALU and a file of 32-bit general-purpose registers. To permit direct interaction between data and control registers, the ARM has the unusual feature of placing its PC and status registers in the register file; conceptually, we will continue to view these registers as part of the PCU. There are several modes of operation, including the normal user and supervisor modes, and four special modes associated with interrupt handling. In user mode the register file appears to contain sixteen 32-bit registers designated R0:R15, where R15 is also the program counter PC, as well as a current program status register designated CPSR. (Additional registers, which we will not discuss here, are used when the CPU is in other operating modes; they are "invisible" in user mode.) The ALU is designed to perform basic arithmetic operations on 32-bit integers. It employs combinational logic for addition and subtraction and a sequential shift-and-add method similar to that described in Example 2.7 for multiplication. A combinational shift circuit is attached to the ALU to support multiplication and other operations. A separate address-incrementer circuit implements address-manipulation operations such as PC := PC + 1 independently of the ALU. Access to external memory M (a cache or main memory) is straightforward. The address of the desired location in M is placed in the PCU's address register. In the case of a store instruction, the data to be stored is also placed in the DPU's write data register. A load instruction causes a data word to be fetched from memory and placed in the read data register. Several internal buses transfer data efficiently among the DPU's registers and data processing circuits.

All ARM6 instructions are 32 bits long, and they have a variety of formats and addressing modes. There are about 25 main instruction types, which are listed in Figure 3.10. (We have omitted block move and coprocessor instructions.) This number is deceptively small, however, as instructions have options that substantially increase the number of operations they can perform. Most instructions can be applied either to 32-bit operands (words) or to 8-bit operands (bytes). Operands and addresses are usually stored in registers that can be referred to by short, 4-bit names, allowing a single ARM6 instruction to specify as many as four operands. The available address space is shared between memory and IO devices (memory-mapped IO). Consequently, the load/store instructions used for CPU-memory transfers are also used for IO operations.

Any instruction can be conditionally executed, meaning that execution may or may not occur depending on the value of designated status bits (flags) in the CPSR. The status flags are set by a previous instruction and include a negative flag N (the previous result *R* computed by the ALU was a negative number), a zero flag Z (*R* was zero), a carry flag C (*R* generated an output carry), and an overflow flag V (*R* generated a sign overflow). Hence every ARM6 instruction is effectively combined with a conditional branch instruction. The basic unconditional move instruction MOV R0, R1 can have any of 15 conditions attached to it to determine if it is to be executed (see problem 3.8). Some examples:

```
MOVCC  R0, R1      ; If flag C = 0, then R0 := R1
MOVCS  R0, R1      ; If flag C = 1, then R0 := R1
MOVHI  R0, R1      ; If flag C = 1 and flag Z = 0, then R0 := R1
```

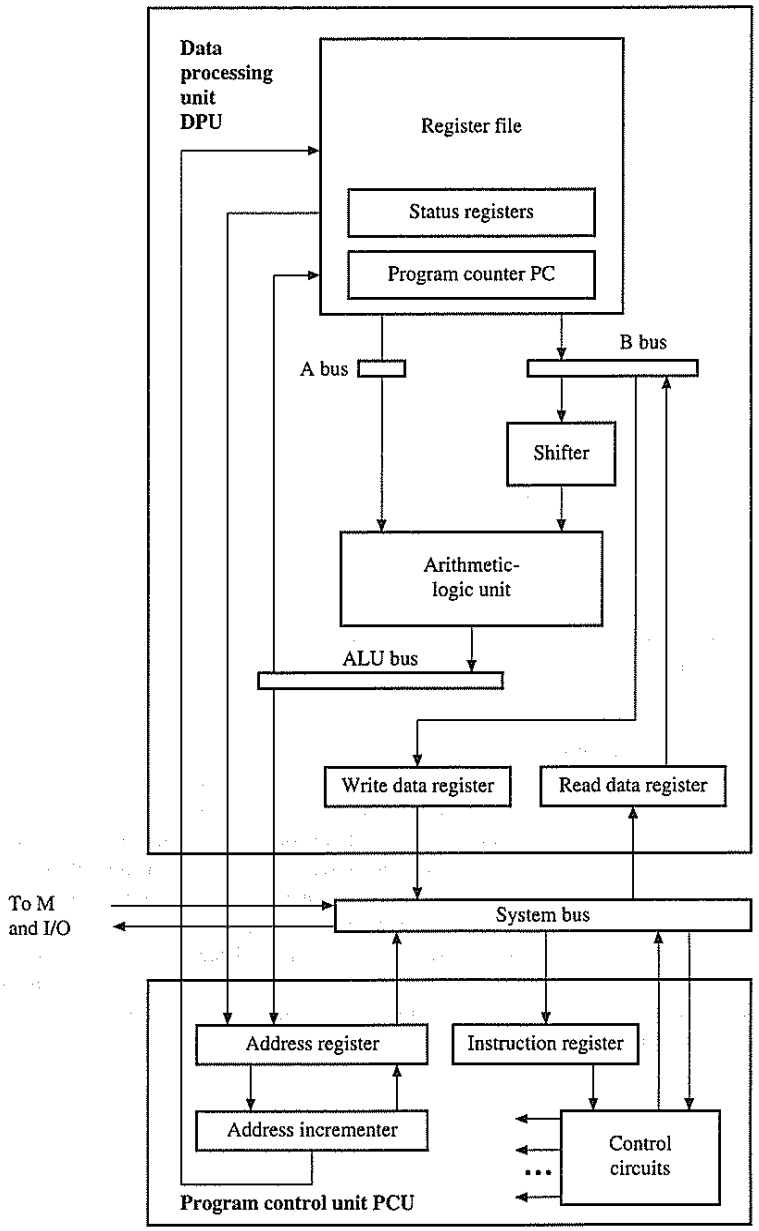


Figure 3.9
Overall organization of the ARM6.

An ARM6 instruction can also include a shift or rotation operation that is applied to one of its operands. For instance:

```
MOV     R0, R1, LSL #2          ; R0 := R1 × 4          (3.4)
```

means logically left shift (LSL) the contents of R1 by 2 bits and move the result to RO. This shift is tantamount to multiplying R1 by four before the move.

The opcode suffix S specifies whether or not an instruction affects the status flags. If S is present, appropriate flags are changed; otherwise, the flags are not affected. For example, the ARM6's move instructions affect the N, Z, and C flags, so appending S

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Move register	R3 := R9	MOV R3,R9	Copy contents of register R9 to register R3.
	Move register	R0 := 12	MOV R0,#12	Copy operand (decimal number 12) to register R0.
	Move inverted	R7 := $\overline{R0}$	MVN R7,R0	Copy bitwise inverted contents of R0 to R7
	Load	R5 := M(adr)	LDR R5, adr	Load R5 with contents of memory location adr.
	Store	M(adr) := R8	STR R8,adr	Store contents of R8 in memory location adr.
Data processing	Add	R3 := R5 + 25	ADD R3,R5,#25	Add 25 to R5; place sum in R3.
	Add with carry	R3 := R5 + R6 + C	ADC R3,R5,R6	Add R6 and carry bit C to R5; place sum in R3.
	Subtract	R3 := R5 – 9	SUB R3,R5,#9	Subtract 9 from R5; place difference in R3.
	Subtract with carry	R3 := R5 – 9 – C	SBC R3,R5,#9	Subtract 9 and borrow bit from R5; place difference in R3.
	Reverse subtract	R3 := 9 – R5	RSB R3,R5,#9	Subtract R5 from 9; place difference in R3.
	Reverse subtract with carry	R3 := 9 – R5 – C	RSC R3,R5,#9	Subtract R5 and borrow bit from 9; place difference in R3.
	Multiply	R1 := R3 × R2	MUL R1,R2,R3	Multiply R3 by R2; place result in R1.
	Multiply and add	R1 := (R3 × R2) + R4	MLA R1,R2,R3,R4	Multiply R3 by R2; add R4; place result in R1.
	And	R4 := R11 ***and*** 25_{16}	AND R4,R11,0x25	Bitwise AND R11 and 25_{16}; place result in R4.
	Or	R4 := R11 ***or*** 25_{16}	ORR R4,R11,0x25	Bitwise OR R11 and 25_{16}; place result in R4.
	Exclusive-or	R4 := R11 ***xor*** 25_{16}	EOR R4,R11,0x25	Bitwise XOR R11 and 25_{16}; place result in R4.
	Bit clear	R4 := R11 ∧ $\overline{25}_{16}$	BIC R4,R11,#25	Bitwise invert 25; AND it to R11; place result in R4.
Program control	Branch	PC := PC + *adr*	B *adr*	Jump to designated instruction.
	Branch and link	R14 := PC, PC := PC + *adr*	BL *adr*	Save old PC in "link" register R14; then jump to designated instruction.
	Software interrupt		SWI	Enter supervisor mode.
	Compare	Flags := R1 – 14	CMP r1,#14	Subtract 14 from R1 and set flags.
	Compare inverted	Flags := R1 + 14	CMN r1,#14	Add 14 to R1 and set flags.
	Logical compare	Flags := R1 ***xor*** 14	TEQ r1,#14	XOR 14 to R1 and set flags.
	Compare inverted	Flags := R1 ***or*** 14	TST r1,#14	AND 14 to R1 and set flags.

Figure 3.10
Core instruction set of the ARM6.

to, say, MOVCS, yields MOVCSS, which checks the moved data item *D*. It sets N = 1 (0) if D_{31} = 1 (0), it sets Z = 1 (0) if *D* is zero (nonzero), and it sets C to the shifter's output value.

Like other RISCs, the ARM6 has an instruction pipeline that permits the various stages of instruction processing to be overlapped. The pipeline has three stages: fetch, decode, and execute; in effect, the ARM6 breaks the first stage of the two-stage pipeline of Figure 3.8 in two. This structure permits the CPU to check every instruction's condition code in stage 2 to determine whether the instruction should be executed in stage 3. Some instructions such as multiply require more than one cycle for execution, but most require only one. Note that inclusion of an operand shift in an instruction as in (3.4) does not require an additional cycle, thanks to the fast (combinational) shifter.

A CISC machine. We turn next to a widely used CPU family, the Motorola 680X0 family, which was introduced in 1979 with the 68000 microprocessor. This example of an older CISC architecture is more streamlined and "RISC-like" than other CISCs. Later members of the family such as the 68060 [Circello et al. 1995] have speedup features such as instruction pipelining, floating-point execution units, and superscalar instruction issue. We examine an intermediate member of the series, the 68020, a 32-bit machine whose design broadly resembles that of a third-generation mainframe computer [Motorola 1989].

The 68020 is a one-chip microprocessor introduced in 1985 to serve as the CPU of a general-purpose computer such as a personal computer or workstation. Figure 3.11 outlines the organization of the 68020. It is designed to handle 32-bit words (termed *long* words in 680X0 literature) efficiently, but instructions are also provided to handle operands of 1, 8, 16, and 64 bits. As in the ARM6, memory addresses are 32 bits long, permitting a total of 2^{32} different memory locations, each storing 1 byte. Memory-mapped IO is also used in the 680X0 series. The data-processing unit has a register file containing sixteen 32-bit registers, half of which are data registers designated D0:D7 and half are address registers designated A0:A7. The ALU can execute a large set of fixed-point (but not floating-point) instructions. Instruction interpretation and other control functions of the CPU are implemented by a microprogrammed control unit.

The 68020 has about 70 distinct instruction types (or around 200 if all opcode variants are distinguished), which are summarized in Figure 3.12. A given instruction such as MOVE can be defined with several different types of operands, and the operands can be addressed in various ways. For example, the following move-register instruction written in 680X0 assembly-language format

```
MOVE.L  D1, A6                    (3.5)
```

causes the entire contents (a long word as indicated by the opcode suffix .L) of data register D1 to be copied to address register A6. In other words, (3.5) implements the register transfer A6 := D1. If .L is replaced by .B, then the resulting instruction

```
MOVE.B  D1, A6
```

causes only the byte stored in the low-order position (bits 0:7) of D1 to be copied to the corresponding part of A6.

Besides the *direct addressing* mode illustrated by the preceding example, the 68020 has several other addressing modes that give the programmer considerable

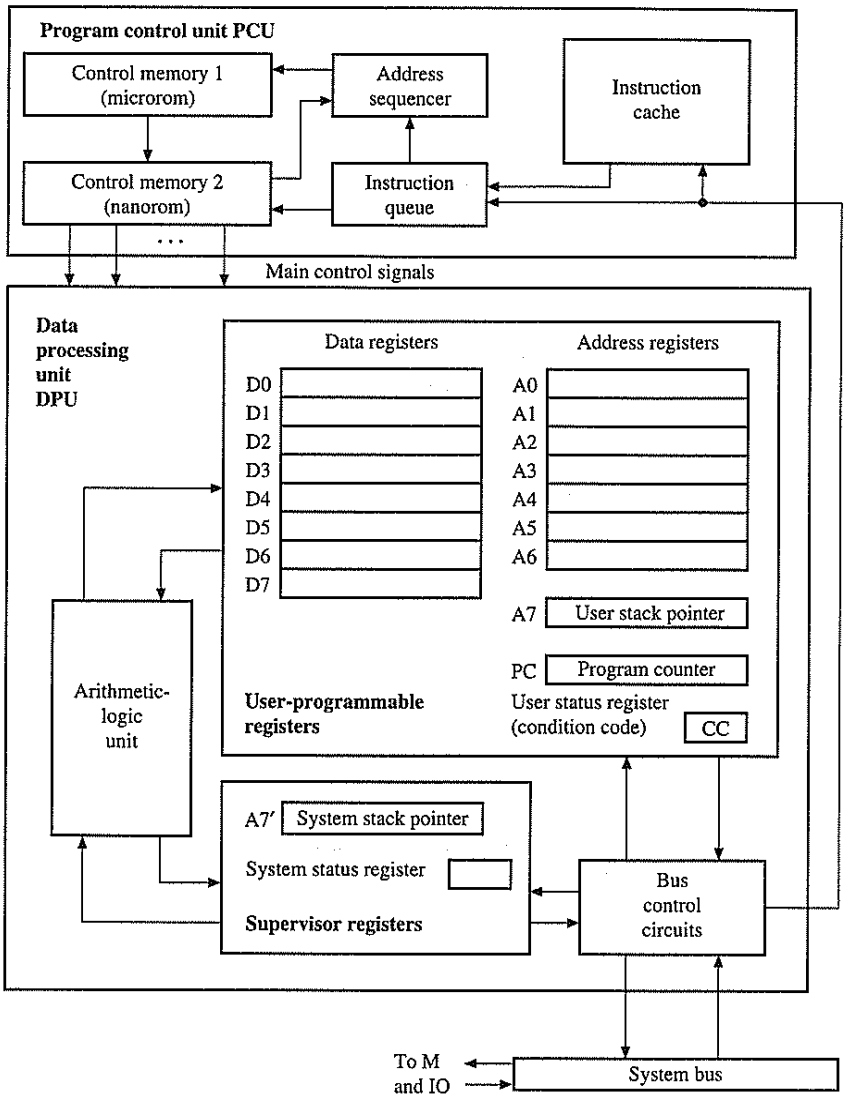


Figure 3.11
Organization of the 68020.

flexibility in accessing data. Most instructions can address memory as well as CPU registers. For example, if (3.5) is replaced by

MOVE.L D1, (A6) (3.6)

the resulting operation is M(A6) := D1, that is, a store operation with A6 serving as the memory-address register. This is an instance of *indirect addressing*. Note that while (3.5) takes 4 clock cycles to execute, (3.6) takes 12 cycles because of the time required to access external memory. The 68020's data-processing instructions can also access M directly, so the 68020 does *not* have the load/store architecture

Type	Opcode	Description
Data transfer	EXG	Exchange (swap) contents of two registers.
	MOVE	Move (copy) data unchanged from source to destination in CPU or M.
	MOVEA	Copy data to address register.
	MOVEC	Copy data to or from control register (privileged instruction).
	MOVEM	Copy multiple data items to or from specified list of registers.
	MOVEP	Copy data between register and alternate bytes of memory.
	MOVEQ	Copy "quick" (8-bit) immediate data to register.
	MOVES	Copy data using address space specified by a control register (privileged instruction).
	SWAP	Swap left and right halves of register.
Data processing	ABCD	Add decimal (BCD) numbers with carry (extend) flag.
	ADD	Add binary (twos-complement) numbers.
	ADDA	Add to address register (unsigned binary addition).
	ADDI	Add immediate binary operand.
	ADDQ	Add "quick" (3-bit) immediate binary operand.
	ADDX	Add binary with carry (extension) flag.
	AND*x*	Bitwise logical AND (*x* = I denotes immediate operand).
	AS*x*	Arithmetic left (*x* = L) or right (*x* = R) shift with extension.
	CLR	Clear operand by resetting all bits to 0.
	DIV*x*	Divide signed (*x* = S) or unsigned (*x* = U) binary numbers.
	EOR*x*	Bitwise logical EXCLUSIVE OR (*x* = I denotes immediate operand).
	EXT	Extend the sign bit of subword to fill register.
	LS*x*	Logical (simple) left (*x* = L) or right (*x* = R) shift.
	MUL*x*	Multiply signed (*x* = S) or unsigned (*x* = U) binary numbers.
	NBCD	Negate decimal number (subtract with carry from zero).
	NEG	Negate binary number (subtract from zero).
	NEGX	Negate binary number (subtract with carry from zero).
	NOT	Bitwise logical complement.
	OR*x*	Bitwise logical OR (*x* = I denotes immediate operand).
	PACK*	Convert number from unpacked to packed BCD format.
	RO*x*	Rotate (circular shift) left (*x* = L) or right (*x* = R).
	ROX*x*	Rotate left (*x* = L) or right (x = R) including the X (extend) flag.
	SBCD	Subtract decimal (BCD) numbers.
	SUB	Subtract binary (twos-complement) numbers.
	SUBA	Subtract from address register (unsigned binary subtraction).
	SUBI	Subtract immediate binary operand.
	SUBQ	Subtract "quick" (3-bit) immediate binary operand.
	SUBX	Subtract binary with borrow (extend) flag.
	UNPK*	Convert number from packed to unpacked BCD format.

Figure 3.12
Instruction set of the 68020.

characteristic of a RISC. For example:

```
ADD   (A0), D0
```

specifies the memory-to-register add operation D0 := M(A0) + D0.

EXAMPLE 3.3 680X0 PROGRAM FOR VECTOR ADDITION. Figure 3.13 gives an example of 680X0 assembly-language code that illustrates several of its basic instruction types and addressing methods. This program adds two 1000-element vectors A and B to produce a third vector C. Each vector is assumed to be a decimal

Type	Opcode	Description
Program control	B*cc*	Branch relative to PC if specified condition code *cc* is set.
	B*xxx*	Test, modify, and/or transfer (depending on *xxx*) a specified bit; set Z flag to indicate old bit value.
	BF*xxx**	Test, modify, and/or transfer (depending on *xxx*) a specified bit field; set flags to indicate old bit-field value.
	BKPT*	Execute a breakpoint trap (used for debugging).
	BRA	Branch unconditionally relative to PC.
	BSR	Call (branch to) subroutine at address relative to PC; save old PC in stack.
	CALLM*	Call subroutine (program module) saving specified control information in stack.
	CAS*x**	Compare specified operands and update register.
	CHK*x*	Check register against specified values (address bounds); trap if bounds are exceeded.
	CMP*x*	Compare two operand values; set flags based on result; *x* indicates operand type.
	DB*cc*	Loop instruction: Test condition *cc* and perform no operation if condition is met; otherwise, decrement specified register and branch to specified address.
	ILLEGAL*	Perform trap operation corresponding to an illegal opcode.
	JMP	Branch unconditionally to specified (nonrelative) address.
	JSR	Call (jump to) subroutine at specified (nonrelative) address; save old PC in stack.
	LEA	Compute effective address and load into address register.
	LINK	Allocate local data and parameter region in the stack.
	NOP	No operation (except increment PC); instruction execution continues.
	PEA	Compute effective address and push into stack.
	RTD	Return from subroutine and deallocate stack parameter region.
	RTE	Return from exception (privileged instruction).
	RTM*	Return and restore control (module state) information.
	RTR	Return and restore condition codes.
	RTS	Return from subroutine.
	S*cc*	Set operand to 1s (0s) if condition code *cc* is true (false).
	STOP	Load status register and halt (privileged instruction).
	TRAP	Begin exception processing at specified address.
	TRAP*cc*	If condition *cc* is true, then begin exception processing.
	TST	Test an operand by comparing it to zero and setting flags.
	UNLK	Deallocate local data and parameter area in the stack.
External synchronization	cp*xxx**	If condition holds, then branch with external coprocessor as specified by *xxx*.
	RESET	Reset or restart external device (privileged instruction).
	TAS	Test operand and set one of its bits to 1 using an indivisible memory-access cycle.

*Instruction not in the original 68000 instruction set.

Figure 3.12
(continued).

number composed of 1000 two-digit bytes. Each vector is stored in a fixed block of main memory whose location is known. For example, vector A is stored in memory locations 1001,1002,1003, ...,1999,2000.

The desired addition is accomplished by executing the ABCD (add using the BCD number format) instruction 1000 times. The address registers A0, A1, and A2 are used as pointers to the current 1-byte operands, and they are initialized to the required starting values using the first three MOVE instructions. These instructions use immediate addressing denoted by the prefix # to specify instruction fields that contain actual address values, while a register name such as A0 indicates that the desired operand is

Location	Instruction		Comment
	MOVE.L	#2001, A0	Load address 2001 into register A0 (pointer to vector A).
	MOVE.L	#3001, A1	Load address 3001 into register A1 (pointer to vector B).
	MOVE.L	#4001, A2	Load address 4001 into register A2 (pointer to vector C).
START	ABCD	–(A0), –(A1)	Decrement contents of A0 and A1 by 1, then add M(A0) to M(A1) using 1-byte decimal addition.
	MOVE.B	(A1), –(A2)	Decrement A2 and then store the 1-byte sum M(A1) in location M(A2) of vector C.
TEST	CMPA	#1001, A0	Compare 1001 to address in A0. If equal, set the Z flag (condition code) to 1; otherwise, reset Z to 0.
	BNE	START	Branch to START if Z is not equal to 1.

Figure 3.13
680X0 assembly-language program for vector addition.

the contents of the named register—this is direct addressing. The ABCD and MOVE.B (move byte) instructions use indirect addressing, indicated by parentheses. In this case the data specified by (A0) is the content of the memory location whose address is stored in A0, that is, the data in M(A0). Finally the minus prefix in the operand –(A0) means that A0 is decremented by one before it is used to access main memory, a mode of addressing called *autoindexing*.

The program of Figure 3.13 loads three starting addresses into the selected address registers. Since the ABCD and MOVE.B instructions begin by automatically decrementing these registers, their initial values are made one bigger than the biggest address assigned to the corresponding vector. The ABCD instruction performs the following set of operations:

$$A0 := A0 - 1,\ A1 := A1 - 1;\ M(A1) := M(A1) + M(A0);\ \text{set flags}$$

which are relatively slow because of the memory access required. The MOVE.B instruction implements the memory-to-memory move operation with autoindexing

$$A2 := A2 - 1;\ M(A2) := M(A1);\ \text{set flags}$$

The compare-address instruction CMPA checks for program termination by comparing the current address in A0 to 1001, the lowest address assigned to vector *A*. It actually subtracts its first operand (1001 in this case) from its second and sets the status flags (condition code) based on the result. Hence if A0 > 1001, then A0 – 1001 > 0 and CMPA sets the zero flag Z to 0, indicating a nonzero result. (It also sets various other flags not used by this program). When A0 finally reaches 1001, A0 – 1001 = 0, so CMPA sets Z to 1. Now the last instruction BNE, which stands for branch if not equal to zero, is a conditional branch instruction whose operation is described by

$$\textbf{if } Z \neq 1 \textbf{ then } PC := \text{START}$$

It therefore transfers execution back to the ABCD instruction in location START as long as A0 >1001. When A0 finally reaches 1001, Z becomes 1, and PC is incremented normally to exit from the program.

It is interesting to compare this 680X0 program with the similar programs given earlier for the IAS (Figure 1.15) and PowerPC (Figure 1.27) computers.

Coprocessors. The built-in instruction repertoire of the 68020 includes fixed-point multiplication and division and stack-based instructions for transferring control between programs. Hardware-implemented floating-point instructions are not available directly; however, they are provided indirectly by means of an auxiliary IC, the 68881 floating-point coprocessor. (The ARM6 also has provisions for external coprocessors.) In general, a *coprocessor P* is a specialized instruction execution unit that can be coupled to a microprocessor so that instructions to be executed by *P* can be included in programs fetched by the microprocessor. Thus the coprocessor serves as an extension to the microprocessor and forms part of the CPU as indicated in Figure 3.14.

The 68881 (and the similar but faster 68882) contains a set of eight 80-bit registers for storing floating-point numbers of various formats, including 32- and 64-bit numbers conforming to the standard IEEE 754 format (presented later). Additional control registers in the 68881 allow it to communicate with the 68020. A set of coprocessor instructions are defined for the 68020; they contain command fields specifying floating-point operations that the 68881 can execute. When the 68020 fetches and decodes such an instruction, it transfers the command portion to the coprocessor, which then executes it. Further exchanges take place between the main processor and the coprocessor until the coprocessor completes execution of its current operation, at which point the 68020 proceeds to its next instruction. The commands executed by the 68881 include the basic

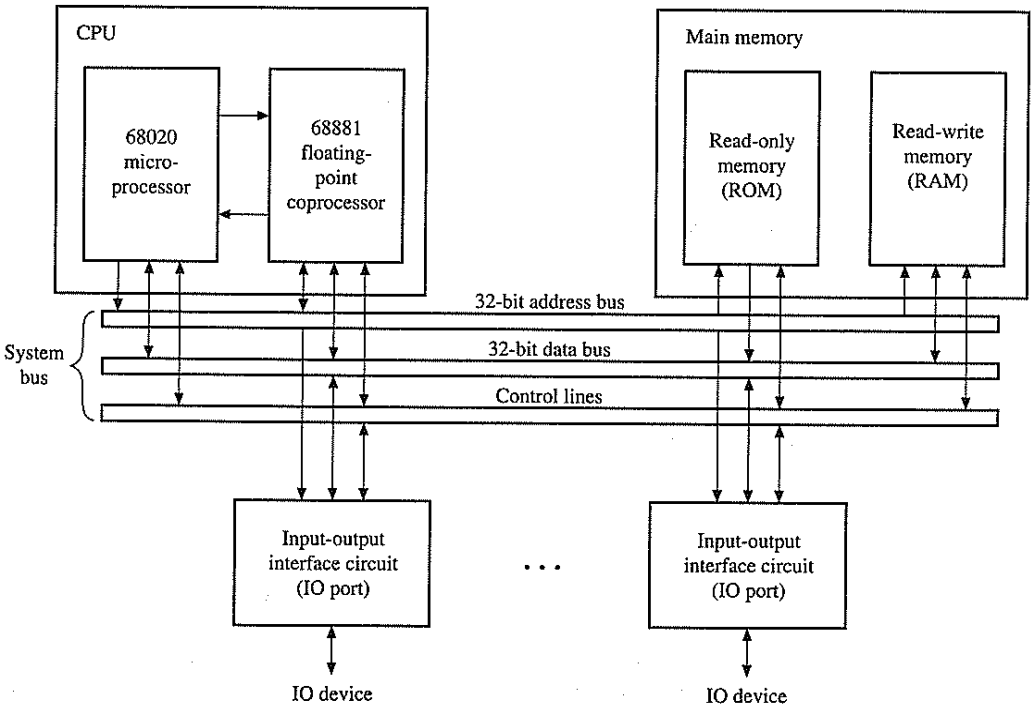


Figure 3.14
68020-based microcomputer with floating-point coprocessor.

arithmetic operations (add, subtract, multiply, and divide), square root, logarithms, and trigonometric functions. Other types of coprocessors may be attached to the 68020 in similar fashion. Later members of the 680X0 family take advantage of advances in VLSI to integrate a floating-point (co)processor into the CPU chip.

Other design features. Like the IBM System/360-370 and the ARM6, the CPU has a supervisor state intended for operating system use and a user state for application programs. As Figures 3.11 and 3.12 indicate, certain "privileged" control registers and instructions can be used only in the supervisor state. User and supervisory programs are thus clearly separated—for example, they employ different stack pointers—thereby improving system security. 680X0-based computers are also designed to allow easy implementation of *virtual memory*, whereby the operating system makes the main memory appear larger to user programs than it really is. Hardware support for virtual memory is provided by the 68851 memory management unit (MMU), another 680X0 coprocessor.

Provided they meet certain independence conditions, up to three 68020 instructions can be processed simultaneously in pipeline fashion. This pipelining is complicated by the fact that instruction lengths and execution times vary, a problem that RISCs try to eliminate. Another speedup feature found in the 68020 is a small instruction-only cache (I-cache). The 68020 prefetches instructions from main memory while the system bus is idle; the instructions can subsequently be read much more quickly from the on-chip cache than from the off-chip main memory. An unusual feature of the 68020 noted in Figure 3.11 is its use of two levels of microprogramming to implement the CPU's control logic. For the manufacturer, this feature increases design flexibility while reducing IC area compared with conventional (one-level) microprogrammed control.

3.2 DATA REPRESENTATION

The basic items of information handled by a computer are instructions and data. We now examine the methods used to represent such information, focusing on the formats for numerical data.

3.2.1 Basic Formats

Figure 3.15 shows the fundamental division of information into instructions (operation or control words) and data (operands). Data can be further subdivided into numerical and nonnumerical. In view of the importance of numerical computation, computer designs have paid a great deal of attention to the representation of numbers. Two main number formats have evolved: fixed-point and floating-point. The binary fixed-point format takes the form $b_A b_B b_C \ldots b_K$, where each b_i is 0 or 1 and a binary point is present in some fixed but implicit position. A floating-point number, on the other hand, consists of a pair of fixed-point numbers M,E, which denote the number $M \times B^E$, where B is a predetermined base. The many formats used to encode fixed-point and floating-point numbers will be examined later in

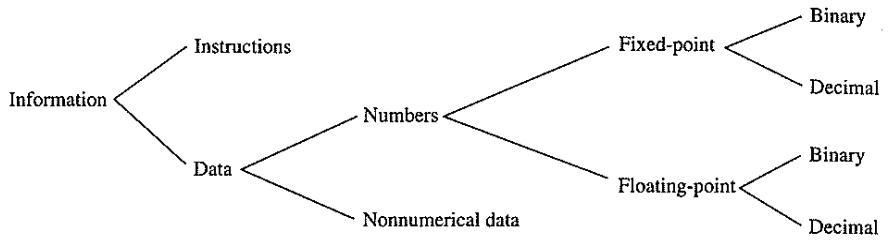


Figure 3.15
The basic information types.

the chapter. Nonnumerical data usually take the form of variable-length character strings encoded in one of several standard codes, such as ASCII (American Standards Committee on Information Exchange) code.

Word length. Information is represented in a digital computer by means of binary words, where a *word* is a unit of information of some fixed length n. An n-bit word allows up to 2^n different items to be represented. For example, with n = 4, we can encode the 10 decimal digits as follows:

$$\begin{array}{lllll} 0 = 0000 & 1 = 0001 & 2 = 0010 & 3 = 0011 & 4 = 0100 \\ 5 = 0101 & 6 = 0110 & 7 = 0111 & 8 = 1000 & 9 = 1001 \end{array} \tag{3.7}$$

To encode alphanumeric symbols or *characters,* 8-bit words called *bytes* are commonly used. As well as being able to encode all the standard keyboard symbols, a byte allows efficient representation of decimal numbers that are encoded in binary according to (3.7). A byte can store two decimal digits with no wasted space. Most computers have the 8-bit byte as the smallest addressable unit of information in their main memories. The CPU also has a standard word size for the data it processes. Word size is typically a multiple of 8, common CPU word sizes being 8, 16, 32, and 64 bits.

No single word length is suitable for representing every kind of information encountered in a typical computer. Even within a single domain such as a computer's instruction set, we often find several different word sizes. For example, instructions such as load and store that reference memory need long address fields. Instructions whose operands are all in the CPU need not contain memory addresses and so can be shorter. The precision of a number word is determined by its length; it is common therefore to have numbers of various sizes. Figure 3.16 gives a sampling of data sizes used by the Motorola 680X0. As here, the term *word* is often restricted to mean a 32-bit (4 byte) word. (680X0 literature refers to 32-bit words with the nonstandard term *long word.*) Fixed-point numbers come in lengths of 1, 2, 4, or more bytes. Floating-point numbers also come in several lengths, the shortest (single precision) number being one word (32 bits) long.

The circuits of a CPU must be carefully designed to permit various information formats to coexist smoothly. For example, if instruction length varies, as is the case in many CISC microprocessors, the program control unit must be designed to determine an instruction's length from its opcode and to fetch a variable number of instruction bytes from memory. It must also increment the program counter by a

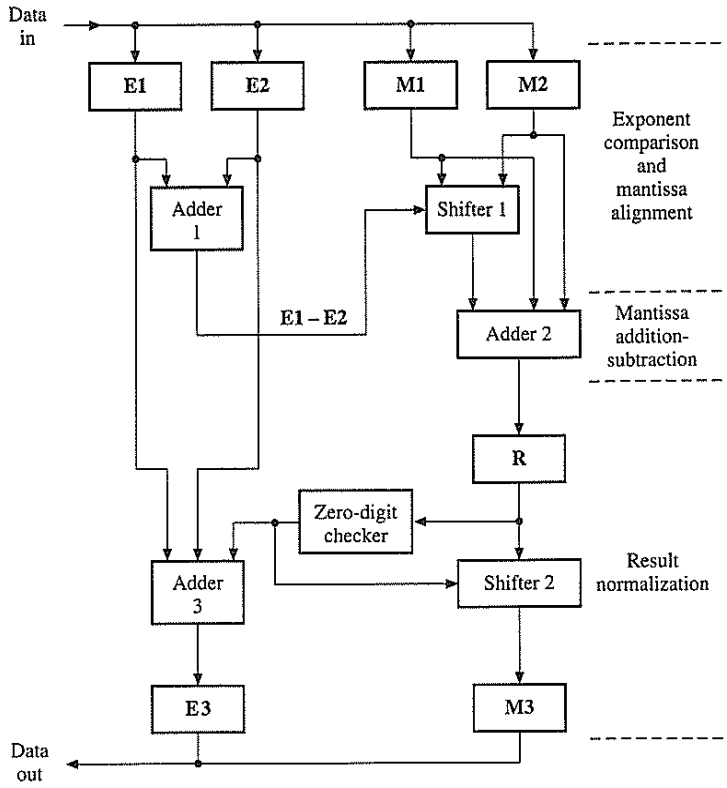


Figure 4.44
Floating-point add unit of the IBM System/360 Model 91.

Coprocessors. Complicated arithmetic operations like exponentiation and trigonometric functions are costly to implement in CPU hardware, while software implementations of these operations are slow. A design alternative is to use auxiliary processors called arithmetic coprocessors to provide fast, low-cost hardware implementations of these special functions. In general, a *coprocessor* is a separate instruction-set processor that is closely coupled to the CPU and whose instructions and registers are direct extensions of the CPU's. Instructions intended for the coprocessor are fetched by the CPU, jointly decoded by the CPU and the coprocessor, and executed by the coprocessor in a manner that is transparent to the programmer. Specialized coprocessors like this are used for tasks such as managing the memory system or controlling graphics devices. The MIPS RX000 series, for example, was designed to allow the CPU to operate with up to four coprocessors [Kane and Heinrich 1992]. One of these is a conventional floating-point processor, which is implemented on the main CPU chip in later members of the series.

Coprocessor instructions can be included in assembly or machine code just like any other CPU instructions. A coprocessor requires specialized control logic to link the CPU with the coprocessor and to handle the instructions that are executed by the coprocessor. A typical CPU-coprocessor interface is depicted in Figure 4.45. The coprocessor is attached to the CPU by several control lines that allow the

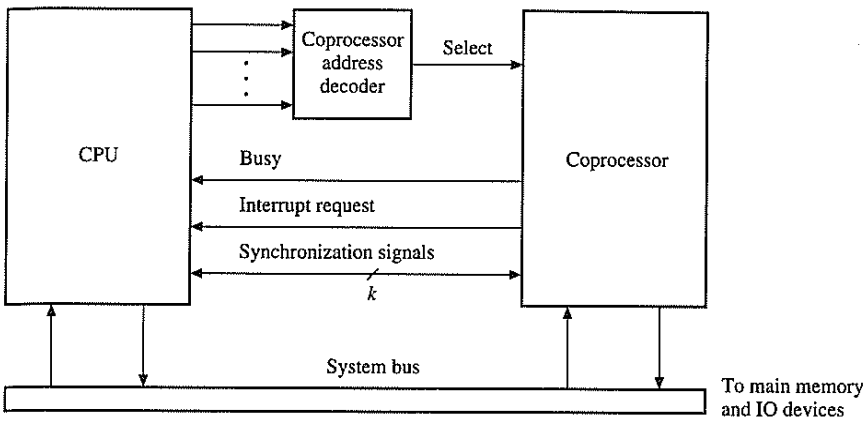


Figure 4.45
Connections between a CPU and a coprocessor.

activities of the two processors to be coordinated. To the CPU, the coprocessor is a passive or slave device whose registers can be read and written into in much the same manner as external memory. Communication between the CPU and coprocessor to initiate and terminate execution of coprocessor instructions occurs automatically as coprocessor instructions are encountered. Even if no coprocessor is actually present, coprocessor instructions can be included in CPU programs, because if the CPU knows that no coprocessor is present, it can transfer program control to a predetermined memory location where a software routine implementing the desired coprocessor instruction is stored. This type of CPU-generated interruption of normal program flow is termed a *coprocessor trap*. Thus the coprocessor approach makes it possible to provide either hardware or software support for certain instructions without altering the source or object code of the program being executed.

A coprocessor instruction typically contains the following three fields: an opcode F_0 that distinguishes coprocessor instructions from other CPU instructions, the address F_1 of the particular coprocessor to be used if several coprocessors are allowed, and finally the type F_2 of the particular operation to be executed by the coprocessor. The F_2 field can include operand addressing information. By having the coprocessor monitor the system bus, it can decode and identify a coprocessor instruction at the same time as the CPU; the coprocessor can then proceed to execute the coprocessor instruction directly. This approach is found in some early coprocessors but has the major drawback that the coprocessor, unlike the CPU, does not know the contents of the registers defining the current memory addressing modes. Consequently, it is common to have the CPU partially decode every coprocessor instruction, fetch all required operands, and transfer the opcode and operands directly to the coprocessor for execution. This is the protocol followed in 680X0-based systems employing the 68882 floating-point coprocessor, which is the topic of the next example.

EXAMPLE 4.7 THE MOTOROLA 68882 FLOATING-POINT COPROCESSOR [MOTOROLA 1989]. The Motorola 68882 coprocessor extends 680X0-series CPUs

Type	Opcode	Operation specified
Data transfer	FMOVE	Move word to/from coprocessor data or control register
	FMOVECR	Move word to/from ROM storing constants (0.0, π, e, etc.)
	FMOVEM	Move multiple words to/from coprocessor
Data processing	FADD	Add
	FCMP	Compare
	FDIV	Divide
	FMOD	Modulo remainder
	FMUL	Multiply
	FREM	Remainder (IEEE format)
	FSCALE	Scale exponent
	FSGLMUL	Single-precision multiply
	FSGLDIV	Single-precision divide
	FSUB	Subtract
	FABS	Absolute value
	FACOS	Arc cosine
	FASIN	Arc sine
	FATAN	Arc tangent
	FATANH	Hyperbolic arc tangent
	FCOS	Cosine
	FCOSH	Hyperbolic cosine
	FETOX	e to the power of x
	FETOXMI	(e to the power of x) minus 1
	FGETEXP	Extract exponent
	FGETMAN	Extract mantissa
	FINT	Extract integer part
	FINTRZ	Extract integer part rounded to zero
	FLOGN	Logarithm of x to the base e
	FLOGNP1	Logarithm of $x + 1$ to the base e
	FLOG10	Logarithm to the base 10
	FLOG2	Logarithm to the base 2
	FNEG	Negate
	FSIN	Sine
	FSINCOS	Simultaneous sine and cosine
	FSINH	Hyperbolic sine
	FSQRT	Square root
	FTAN	Tangent
	FTANH	Hyperbolic tangent
	FTENTOX	10 to the power of x
	FTWOTOX	2 to the power of x
	FLOGN	Logarithm of x to the base e
Program control	FB*cc*	Branch if condition code (status) *cc* is 1
	FDB*cc*	Test, decrement count, and branch on *cc*
	FNOP	No operation
	FRESTORE	Restore coprocessor state
	FSAVE	Save coprocessor state
	FS*cc*	Set ($cc = 1$) or reset ($cc = 0$) a specified byte
	FTST	Set coprocessor condition codes to specified values
	FTRAP*cc*	Conditional trap

Figure 4.46
Instruction set of the Motorola 68882 floating-point coprocessor.

like the 68020 (section 3.1.2) with a large set of floating-point instructions. The 68882 and the 68020 are physically coupled along the lines indicated by Figure 4.45. While decoding the instructions it fetches during program execution, the 68020 identifies coprocessor instructions by their distinctive opcodes. After identifying a coprocessor instruction, the 68020 CPU "wakes up" the 68882 by sending it certain control signals. The 68020 then transmits the opcode to a predefined location in the 68882 that serves as an instruction register. The 68882 decodes the instruction and begins its execution, which can proceed in parallel with other instructions executed within the CPU proper. When the coprocessor needs to load or store operands, it asks the CPU to carry out the necessary address calculations and data transfers.

The 68882 employs the IEEE 754 floating-point number formats described in Example 3.4 with certain multiple-precision extensions; it also supports a decimal floating-point format. From the programmer's perspective, the 68882 adds to the CPU a set of eight 80-bit floating-point data registers FP0:FP7 and several 32-bit control registers, including instruction (opcode) and status registers. Besides implementing a wide range of arithmetic operations for floating-point numbers, the 68882 has instructions for transferring data to and from its registers, and for branching on conditions it encounters during instruction execution. Figure 4.46 summarizes the 68882's instruction set. These coprocessor instructions are distinguished by the prefix F (floating-point) in their mnemonic opcodes and are used in assembly-language programs just like regular 680X0-series instructions; see Fig. 3.12. The status or condition codes *cc* generated by the 68882 when executing floating-point instructions include invalid operation, overflow, underflow, division by zero, and inexact result. Coprocessor status is recorded in a control register, which can be read by the host CPU at the end of a set of calculations, enabling the CPU to initiate the appropriate exception-processing response. As some coprocessor instructions have fairly long (multicyle) execution times, the 68882 can be interrupted in the middle of instruction execution. Its state must then be saved and subsequently restored to complete execution of the interrupted instruction.

The appearance of coprocessors stems in part from the fact that until the 1980s IC technology could not provide microprocessors of sufficient complexity to include on-chip floating-point units. Once such microprocessors became possible, arithmetic coprocessors began to migrate onto CPU chips, losing some of their separate identity in the process —especially in the case of CISC processors. For example, the 1990-vintage Motorola 68040 microprocessor integrates a 68882-style floating-point coprocessor with a 68020-style CPU in a single microprocessor chip [Edenfield et al. 1990]. Arithmetic coprocessors provide an attractive way of augmenting the performance of a RISC CPU without affecting the simplicity and efficiency of the CPU itself. The multiple function (execution) units in superscalar microprocessors like the Pentium resemble coprocessors in that each unit has an instruction set that it can execute independently of the program control unit and the other execution units.

4.3.2 Pipeline Processing

Pipelining is a general technique for increasing processor throughput without requiring large amounts of extra hardware [Kogge 1981; Stone 1993]. It is applied to the design of the complex datapath units such as multipliers and floating-point

adders. It is also used to improve the overall throughput of an instruction set processor, a topic to which we return in Chapter 5.

Introduction. A pipeline processor consists of a sequence of m data-processing circuits, called *stages* or *segments*, which collectively perform a single operation on a stream of data operands passing through them. Some processing takes place in each stage, but a final result is obtained only after an operand set has passed through the entire pipeline. As illustrated in Figure 4.47, a stage S_i contains a multiword input register or latch R_i, and a datapath circuit C_i that is usually combinational. The R_i's hold partially processed results as they move through the pipeline; they also serve as buffers that prevent neighboring stages from interfering with one another. A common clock signal causes the R_i's to change state synchronously. Each R_i receives a new set of input data D_{i-1} from the preceding stage S_{i-1} except for R_1 whose data is supplied from an external source. D_{i-1} represents the results computed by C_{i-1} during the preceding clock period. Once D_{i-1} has been loaded into R_i, C_i proceeds to use D_{i-1} to compute a new data set D_i. Thus in each clock period, every stage transfers its previous results to the next stage and computes a new set of results.

At first sight a pipeline seems a costly and slow way to implement the target operation. Its advantage is that an m-stage pipeline can simultaneously process up to m independent sets of data operands. These data sets move through the pipeline stage by stage so that when the pipeline is full, m separate operations are being executed concurrently, each in a different stage. Furthermore, a new, final result emerges from the pipeline every clock cycle. Suppose that each stage of the m-stage pipeline takes T seconds to perform its local suboperation and store its results. Then T is the pipeline's clock period. The *delay* or *latency* of the pipeline, that is, the time to complete a single operation, is therefore mT. However, the *throughput* of the pipeline, that is, the maximum number of operations completed per second is $1/T$. Equivalently, the number of clock cycles per instruction or *CPI* is one. When performing a long sequence of operations in the pipeline, its performance is determined by the delay (latency) T of a single stage, rather than by the delay mT of the entire pipeline. Hence an m-stage pipeline provides a speedup factor of m compared to a nonpipelined implementation of the same target operation.

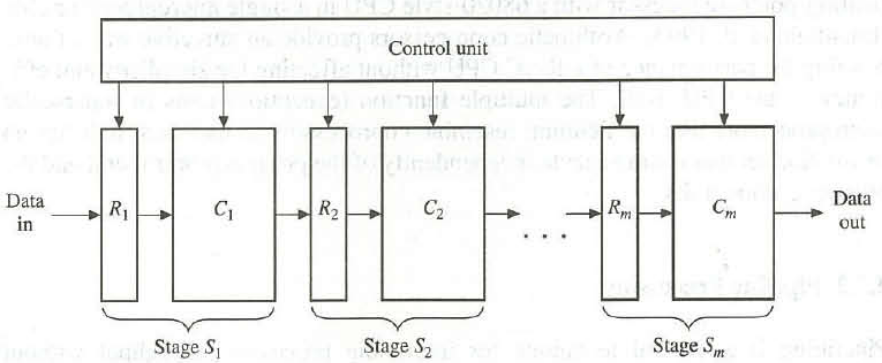


Figure 4.47
Structure of a pipeline processor.

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be realized by a pipeline processor. Consider, for example, the addition of two normalized floating-point numbers x and y, a topic discussed in section 4.3.1. This operation can be implemented by the following four-step sequence: compare the exponents, align the mantissas (equalize the exponents), add the mantissas, and normalize the result. These operations require the four-stage pipeline processor shown in Figure 4.48. Suppose that x has the normalized floating-point representation (x_M,x_E), where x_M is the mantissa and x_E is the exponent with respect to some base $B = 2^k$. In the first step of adding $x = (x_M,x_E)$ to $y = (y_M,y_E)$, which is executed by stage S_1 of the pipeline, x_E and y_E are compared, an operation performed by subtracting the exponents, which requires a fixed-point adder (see Example 4.6). S_1 identifies the smaller of the exponents, say, x_E, whose mantissa x_M can then be modified by shifting in the second stage S_2 of the pipeline to form a new mantissa x'_M that makes $(x'_M,y_E) = (x_M,x_E)$. In the third stage the mantissas x'_M and y_M, which are now properly aligned, are added. This fixed-point addition can produce an unnormalized result; hence a fourth and final step is needed to normalize the result. Normalization is done by counting the number k of leading zero digits of the mantissa (or leading ones in the negative case), shifting the mantissa k digit positions to normalize it, and making a corresponding adjustment in the exponent.

Figure 4.49 illustrates the behavior of the adder pipeline when performing a sequence of N floating-point additions of the form $x_i + y_i$ for the case $N = 6$. Add sequences of this type arise when adding two N-component real (floating-point) vectors. At any time, any of the four stages can contain a pair of partially processed scalar operands denoted (x_i,y_i) in the figure. The buffering of the stages ensures that S_i receives as inputs the results computed by stage S_{i-1} during the preceding clock period only. If T is the pipeline's clock period, then it takes time $4T$ to compute the single sum $x_i + y_i$; in other words, the pipeline's delay is $4T$. This value is approximately the time required to do one floating-point addition using a nonpipelined processor plus the delay due to the buffer registers. Once all four stages of the pipeline have been filled with data, a new sum emerges from the last stage S_4 every T seconds. Consequently, N consecutive additions can be done in time $(N + 3)T$, implying that the four-stage pipeline's speedup is

$$S(4) = \frac{4N}{N+3}$$

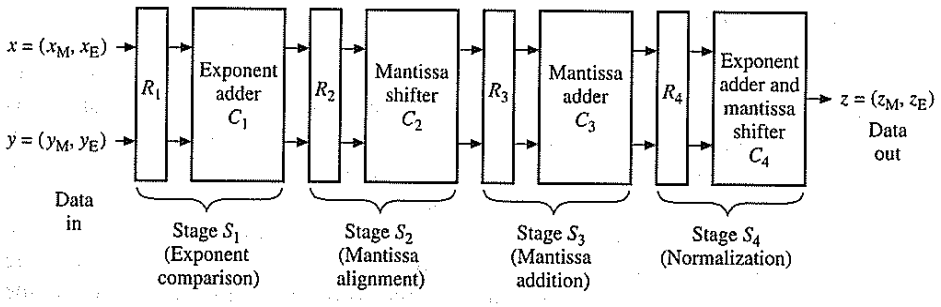


Figure 4.48
Four-stage floating-point adder pipeline.

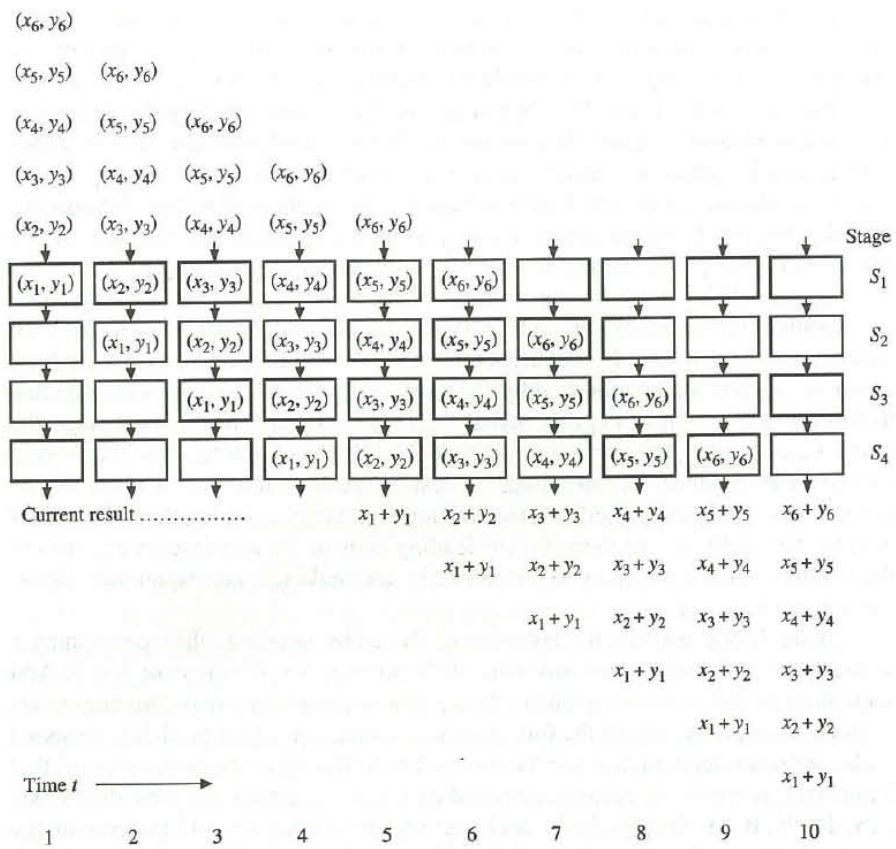


Figure 4.49
Operation of the four-stage floating-point adder pipeline.

For large N, $S(4) \approx 4$ so that results are generated at a rate about four times that of a comparable nonpipelined adder. If it is not possible to supply the pipeline with data at the maximum rate, then the performance can fall considerably, an issue to which we return in Chapter 5.

Pipeline design. Designing a pipelined circuit for a function involves first finding a suitable multistage sequential algorithm to compute the given function. This algorithm's steps, which are implemented by the pipeline's stages, should be balanced in the sense that they should all have roughly the same execution time. Fast buffer registers are placed between the stages to allow all necessary data items (partial or complete results) to be transferred from stage to stage without interfering with one another. The buffers are designed to be clocked at the maximum rate that allows data to be transferred reliably between stages.

Figure 4.50 shows the register-level design of a floating-point adder pipeline based on the nonpipelined design of Figure 4.44 and employing the four-stage organization of Figure 4.48. The main change from the nonpipelined case is the inclusion of buffer registers to define and isolate the four stages. A further modification has been made to implement fixed-point as well as floating-point addition.

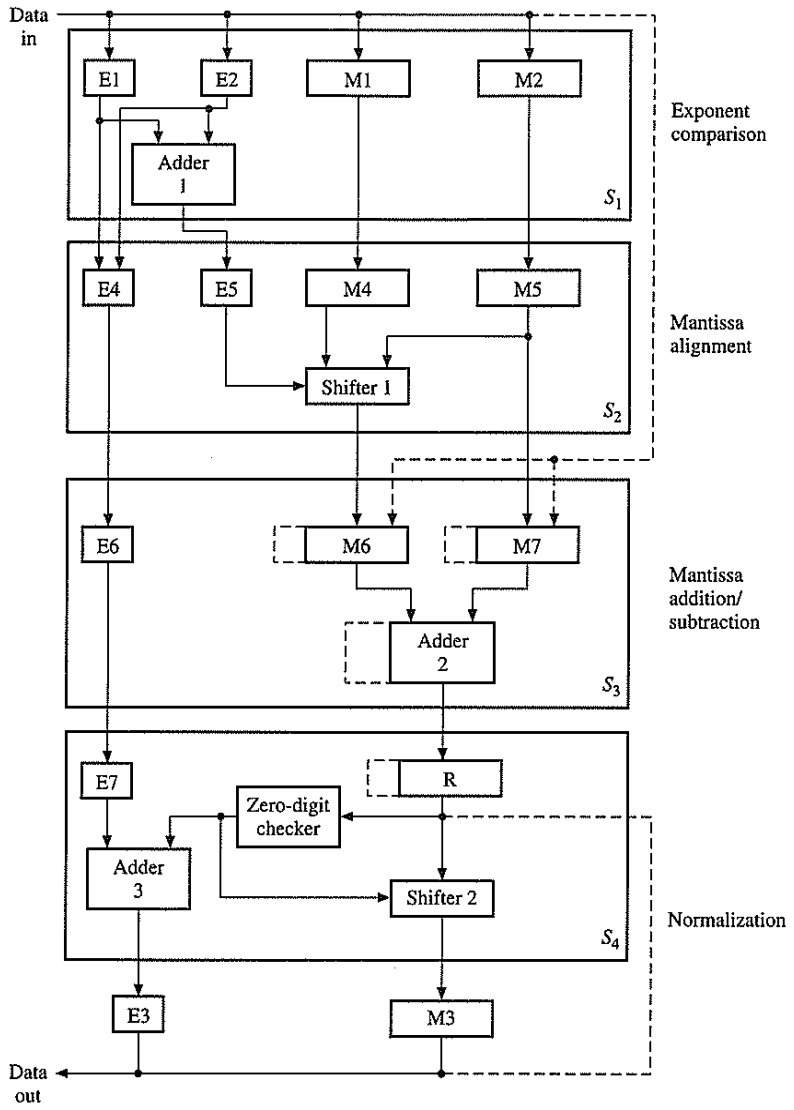


Figure 4.50
Pipelined version of the floating-point adder of Figure 4.44.

The circuits that perform the mantissa addition in stage S_3 and the corresponding buffers are enlarged, as shown by broken lines in Figure 4.50, to accommodate full-size fixed-point operands. To perform a fixed-point addition, the input operands are routed through S_3 only, bypassing the other three stages. Thus the circuit of Figure 4.50 is an example of a *multifunction pipeline* that can be configured either as a four-stage floating-point adder or as a one-stage fixed-point adder. Of course, fixed-point and floating-point subtraction can also be performed by this circuit; subtraction and addition are not usually regarded as distinct functions in this context, however.

The same function can sometimes be partitioned into suboperations in several different ways, depending on such factors as the data representation, the style of the logic design, and the need to share stages with other functions in a multifunction pipeline. A floating-point adder can have as few as two stages and as many as six. For example, five-stage adders have been built in which the normalization stage (S_4 in Figure 4.50) is split into two stages: one to count the number k of leading zeros (or ones) in an unnormalized mantissa and a second stage to perform the k shifts that normalize the mantissa.

Whether or not a particular function or set of functions F should be implemented by a pipelined or nonpipelined processor can be analyzed as follows. Suppose that F can be broken down into m independent sequential steps $F_1,F_2,\ldots,F_m$ so that it has an m-stage pipelined implementation P_m. Let F_i be realizable by a logic circuit C_i with propagation delay (execution time) T_i. Let T_R be the delay of each stage S_i due to its buffer register R_i and associated control logic. The longest T_i times create bottlenecks in the pipeline and force the faster stages to wait, doing no useful computation, until the slower stages become available. Hence the delay between the emergence of two results from P_m is the maximum value of T_i. The minimum clock period (the *pipeline period*) T_C is defined by the equation

$$T_C = \max\{T_i\} + T_R \quad \text{for } i = 1,2,\ldots,m \tag{4.44}$$

The throughput of P_m is $1/T_C = 1/(\max\{T_i\} + T_R)$. A nonpipelined implementation P_1 of F has a delay of $\Sigma_{i=1}^{m} T_i$ or, equivalently, a throughput of $1/(\Sigma_{i=1}^{m} T_i)$. We conclude the m-stage pipeline P_m has greater throughput than P_1; that is, pipelining increases performance if

$$T_C < \Sigma_{i=1}^{m} T_i$$

Equation (4.44) also implies that it is desirable for all T_i times to be approximately the same; that is, the pipeline stages should be balanced.

Feedback. The usefulness of a pipeline processor can sometimes be enhanced by including feedback paths from the stage outputs to the primary inputs of the pipeline. Feedback enables the results computed by certain stages to be used in subsequent calculations by the pipeline. We next illustrate this important concept by adding feedback to a four-stage floating-point adder pipeline like that of Figure 4.50.

EXAMPLE 4.8 SUMMATION BY A PIPELINE PROCESSOR . Consider the problem of computing the sum of N floating-point numbers $b_1,b_2,\ldots,b_N$. It can be solved by adding consecutive pairs of numbers using an adder pipeline and storing the partial sums temporarily in external registers. The summation can be done much more efficiently by modifying the adder as shown in Figure 4.51. Here a feedback path has been added to the output of the final stage S_4, allowing its results to be fed back to the first stage S_1. A register R has also been connected to the output of S_4, so that stage's results can be stored indefinitely before being fed back to S_1. The input operands of the modified pipeline are derived from four separate sources: a variable X that is typically obtained from a CPU register or a memory location; a constant source K that can apply such operands as the all-0 and all-1 words; the output of stage S_4, representing the result computed by S_4 in the preceding clock period; and, finally, an earlier result computed by the pipeline and stored in the output register R.

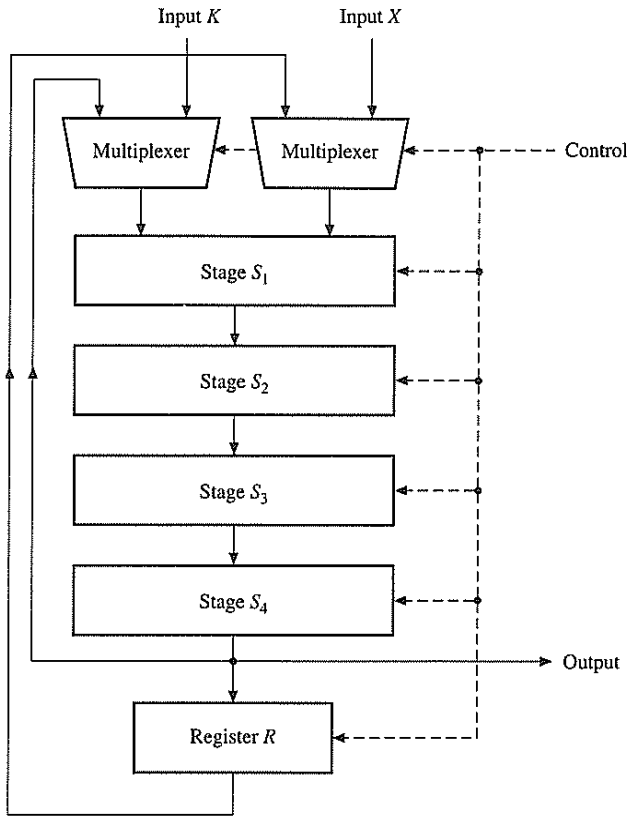


Figure 4.51
Pipelined adder with feedback paths.

The N-number summation problem is solved by the pipeline of Figure 4.51 in the following way. The external operands $b_1,b_2,\ldots,b_N$ are entered into the pipeline in a continuous stream via input X. This process requires a sequence of register or memory fetch operations, which are easily implemented if the operands are stored in contiguous register/memory locations. While the first four numbers b_1,b_2,b_3,b_4 are being entered, the all-0 word denoting the floating-point number zero is applied to the pipeline input K, as illustrated in Figure 4.52 for times $t = 1{:}4$. After four clock periods, that is, at time $t = 5$, the first sum $0 + b_1 = b_1$ emerges from S_4 and is fed back to the primary inputs of the pipeline. At this point the constant input $K = 0$ is replaced by the current result $S_4 = b_1$. The pipeline now begins to compute $b_1 + b_5$. At $t = 6$, it begins to compute $b_2 + b_6$; at $t = 7$, computation of $b_3 + b_7$ begins, and so on. When $b_1 + b_5$ emerges from the pipeline at $t = 8$, it is fed back to S_1 to be added to the latest incoming number b_9 to initiate computation of $b_1 + b_5 + b_9$. (This case does not apply to Figure 4.52, where $b_8 = b_N$ is the last item to be summed.) In the next time period, the sum $b_2 + b_6$ emerges from the pipeline and is fed back to be added to the incoming number b_{10}. Thus at any time, the pipeline is engaged in computing in its four stages four partial sums of the form

$$\begin{aligned} &b_1 + b_5 + b_9 + b_{13} + \ldots \\ &b_2 + b_6 + b_{10} + b_{14} + \ldots \\ &b_3 + b_7 + b_{11} + b_{15} + \ldots \\ &b_4 + b_8 + b_{12} + b_{16} + \ldots \end{aligned} \tag{4.45}$$

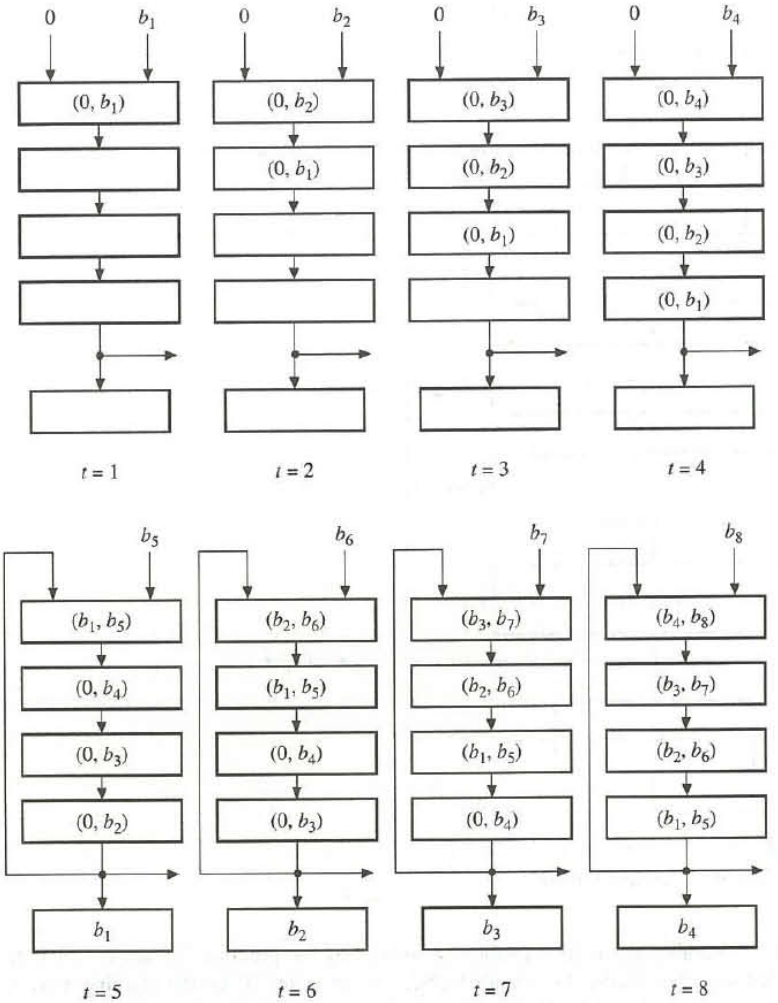


Figure 4.52
Summation of an eight-element vector.

When the last input operand b_N has entered the pipeline, the feedback structure is again altered to allow the four partial sums in (4.45) to be added together to produce the desired result $b_1 + b_2 + \ldots + b_N$. The necessary modification to the feedback structure is shown in Figure 4.52 for the case $N = 8$. At $t = 9$, the external inputs to the pipeline are disabled by setting them to zero, and the first of the four partial sums $b_1 + b_5$ at the output of stage S_4 is stored in register R. Then at $t = 10$, the new result $b_2 + b_6$ from S_4 is fed back to the pipeline inputs, along with the previous result $b_1 + b_5$ obtained from R. Thus computation of $b_1 + b_5 + b_2 + b_6$, which is the sum of half of the input operands, begins at this point. After a further delay of one time period, computation of the other half-sum $b_3 + b_7 + b_4 + b_8$ begins. When $b_1 + b_5 + b_2 + b_6$ emerges from S_4 at $t = 14$, it is stored in R until $b_4 + b_8 + b_3 + b_7$ emerges from S_4 at $t = 16$. At this point the outputs of S_4 and R are fed back to S_1. The final result is produced four time periods later—at $t = 20$ in the case of $N = 8$.

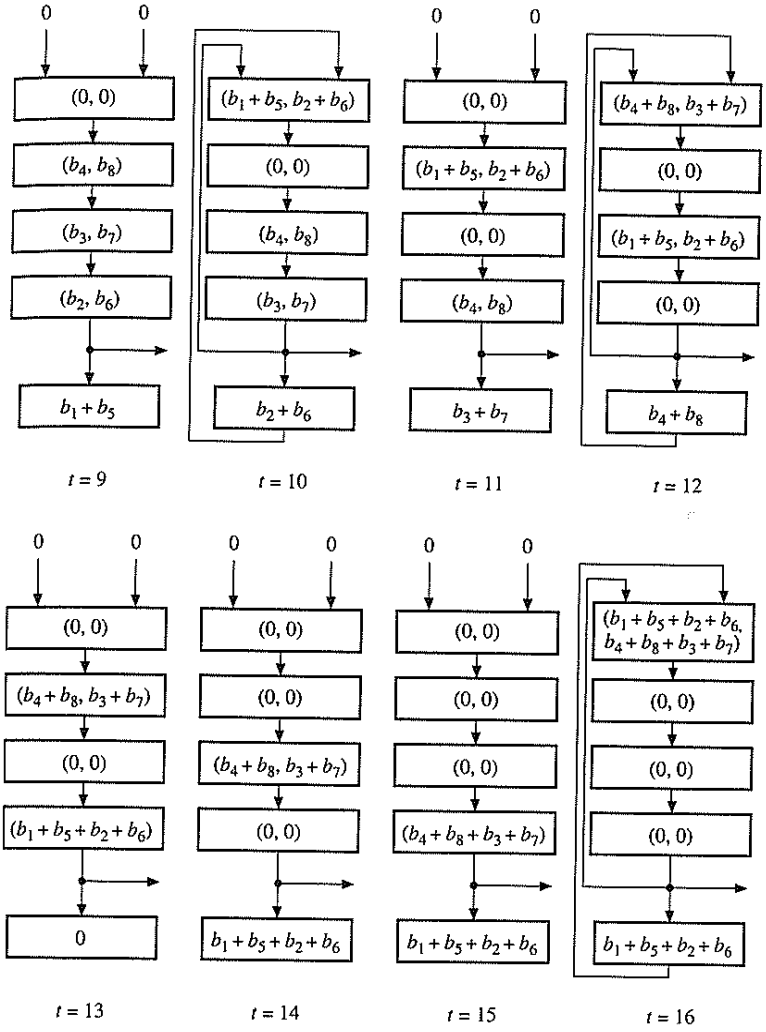


Figure 4.52
(continued)

It is easily seen that for the general case of N operands, the scheme of Figure 4.52 can compute the sum of $N > 4$ floating-point numbers in time $(N + 11)T$, where T is the pipeline's clock period, that is, the delay per stage. Since a comparable nonpipelined adder requires time $4NT$ to compute SUM, we obtain a speedup here of about $4N/(N + 11)$, which approaches 4 as N increases.

The foregoing summation operation can be invoked by a single *vector instruction* of a type that characterized the vector-processing, pipeline-based "supercomputers" of the 1970s and 1980s [Stone 1993]. For instance, Control Data Corp.'s STAR-100 computer [Hintz and Tate 1972] has an instruction SUM that computes

the sum of the elements of a specified floating-point vector $B = (b_1, b_2,\ldots,b_N)$ of arbitrary length N and places the result in a CPU register. The starting (base) address of B, which corresponds to a block of main memory, the name C of the result register, and the vector length N are all specified by operand fields of SUM. We can see from Figure 4.52 that a relatively complex pipeline control sequence is needed to implement a vector instruction of this sort. This complexity contributes significantly to both the size and cost of vector-oriented computers. Moreover, to achieve maximum speedup, the input data must be stored in a way that allows the vector elements to enter the pipeline at the maximum possible rate—generally one number-pair per clock cycle.

The more complex arithmetic operations in CPU instruction sets, including most floating-point operations, can be implemented efficiently in pipelines. Fixed-point addition and subtraction are too simple to be partitioned into suboperations suitable for pipelining. As we see next, fixed-point multiplication is well suited to pipelined design.

Pipelined multipliers. Consider the task of multiplying two n-bit fixed-point binary numbers $X = x_{n-1}x_{n-2}\ldots x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_0$. Combinational array multipliers of the kind described in section 4.1.2 are easily converted to pipelines by the addition of buffer registers. Figure 4.53 shows a pipelined array multiplier that employs the 1-bit multiply-and-add cell M of Figure 4.19 and has $n = 3$. Each cell M computes a 1-bit product x_iy_j and adds it to both a product bit from the preceding stage and a carry bit generated by the cell on its right. Thus the n cells in each stage S_i, $0 \le i \le n-1$, compute a partial product of the form

$$P_i = P_{i-1} + x_i 2^i Y \tag{4.46}$$

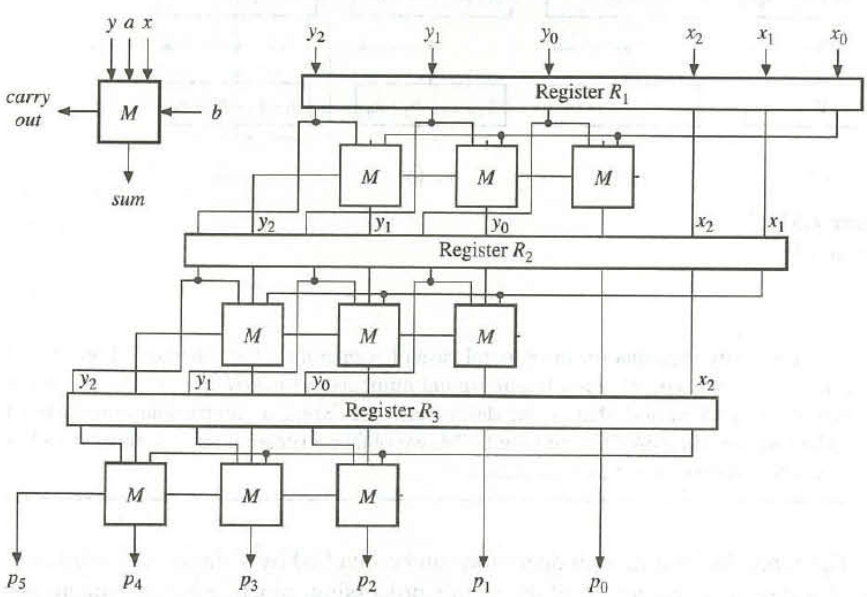


Figure 4.53
Multiplier pipeline using ripple-carry propagation.

with the final product $P_{n-1} = XY$ being computed by the last stage. In addition to storing the partial products in the buffer registers denoted R_i, the multiplicand Y and all hitherto unused multiplier bits must also be stored in R_i.

An n-stage multiplier pipeline of this type can overlap the computation of n separate products, as required, for example, when multiplying fixed-point vectors, and can generate a new result every clock cycle. Its main disadvantage is the relatively slow speed of the carry-propagation logic in each stage. The number of M cells needed is n^2, and the capacity of all the buffer registers is approximately $3n^2$ (see problem 4.31); hence this type of multiplier is also fairly costly in hardware. For these reasons, it is rarely used.

Multipliers often employ a technique called *carry-save addition,* which is particularly well suited to pipelining. An n-bit carry-save adder consists of n disjoint full adders. Its input is three n-bit numbers to be added, while the output consists of the n sum bits forming a word S and the n carry bits forming a word C. Unlike the adders discussed so far, there is no carry propagation within the individual adders. The outputs S and C can be fed into another n-bit carry-save adder where, as shown in Figure 4.54, they can be added to a third n-bit number W. Observe that the carry connections are shifted to the left to correspond to normal carry propagation. In general, m numbers can be added by a treelike network of carry-save adders to produce a result in the form (S,C). To obtain the final sum, S and C must be added by a conventional adder with carry propagation.

Multiplication can be performed using a multistage carry-save adder circuit of the type shown in Figure 4.55; this circuit is called a *Wallace tree* after its inventor [Wallace 1964]. The inputs to the adder tree are n terms of the form $M_i = x_i Y 2^k$. Here M_i represents the multiplicand Y multiplied by the ith multiplier bit weighted by the appropriate power of 2. Suppose that M_i is $2n$ bits long and that a full double-length product is required. The desired product P is $\Sigma_{i=0}^{n-1} M_i$. This sum is computed by the carry-save adder tree, which produces a $2n$-bit sum and a

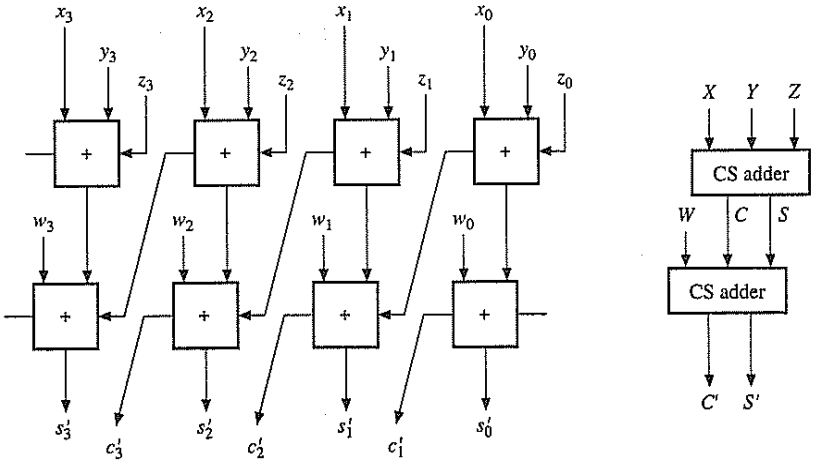


Figure 4.54
A two-stage carry-save adder.

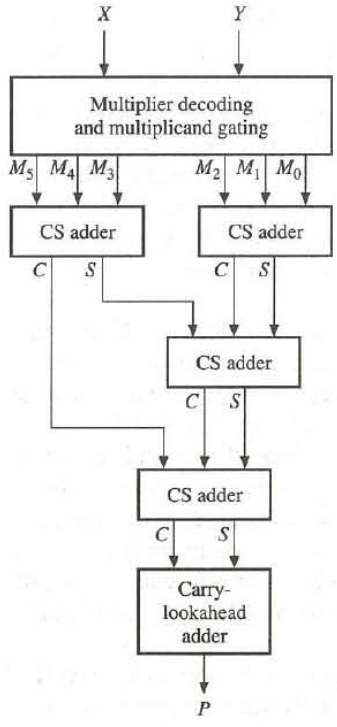


Figure 4.55
A carry-save (Wallace tree) multiplier.

$2n$-bit carry word. The final carry assimilation is performed by a fast adder—a carry-lookahead adder, for instance—with normal internal carry propagation.

The strictly combinational multiplier of Figure 4.55 is practical for moderate values of n, depending on the level of circuit integration used. For large n, the number of carry-save adders required can be excessive. Carry-save techniques can still be used, however, if the multiplier is partitioned into k m-bit segments. Only m terms M_i are generated and added via the carry-save adder circuits. The process is repeated k times, and the resulting sums are accumulated. The product is therefore obtained after k iterations.

Carry-save multiplication is well suited to pipelined implementation. Figure 4.56 shows a four-stage pipelined version of the carry-save multiplier of Figure 4.55. The first stage decodes the multiplier and transfers appropriately shifted copies of the multiplicand into the carry-save adders. The output of the first stage is a set of numbers (partial products) that are then summed by the carry-save adder tree. The carry-save logic has been subdivided into two stages by the insertion of buffer registers (denoted R in the figure). The fourth and final stage contains a carry-lookahead adder to assimilate the carries. This type of multiplier is easily modified to handle floating-point numbers. The input mantissas are processed in a fixed-point multiplier pipeline. The exponents are combined by a separate fixed-point adder, and a normalization circuit is also introduced.

The next example describes the pipelined floating-point unit of the Motorola 68040 microprocessor, which integrates the functions of the 68020 microprocessor (section 3.1.2 and Examples 3.3, 3.6, and 3.8) and its 68882 floating-point coprocessor (Example 4.7) in a single IC containing more than 1.2 million transistors.

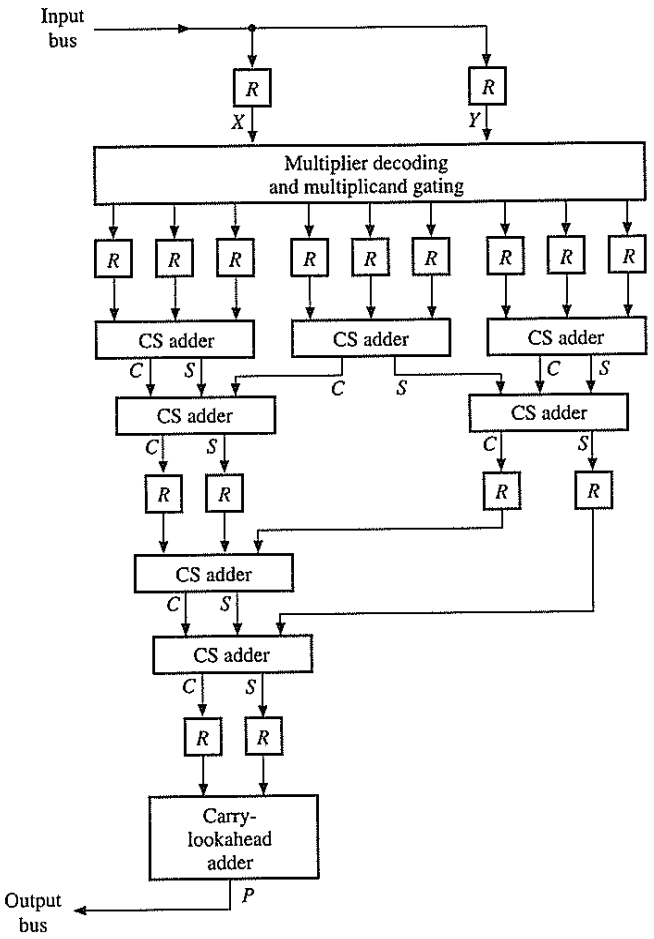


Figure 4.56
A pipelined carry-save multiplier.

This floating-point unit also implements many of the design techniques covered in this section.

EXAMPLE 4.9 THE PIPELINED FLOATING-POINT UNIT OF THE MOTOROLA 68040 [EDENFIELD ET AL. 1990]. This member of the 680X0 series of one-chip 32-bit microprocessors was introduced in 1990. It executes the combined instruction sets of the 68020 CPU and the 68882, which are listed in Figures 3.12 and 4.46, respectively, and is about four times as fast as the 68020 for a fixed clock rate. The 68040 contains two pipelined arithmetic processors: an integer unit (IU), which handles integer instructions, logical instructions, and address calculations, and a floating-point unit (FPU), which we now examine. A key design goal of the FPU is compatibility with object code written for the 68020 and 68882, as well as compatibility with the IEEE 754 floating-point standard. Only the subset of the 68882's instructions listed in Figure 4.57, including the four basic arithmetic operations and square root, are actually realized in hardware. Also included is a small set of data-transfer and program-control

Type	Opcode	Operation specified
Data transfer	FMOVE	Move word to/from coprocessor data or control register
	FMOVEM	Move multiple words to/from coprocessor
Data processing	FADD	Add
	FCMP	Compare
	FDIV	Divide
	FMUL	Multiply
	FSUB	Subtract
	FABS	Absolute value
	FNEG	Negate
	FSQRT	Square root
Program control	FB*cc*	Branch if condition code (status) *cc* is 1
	FDB*cc*	Test, decrement count, and branch on *cc*
	FRESTORE	Restore coprocessor state
	FSAVE	Save coprocessor state
	FS*cc*	Set (*cc* = 1) or reset (*cc* = 0) a specified byte
	FTST	Set coprocessor condition codes to specified values
	FTRAP*cc*	Conditional trap

Figure 4.57
Subset of the 68882 floating-point instruction set implemented by the 68040.

instructions to support floating-point operations. The remaining 68882 instructions must be simulated by software, for which the 68040 provides some hardware support.

The 68040's FPU has the three-stage pipeline organization shown in Figure 4.58. It is designed to handle floating-point number sizes of 32, 64, and 80 bits. The FPU is divided into two largely independent subunits: one for 64-bit mantissas (which expand to 67 bits when guard digits are included) and the other for 16-bit exponents. The FPU obtains its operands from and sends its results to the IU in a way that mimics the 68882's communication with its host CPU. The pipeline's first stage S_1 (referred to as the floating-point conversion unit) reformats input and output operands to meet IEEE 754 requirements, and is the only stage that communicates with the IU. Stage S_1 also has an ALU for comparing input exponents, as required in floating-point addition or subtraction. The second stage S_2 (the floating-point execution unit) contains a large (67 bit) ALU, a fast barrel shifter, and an array multiplier; this stage is responsible for executing all major operations on mantissas. The final stage S_3 (the floating-point normalization unit) rounds off and normalizes results; it also deals with exceptional cases. Various buses shown in simplified form in Figure 4.58 provide bypass and feedback paths through the pipeline. The clocking of the pipeline is complicated by the need to use several cycles to transfer long operands so that the minimum delay of each stage is two cycles. The delay of a floating-point operation can vary from 2 clock cycles to more than 100 in the case of the FSQRT instruction.

Certain instructions such as FABS, FMOVE, and FNEG are executed entirely within stage S_1 and thus have a delay of two clock cycles. The add and subtract instructions FADD and FSUB use all three stages of the FPU and have a delay of three. These instructions see a pipeline whose organization resembles that of Figure 4.50, with the latter's middle stages S_2 and S_3 merged into the 68040's second (execution) stage S_2. FMUL is executed primarily by the 64 × 8-bit, fixed-point multiplier in S_2. The multiplication of two mantissas requires several passes through the multiplier circuit, which

implements the carry-save multiplication method discussed earlier. Two passes can be made per clock cycle of the pipeline, so that the final set of sum-carry pairs is generated in four clock cycles. An additional cycle through S_2's ALU assimilates the carries and yields the final product. The floating-point division instruction FDIV is implemented by a shift-and-subtract algorithm of the non-restoring type, which requires no special division hardware but takes up to 38 clock cycles.

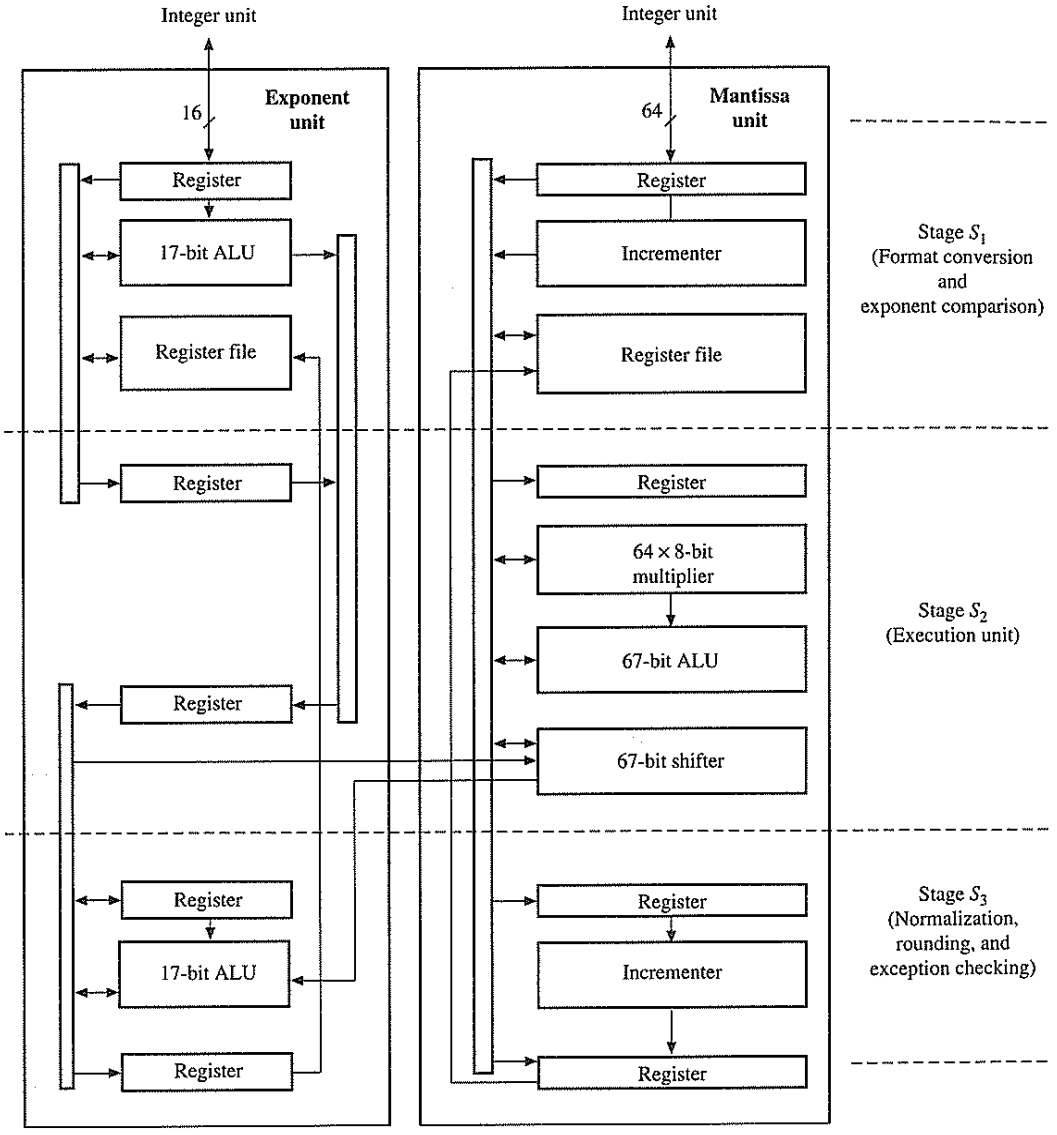


Figure 4.58
Pipelined floating-point unit of the Motorola 68040.

Systolic arrays. Closely related conceptually to arithmetic pipelines are the data-processing circuits called *systolic arrays* [Johnson, Hurson, and Shirazi 1993] formed by interconnecting a set of identical data-processing cells in a uniform manner. Data words flow synchronously from cell to cell, with each cell performing a small step in the overall operation of the array. The data are not fully processed until the end results emerge from the array's boundary cells. A one-dimensional systolic array is therefore a kind of pipeline with identical stages. A two-dimensional systolic array has a structure not unlike the divider array in Figure 4.27, but its cells are sequential rather than combinational. In general, a systolic array permits data to flow through the cells in several directions at once. As in pipelines, buffering must be included within the cells to isolate different sets of operands from one another. The name *systolic* derives from the rhythmic nature of the data flow, which can be compared with the rhythmic contraction of the heart (the systole) in pumping blood through the body. Systolic processors have been designed to implement various complex arithmetic operations such as convolution (problem 4.34), matrix multiplication, and solution techniques for linear equations. We illustrate the concepts involved by a two-dimensional systolic array that performs matrix multiplication.

Let X be an $n \times n$ matrix of fixed-point or floating-point numbers defined by

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots x_{2,n} \\ & & \cdots \\ x_{n,1} & x_{n,2} & \cdots x_{n,n} \end{bmatrix}$$

For brevity we write $X = [x_{i,j}]$, where $x_{i,j}$ is the element in the ith row and jth column of X. The product of X and another $n \times n$ matrix $Y = [y_{i,j}]$ is the $n \times n$ matrix $Z = [z_{i,j}]$ given by

$$z_{i,j} = \sum_{k=1}^{n} x_{i,k} \times y_{k,j} \tag{4.47}$$

A systolic array for matrix multiplication may be constructed from a cell (Figure 4.59*a*) that executes the following multiply-and-add operation on individual numbers (scalars):

$$z := z' + x \times y \tag{4.48}$$

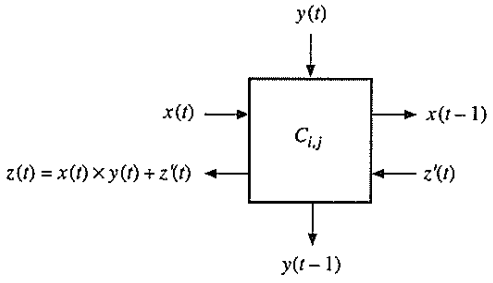
Note that the same type of operation appears in the cell M of the fixed-point array multiplier in Figure 4.53, with 1-bit operands replacing the n-bit numbers used here. Multiply-and-add is also a basic instruction type in recent CPUs such as the PowerPC.

Each cell $C_{i,j}$ of the matrix multiplier receives its x and y operands from the left and top, respectively. In addition to computing z, $C_{i,j}$ propagates its x and y input operands rightward and downward, respectively. The systolic matrix multiplier is constructed from $n(2n - 1)$ copies of $C_{i,j}$, which are connected in the two-dimensional mesh configuration depicted in Figure 4.59*b*. The n operands forming the ith row of X flow horizontally from left to right through the ith row of cells as they might through a one-dimensional pipeline. The n operands forming the jth

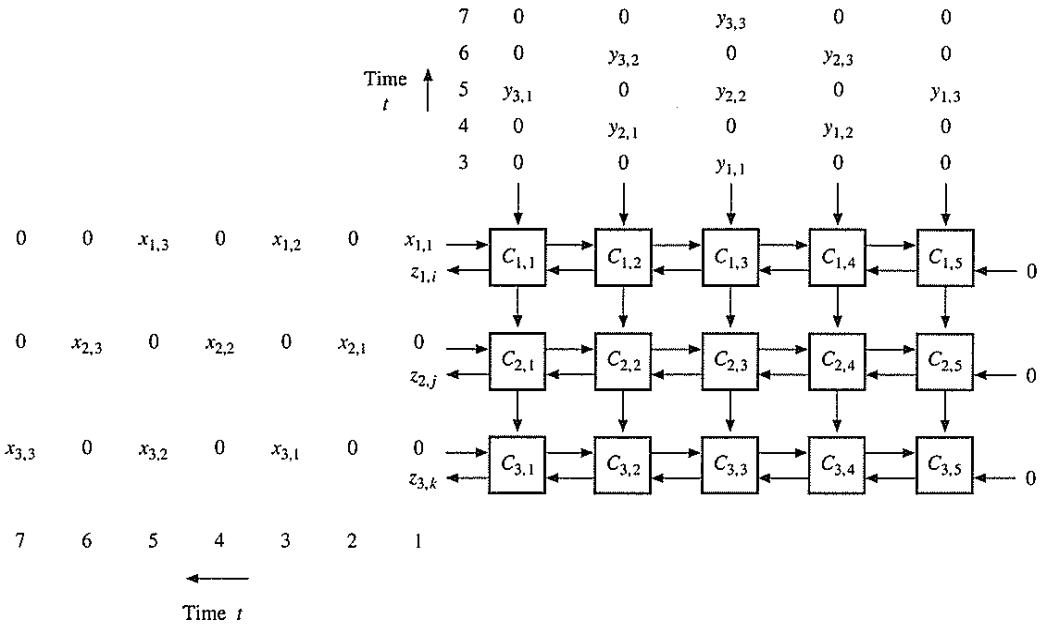
column of $\boldsymbol{Y}$ flow vertically through the jth column of cells in a similar manner. The x and y operands are carefully ordered and separated by zeros as shown in the figure so that the specific operand pairs $x_{i,k}, y_{k,j}$ appearing in (4.47) meet at an appropriate cell of the array, where they are multiplied according to (4.48) and added to a running sum z'. The z's emerge from the left side of $C_{i,j}$, so that there is a flow of partial results from right to left through the cell array. Each row of cells eventually issues the corresponding row of the matrix product $\boldsymbol{Z}$ from its left side.

To illustrate the operation of the matrix multiplier, consider the computation of $z_{1,1}$ in Figure 4.59*b*. Specializing Equation (4.47) for the case where $n = 3$, we get

$$z_{1,1} = x_{1,1}y_{1,1} + x_{1,2}y_{2,1} + x_{1,3}y_{3,1}$$



(*a*)



(*b*)

Figure 4.59
Systolic array for matrix multiplication: (*a*) basic cell and (*b*) 3 × 5 array.

The operand $x_{1,1}$ flows rightward through the top row of cells meeting only zero values of y and z until it encounters $y_{1,1}$ at cell $C_{1,3}$ at time $t = 3$. This cell then computes $z = x_{1,1}y_{1,1} + 0$, which it sends to the cell $C_{1,2}$ on its left. At the same time $C_{1,3}$ forwards $y_{1,1}$ to the second row of cells for use in computing the second row of the result matrix $\boldsymbol{Z}$; it also forwards $x_{1,1}$ to its right neighbor $C_{1,4}$. In the next clock cycle ($t = 4$), $x_{1,2}$ and $y_{2,1}$ are applied to $C_{1,2}$. This cell therefore computes $z = x_{1,2}y_{2,1} + z'$, where $z' = x_{1,1}y_{1,1}$. Finally at $t = 5$, the last pair of operands $x_{1,3}$ and $y_{3,1}$ converge at the boundary cell $C_{1,1}$, which computes $z = x_{1,3}y_{3,1} + z'$, using the value $z' = x_{1,1}y_{1,1} + x_{1,2}y_{2,1}$ supplied by $C_{1,2}$; z is the desired result $z_{1,1}$. At time $t = 6$, $C_{1,1}$ emits a zero, and at $t = 7$, it emits the next element $z_{1,2}$ of $\boldsymbol{Z}$. This process continues until all the elements of the first row of $\boldsymbol{Z}$ have been generated. Concurrently and in a similar way, the remaining rows of cells compute the other rows of $\boldsymbol{Z}$. Note, however, that $x_{i+1,j}$ is produced two cycles later than $x_{i,j}$. The last result $z_{n,n}$ emerges from the array at $t = 4n - 3$. Thus using $O(n^2)$ cells, this systolic array performs matrix multiplication in $O(n)$ time, that is, linear time. Roughly speaking, the array generates n elements of the product matrix $\boldsymbol{Z}$ in one step (two clock cycles in the present example).

The major characteristics of a systolic array can be deduced from the preceding example.

1. It provides a high degree of parallelism by processing many sets of operands concurrently.
2. Partially processed data sets flow synchronously through the array in pipeline fashion, but possibly in several directions at once, with complete results eventually appearing at the array boundary.
3. The use of uniform cells and interconnection simplifies implementation, for example, when using single-chip VLSI technology.
4. The control of the array is simple, since all cells perform the same operations; however, care must be taken to supply the data in the correct sequence for the operation being implemented.
5. If the $\boldsymbol{X}$ and $\boldsymbol{Y}$ matrices are generated in real time, it is unnecessary to store them before computing $\boldsymbol{X} \times \boldsymbol{Y}$, as with most sequential or parallel processing techniques. Thus the use of systolic arrays reduces overall memory requirements.
6. The amount of hardware needed to implement a systolic array like that of Figure 4.59 is relatively large, even taking maximum advantage of VLSI.

Systolic arrays have found successful application in the design of special-purpose arithmetic circuits for digital signal processing, where data must be processed in real time at very high speeds using operations like matrix multiplication.

4.4 SUMMARY

The datapath or data-processing part of a CPU is responsible for executing arithmetic and logical (nonnumerical) instructions on various operand types, including fixed-point and floating-point numbers. The power of an instruction set is often measured by the arithmetic instructions it contains. The arithmetic functions of simpler machines such as RISC processors may be limited to the addition and sub-

traction of fixed-point numbers. More powerful processors incorporate multiply and divide instructions and in many cases have the hardware needed to process floating-point instructions as well.

Arithmetic circuit design is a well-developed field. Fixed-point adders and subtracters are easily constructed from combinational logic. The simplest but slowest adder circuits employ ripple-carry propagation. High-speed adders reduce carry-propagation delays by techniques such as carry lookahead. Fixed-point multiplication and division can be implemented by shift-and-add/subtract algorithms that resemble manual methods. The product or quotient of two km-bit numbers is formed in k sequential steps, where each step involves an m-bit shift and, possibly, a km-bit addition or subtraction. Division is inherently more difficult than multiplication due to the problem of determining quotient digits. Both multipliers and dividers can be implemented by combinational logic array circuits but at a substantial increase in the amount of hardware required.

The simplest ALU is a combinational circuit that implements fixed-point addition and subtraction, typically using the carry-lookahead method; it also implements a set of bitwise (word) logical operations. Multiplication and division algorithms of the shift-and-add/subtract type can be realized by adding a few operand registers—an accumulator AC, a multiplier-quotient register MQ, and a multiplicand-dividend register MD—as well as a small control unit. Datapath units usually contain an addressable register file—in effect, a small, high-speed RAM—to store ALU operands. The register file has several IO ports to allow operands in several different registers to be accessed simultaneously. Bit slicing is a useful technique for constructing a large ALU from multiple copies of a small ALU slice. Multicycling allows a small ALU to process large operands at lower hardware cost but more slowly than bit slicing.

Floating-point and other complex operations can be implemented by an autonomous execution unit within the CPU or by a program-transparent extension to the CPU called a coprocessor. A floating-point processor is typically composed of a pair of fixed-point ALUs—one to process exponents and the other to process mantissas. Special circuits are needed for normalization and, in the case of floating-point addition and subtraction, exponent comparison and mantissa alignment.

Finally, the throughput of a complex datapath circuit such as a floating-point processor can be substantially increased with low hardware overhead by a technique called pipelining. The operations of interest are broken into a sequence of steps, each of which is implemented by a pipeline stage. Buffering between the stages allows an n-stage pipeline to execute up to n separate instructions concurrently. Hence the pipeline's throughput when executing a long sequence of instructions exceeds by a factor of up to n that of a similar but nonpipelined processor. Systolic arrays extend the pipeline concept from one to two or more data-processing dimensions.

4.5 PROBLEMS

4.1. Figure 4.60 gives the logic diagram of a small arithmetic circuit found in a commercial IC with the IO signals renamed to conceal their identities. (*a*) What is the overall function of this circuit? (*b*) Identify the purpose of every IO signal. (*c*) Why do all input

CHAPTER 7

System Organization

This chapter considers how computers and their major components are interconnected and managed at the processor or system level. It examines the methods used for internal and external communication, as well as the design of input-output (IO) systems. The final topic is the use of multiple processors to achieve high performance, fault tolerance, or both.

7.1 COMMUNICATION METHODS

In recent years computing has become intimately associated with communication. A computer's internal or local communication methods significantly affect its flexibility and performance. External, long-distance communication allows computers to be linked together, for example, via the global Internet network. This section examines the general nature of the local and long-distance communication mechanisms used with computer systems.

7.1.1 Basic Concepts

The difficulty in transferring information among the units of a computer largely depends on the physical distances separating them. We distinguish two cases: *intrasystem* communication, which occurs within a single computer system and involves information transfer over distances of less than a meter; and *intersystem* communication, which can involve communication over much longer distances. Intrasystem communication is primarily implemented by groups of electrical wires called *buses,* which support parallel, that is, word-by-word, data transmission. Intersystem communication, on the other hand, is realized by a variety of physical media, including electrical cables, optical fibers, and wireless (radio) links. Serial

(bit by bit) rather than parallel data transmission is preferred for communication over longer distances. Serial links cost less, are more reliable, and are also easier to control than parallel links. A set of computers and other system components that are linked together over relatively long distances constitute a *computer network.*

Buses. The various processor-level components, CPU, caches, main memory, and IO (peripheral) devices within a computer system communicate via buses. The term *bus* in this context covers not only the physical links among the components, but also the mechanisms for controlling the exchange of signals over the bus.

Figure 7.1 depicts the most basic computer bus structure. Here a single bus, the *system bus,* handles all intrasystem communication. All units share the system bus, therefore at any time only two units can communicate with each other. A typical system bus transaction is a memory read (load) operation that involves the transfer of one or more data words over the system bus from the memory (cache or main) M to the CPU. A memory write (store) operation transfers data over the system bus in the opposite direction. Input-output operations normally involve data transfers between an IO device and M. In all the preceding operations M is a passive or *slave* device with respect to system bus transactions, whereas the CPU can actively control the system bus, that is, serve as a bus *master.* IO devices are normally thought of as slave units, but they can be made into bus masters via control units such as specialized IO controllers or general-purpose IO processors.

As Figure 7.1 indicates, the system bus consists of three main groups of lines: address, data, and control. (Not shown are the lines that distribute electrical power to the bus units.) The address lines, typically 8 to 32 in number, transmit the addresses of data items stored in the system's main memory or IO address space. The data lines, typically 16 to 128 in number, transmit data words over the bus. Finally, the control lines perform such functions as identifying the transaction type (memory read, memory write, IO interrupt, and so forth) and synchronizing communication between fast and slow units.

The characteristics of a system bus tend to closely match those of its host CPU and vary widely between different microprocessor families and even between members of the same family. The evolution of CPUs in speed and word size has been matched by a corresponding evolution in their system buses. For example, the first member of Intel's 80X86 family, the 8086 microprocessor, had internal data and (real) address word sizes of 16 and 20 bits, respectively. The CPU data word size became 32 bits, and the address size 24 bits, with the 80286 microprocessor;

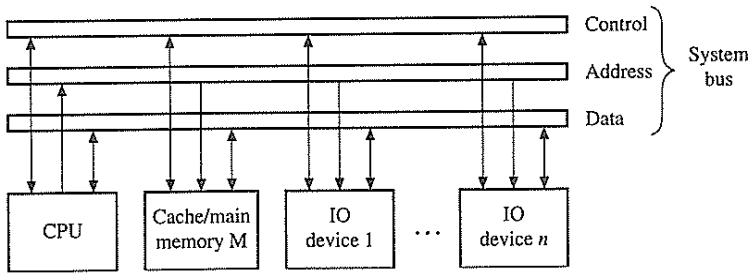


Figure 7.1
Communication within a computer via a single shared bus.

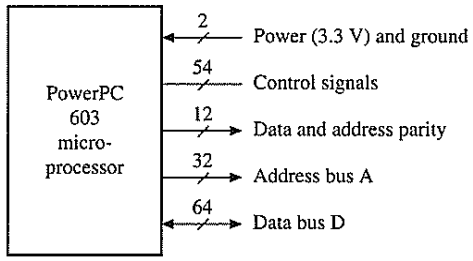


Figure 7.2
System bus of the PowerPC 603 microprocessor.

both became 32 bits with the 80386. The data and address sizes used inside the CPU are often, but not always, the same as those found in the external system bus. The 16-bit 8088, a variant of the 8086 used in the first IBM PC, has an 8-bit external data bus. On the other hand, the Pentium's external data bus is 64 bits wide.

Figure 7.2 outlines the system bus of the PowerPC 603 microprocessor, which is typical of personal computers. It has 64 data-transfer lines D which are bidirectional; that is, they act either as inputs or outputs of the CPU—but not simultaneously. The system bus can transfer from 1 to 8 bytes at a time. Its 32 address lines A allow 2^{32} = 4G memory or IO locations to be specified. Twelve parity check lines, one for each byte of D and A, provide error detection. A large set of control lines supports data transfers, exchange of bus control, interrupt processing, and other bus functions.

The principal use of the system bus is high-speed data transfer between the CPU and M. Most IO devices are slower than the CPU or M and present an external interface that is different from that of the system bus. For example, magnetic-disk units and other secondary memories transfer data serially. Therefore, they need to be connected to the system bus via interface circuits called *IO controllers* that perform series-to-parallel and parallel-to-series format conversions and other control functions. A single IO controller can interface many IO devices to the system bus. This leads to the structure shown in Figure 7.3 in which the IO devices are connected to a separate bus called an *IO bus*.

Computer manufacturers and standards organizations have standardized various IO bus types. For example, the Small Computer System Interface known as the *SCSI* (pronounced "scuzzy") *bus* was adopted as a standard by the American

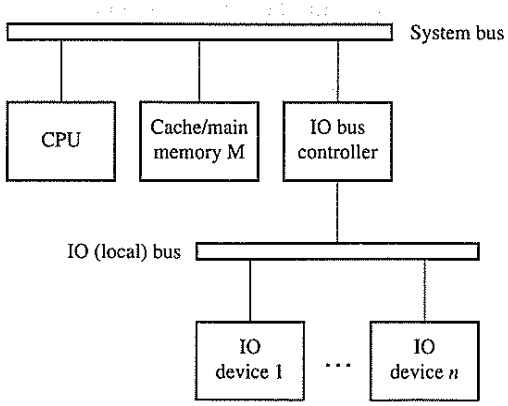


Figure 7.3
Computer with separate system and IO buses.

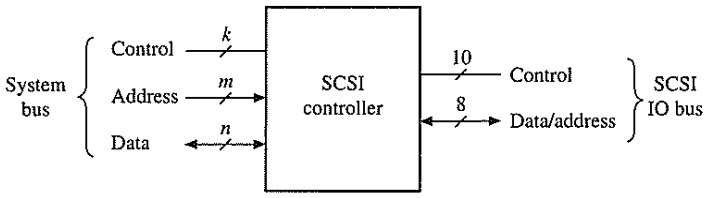


Figure 7.4
The Small Computer System Interface (SCSI) IO bus.

National Standards Institute (ANSI) in 1986. This bus connects IO devices such as hard disk units and printers to personal computers. SCSI was originally designed to transfer data a byte at a time at rates up to 5 MB/s. As can be seen from Figure 7.4, the SCSI bus is smaller and simpler than a system bus like the PowerPC's. Its data subbus is only 8 bits wide and is also used to transfer addresses. Ten additional lines provide all the necessary control functions. Recent extensions to the original SCSI standard have wider data buses (16 and 32 bits), more control features, and higher data-transfer rates.

Another bus with a role similar to SCSI is the so-called Industry Standard Architecture (ISA) bus originally developed by Intel for the IBM PC. Since it allows extra main-memory units as well as IO devices to be added to a computer, it is often referred to as a *local* or *expansion bus,* rather than an IO bus. A more recent bus standard that we examine in detail later is the Peripheral Component Interconnect (PCI) bus, which can transmit 4- or 8-byte words at rates of 500 MB/s or more.

Long-distance communication. There are several important differences between intra- and intersystem communication methods. Whereas intrasystem communication is serial by word, intersystem communication is usually serial by bit because of the difficulty of synchronizing data bits sent in parallel over long distances. Serial transfers also reduce the cost of the communication equipment. Every long-distance data transfer requires a substantial amount of time to establish the communication path to be used, for instance, the time associated with entering a telephone number. To reduce this overhead, a sequence of many bits called a *message,* which corresponds to the concept of block or page in memory systems, is transmitted at one time.

Intrasystem communication is implemented by transmitting digital signals in the form of discrete 0 and 1 pulses over multiline buses. As they are transmitted, the pulses are distorted by variations in the bus's electrical properties, interference between adjacent lines (crosstalk), and similar phenomena collectively known as *noise.* The distortion caused by noise increases with the number of lines in the bus and the signal transmission frequency; it is also affected by the quality of the transmission medium. Beyond some point the pulses become unrecognizable and transmission errors result. Over long distances, therefore, it is more cost-effective to embed the data in analog signals that are transmitted serially, in much the same way as voice traffic has long been sent over telephone lines. Continuous analog signals called *carriers* are generated and varied (modulated) in some manner to produce distinct signal types that denote 0 and 1. A device called a modulator-

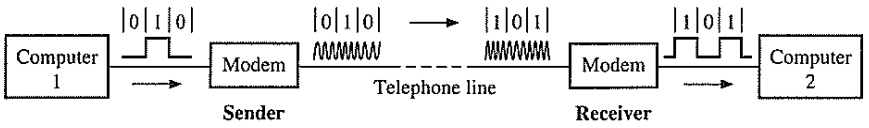


Figure 7.5
Long-distance data transmission using frequency-modulated (FM) signals.

demodulator, or *modem,* converts data between the modulated analog form used for long-distance communication and the pulse form used inside the computer.

Figure 7.5 illustrates the modulation method called *frequency modulation* (FM) used by modems that connect a computer to a low-speed, "voice grade" telephone line. The carrier is a sine wave whose frequency f can be shifted slightly to create two distinct frequency levels: f_0 denoting 0 and f_1 denoting 1. Such signals are heard as beeps of different pitch. Since the 1980s, complex signal-processing techniques have been developed to increase the data-transfer rates over telephone lines from 300 bits/s—bits per second is often denoted bps in this context—to 56,000 bits/s, which is close to the maximum possible. These techniques include the assignment of multiple carrier frequencies to the sender and receiver, error-correcting codes that mask noise-induced errors, and data compression that detects and eliminates redundant information in the data being transmitted.

Digital communication networks, that is, networks designed expressly for transmitting information in digital form, can achieve much higher data-transfer rates. An example is the *integrated services digital network* (ISDN), an international standard for transmitting audio, video, and other data in digital form. Although ISDN was originally proposed around 1960, it has only recently been deployed worldwide. ISDN takes advantage of fiber-optic technology and fast communication methods to achieve data-transfer rates of 600 Mb/s or more. Wireless (radio) transmission using orbiting satellites to relay messages can also achieve very high data rates.

Computer networks. Digital communication networks designed to link many independent computers are called computer networks. Their rationale is to permit sharing of computing resources (hardware, software, or data) that are widely dispersed. For communication over distances of a few kilometers or so—within a single office building, for instance—*local-area networks* (LANs) are used. A LAN is a computer network employing data-transmission links that are private to the network in question. Computer networks spread over large geographical areas, that is, *wide-area networks* (WANs), use data-transmission facilities supplied by telecommunications companies, which in many countries are government-owned or -regulated organizations.

Various techniques exist for sharing the links of a computer network that aim at reducing communication costs. One such technique is *message switching,* which uses intermediate switching centers (servers) on long communication paths to store messages and subsequently forward them toward the final destination; this process is called *store and forward.* Messages are collected by each server, where they are organized (grouped into batches) to make efficient use of the data paths connected to that server. Complete message transmission is thus accomplished by a sequence

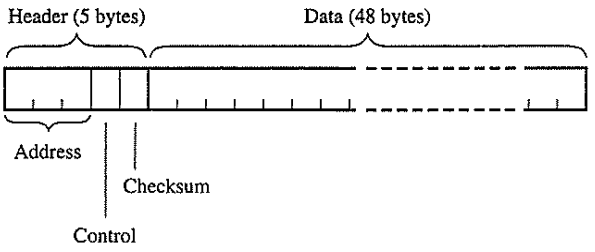


Figure 7.6
Format of a data packet (cell) used in asynchronous transfer mode (ATM) networks.

of *hops* through a variable number of servers. Message switching utilizes the available communication links far more efficiently than circuit switching.

Messages vary greatly in length so that short messages can be delayed while longer messages are being transmitted. This problem is reduced by dividing messages into *packets* of fixed length and format and then transmitting packets from long messages interspersed with packets from short messages. The store-and-forward servers are then responsible for sorting the packets from the various messages and transmitting them to their proper next destinations. Different packages can be sent by different routes dictated by network traffic conditions. At the final destination a message must be reassembled from its constituent packets. This form of communication is called *packet switching* and is used for fast communication of large amounts of data. A type of packet switching called *asynchronous transfer mode* (ATM) combines voice and data communication using short packets that can be transmitted very fast. An ATM packet called a *cell* consists of a 5-byte header containing the destination address and certain control information, followed by a 48-byte data field, as depicted in Figure 7.6.

Although the goal of a universal or *open* computer network to which any manufacturer's computers can be attached remains elusive, the International Standards Organization (ISO) has developed a set of guidelines that provides a common basis for computer network design. These guidelines are known as the *ISO Reference Model for Open Systems Interconnection* (OSI) and define seven functional levels or *layers* through which users exchange messages in a computer network; see Figure 7.7. Each layer is associated with certain network services—error control, for instance—and different computers in a network can be thought of as exchanging information between corresponding layers. Consequently, a distinct set of communication rules or *protocol* can be defined for each layer. In general, layers 1 to 3 of the OSI Reference Model involve services associated with data communications functions close to the network hardware, while layers 5 to 7 involve software (operating systems) functions close to the network user. The intermediate transport layer (layer 4) serves to interface the network's hardware and software.

EXAMPLE 7.1 THE ETHERNET NETWORK ACCESS METHOD [SIMONDS 1994]. Ethernet is a popular bus-oriented architecture for LANs. Its specification involves only the physical and data-link layers, so it is seen as primarily an access method for LANs. Computer-specific hardware (Ethernet controllers) and software (Ethernet drivers) implement the remaining layers of network control. At the physical level an Ethernet LAN has the structure shown in Figure 7.8. Up to 1024 nodes (computers) can be connected via coaxial cable; their maximum separation is limited to 2.8 km. At the data-link level, communication is by messages or *frames* that contain

Layer	Associated services
1. Physical	Electrical and mechanical hardware interfacing to the physical communication medium.
2. Data link	Message setup, transmission, and error control.
3. Network	Establishing message paths in the network (message routing and flow control).
4. Transport	Interfacing network-independent messages with the specific network being used.
5. Session	Creation and management of communication channels between the communicating applications programs.
6. Presentation	Data-transformation services such as character-code translation or encryption.
7. Application	Providing network support functions such as file-transfer routines to application programs (network users).

Figure 7.7
The protocol layers of the Open Systems Interconnection (OSI) Reference Model.

the address, control, and check bits, as well as a variable-length data field. Total message length can range from 64 to 1518 bytes.

A technique called *carrier sense multiple access with collision detection (CSMA/CD)* controls access to the Ethernet and some other LAN types. A node wishing to send a message over the Ethernet first senses (listens to) the main coaxial cable via a tap unit and transmits the message only if it detects no carrier signal, in which case the network is not currently in use. Each message is broadcast throughout the network, and its destination address is examined by all nodes as it reaches them. Only the node whose address matches that of the message header actually reads the message. Since all computers on the network have equal access to the main cable, it is possible for two nodes to begin message transmission at the same time. Consequently, as it transmits a message, a node monitors the actual signals on the cable and compares them with the signals that the node itself is transmitting. If the transmitted and detected signals differ,

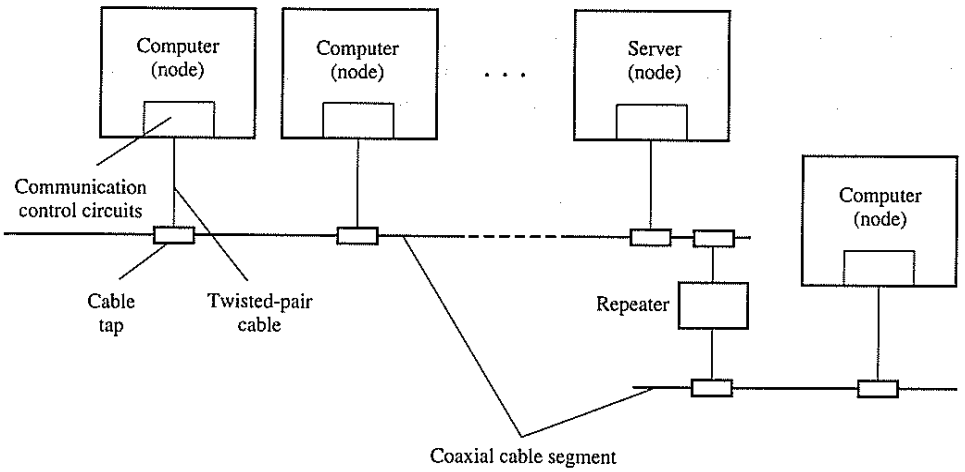


Figure 7.8
Structure of an Ethernet-based LAN.

which will be the case if another computer is transmitting a message at the same time, then a *collision* is said to have occurred. On detecting a collision, an Ethernet node ceases transmission and tries to transmit the same message again later. The time of retransmission is randomly selected so that the chance of another collision is slight, although repeated collisions do occur.

Measurements of Ethernet performance show that the CSMA/CD access scheme is fair in that if n nodes request continuous access to the network over some period of time T, each node gains access to the network for a period very close to T/n. The bandwidth loss due to collisions, even under heavy traffic conditions, is modest—typically less than 10 percent.

Besides the CSMA/CD method used by Ethernet, another common way of controlling access to a LAN is *token passing,* where each node in turn receives and passes on the right to access the network; this right is represented by a special short message called a *token.* The node that possesses the token has exclusive use of the network for transmitting a message, after which it transmits the token to another (fixed) node. Token passing is often used in ring-structured networks (*token rings*), but is also used for bus-structured LANs (*token buses*). When a token ring is not passing normal messages, the token circulates from node to node around the network. A node having a message to transmit waits until the token reaches it. It then holds the token while it transmits its message. In a ring network, a (nontoken) message is usually passed in one direction from node to node until it reaches the destination node; it can then be returned to the source node to confirm its receipt. After transmitting one message, a node puts the token back into circulation so that all nodes get roughly equal access to the network.

The Internet. As discussed in section 1.3.3, the Internet is a huge, worldwide packet-switched computer network descended from the ARPANET, which pioneered the use of packet switching in the 1970s. Each ARPANET site had a computer called an *interface message processor* (IMP), which performed the store-and-forward functions required for packet switching and connected one or more host computers to the ARPANET. Since many types of computers could be hosts, the IMP acted as a standard interface controller between hosts on its local network and a set of remote network servers. To ensure some degree of fault tolerance, the nodes and internode links were chosen so that at least two disjoint communication paths existed between every pair of IMPs.

The *Transmission Control Protocol/Internet Protocol* (TCP/IP) developed for the ARPANET is used by every Internet server. The main function of the IP protocol is to handle the routing of data packets over the Internet; it corresponds to layer 3 (the network layer) of the OSI Reference Model. In particular, IP breaks messages into packets of about 200 bytes each for transmission to remote servers. An Internet address is 4 bytes long, implying a total of more than 4 billion distinct addresses. It is normally represented by a four-part "dotted" symbolic form like *server1.net2.university3.edu.* Because this address format is hierarchical a node needs only limited routing information, for example, the possible paths to all the networks, but not the individual Internet servers, in the domain *edu* assigned to educational institutions.

An Internet packet is transmitted with a header containing its most recent source address and its final destination address H_D, as well as a sequence number

indicating its position in the original message. The packets leave the first server S_S with consecutive sequence numbers 1,2,3,4,...; however, they can travel by different routes to the server S_D of the final destination H_D and arrive there at different times, not necessarily in the original order. An Internet server that is not on the local network containing H_D retransmits each packet it receives to another server to which it is directly connected, following a routing algorithm that aims to find the fastest path to the ultimate destination. The actual path can vary with network traffic conditions. For example, having sent a packet to server S_i, the current server may decide to send the same package to a different server S_j to avoid network congestion, faulty links, or the like.

An Internet package can pass through dozens of servers before reaching the target server S_D. The TCP program on S_D, which operates within the OSI transport layer, is responsible for assembling packets in their proper sequence and checking to see if any are missing or contain errors. If necessary, TCP can send a message to a remote server requesting it to resend a missing or erroneous package. When all a message's packages have been received in satisfactory condition, TCP merges them to reconstruct the original message, which it forwards over the local network to H_D. A higher-level protocol called the *hypertext transport protocol* (http) enables the Internet to transfer multimedia files easily and efficiently and is the basis for the World Wide Web.

Interconnection structures. A system's interconnection structure can be defined by a graph whose nodes denote components such as computers, memories, communications controllers, and so forth, and whose edges denote communication paths such as buses. A path designed to link only two devices is said to be *dedicated.* A path used to transfer information between different sets of devices at different times is said to be *(time)shared* or *multiplexed.*

A conceptually simple interconnection method is to place dedicated buses between all pairs of components that need to communicate. The general case in which n units must be connected in all possible ways needs $n(n-1)/2$ dedicated buses. Figure 7.9 shows such a system when $n = 4$. Dedicated buses allow very fast information transfer: All n devices can send or receive data simultaneously, and there is no delay due to busy connections. Furthermore, systems with dedicated links are inherently reliable because a link failure affects only the two units connected to that link. These units may still be able to communicate if they can send data to each other via other units. For example, if the bus linking U_1 and U_4 in Figure 7.9 fails, U_1 and U_4 can possibly communicate via U_2 or U_3. The main draw-

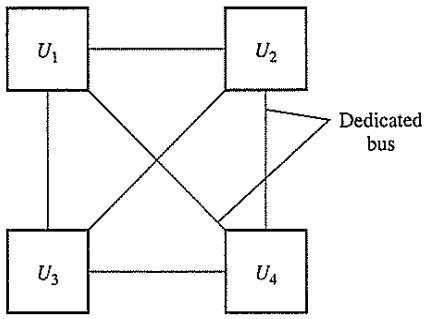


Figure 7.9
System of four units connected by six dedicated buses.

back of dedicated buses is their high cost. The number of buses needed increases as the square of the number of units. Adding a unit to the system is difficult, as the new unit must be physically attached to each existing unit.

At the other end of the spectrum, a single shared bus can provide all communications among n units, as illustrated by Figure 7.1. At any time only two units can communicate with each other via the bus; the remaining units are effectively disconnected from one another. A control method (protocol) is required to supervise sharing of the bus among the n devices. Bus control can be centralized in a special bus-master unit, which can be one of the n communicating units U_i, for example, a CPU. Alternatively, several units can be designed to act as bus masters at different times (decentralized control).

In general, connection to a shared bus is established in two different ways:

- A unit U_i capable of acting as bus master initiates the connection of two units to the bus, perhaps in response to an instruction in a program being executed by U_i.
- A slave unit sends a request to the current bus master for access to the shared bus. The bus master then connects the requesting unit to the link if it is not in use. If the bus is busy, the requesting unit must wait until the bus becomes available. If several conflicting requests are received, the bus master uses some arbitration scheme to decide which request to grant first.

The shared bus is one of the most widely used connection methods in computer systems. Its main advantage is low cost. It is also flexible in that new units can easily be introduced without altering the system's overall structure or the connections to the old units. However, shared buses are relatively slow, since units are forced to wait when the bus is busy. The system is also sensitive to failure of the shared control circuits.

Between the extremes of a set of dedicated buses and a single shared bus lie various interconnection structures that involve some sharing of links, but permit more than one word to be transferred at a time. An example is the *crossbar* network shown in Figure 7.10. A crossbar connects two groups of units $G_1 = \{U_1, U_2, \ldots, U_m\}$ and $G_2 = \{U'_1, U'_2, \ldots, U'_n\}$ so that any unit of G_1 can be connected to any unit of G_2, but two units in the same group need never be connected. For example, G_1 can be a set of memory banks and G_2 a set of processors. Crossbar networks have also been used to connect IO processors to IO devices. As Figure 7.10 shows, each unit in G_1 (G_2) is attached to a shared, horizontal (vertical) bus. The horizontal and vertical buses are in turn connected via a set of $n \times m$ controllers called *crosspoint switches,* which can logically connect any horizontal bus to any vertical bus. At any time only one crosspoint can be active in each row and column. If $k = \min\{m, n\}$, then k units in G_1 can be simultaneously connected to k units in G_2. Hence the crossbar network allows up to k data transfers to take place simultaneously. Access conflicts and delays occur when two units in G_1 attempt to communicate with the same unit in G_2, or vice versa, at the same time.

Many structures employing shared or nonshared buses have been proposed for intra- and intersystem communication in computer systems. More links increase communication speed, but they also increase cost in terms of the buses themselves and their interface circuits. In practice, direct, dedicated connections are provided among only a subset of the communicating units. Units not directly connected must communicate indirectly via intermediate units that relay data in store-and-forward fashion until the final destination is reached. Indirect communication of this type is

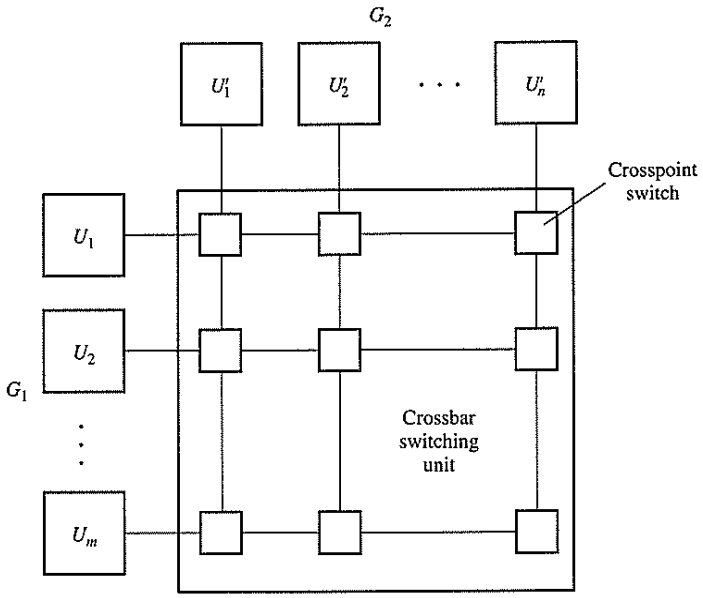


Figure 7.10
Crossbar connection of two groups of units.

slow, and if used extensively, can significantly reduce performance. The amount of such communication occurring depends both on the system's structure and its communication needs. Interconnection structures are therefore selected to balance hardware costs against communication delays for some broad class of applications.

Figure 7.11 shows graphs that abstractly represent some important computer interconnection structures [Feng 1981; Quinn 1994], a few of which we encountered earlier. Here the nodes denote computers or processor-level components such as IO controllers, while the edges denote shared or nonshared buses. The linear or one-dimensional array structure of Figure 7.11*a* models the basic system-bus based structure of Figure 7.1, provided the buses are shared. The mesh (two-dimensional array) structure (Figure 7.11*b*) occurs in the systolic multiplier of Figure 4.59. The ring structure of Figure 7.11*c* adds an extra link to the six-node linear structure, thereby cutting in half the length of the longest path between any two units. It also introduces some tolerance of bus failures by providing two, rather than one, communication paths between each unit pair. The graph of Figure 7.11*d* is called a star for obvious reasons and has a central or root node connected to all $n - 1$ other nodes. The linear and star graphs are special cases of a tree, which is a graph with no cycles. The three-dimensional hypercube is depicted in Figure 7.11*e*, while the complete graph for $n = 6$ nodes appears in Figure 7.11*f*. The ring, hypercube, and complete graphs are considered to be *homogeneous* because all nodes have precisely the same type of connections, making them interchangeable. For instance, each node x has the same number $d(x)$ of neighbors, where $d(x)$ is called the *degree* of x and is a rough indication of the cost of its bus interface. The other examples in Figure 7.11 are not homogeneous, because all nodes do not have the same degree.

Figure 7.12 summarizes some pertinent properties of the preceding interconnection structures. The number of edges and the maximum node degree serve as a

measure of the hardware cost of the structure. The *distance* between two nodes is the number of edges along a shortest path in the graph from one node to the other. The maximum of these distances, called the *diameter* of the graph, is an indication of the worst-case communication delays that can occur. In the examples of Figure 7.12, the total number of connecting edges ranges from approximately $n^2/2$ (for large n) in the complete-graph case to the minimum possible value of $n - 1$ for the linear and star graphs. The complete graph and the star share the largest node degrees, while the linear structure has the largest diameter. The other structures exhibit various compromises between hardware cost and delay. Of particular interest is the hypercube, which achieves a reasonable balance between all three parameters. Therefore, it has been used as the interconnection network in several massively parallel computers.

7.1.2 Bus Control

This section examines the methods to establish and control intrasystem communication via a shared bus [Thurber et al. 1972; Gustavson 1984]. Two key issues are the timing of transfers over the bus and the process by which a unit gains access to the bus. We assume the general structure of Figure 7.1, which applies to most system and IO buses. We also assume that one particular unit acts as the bus master

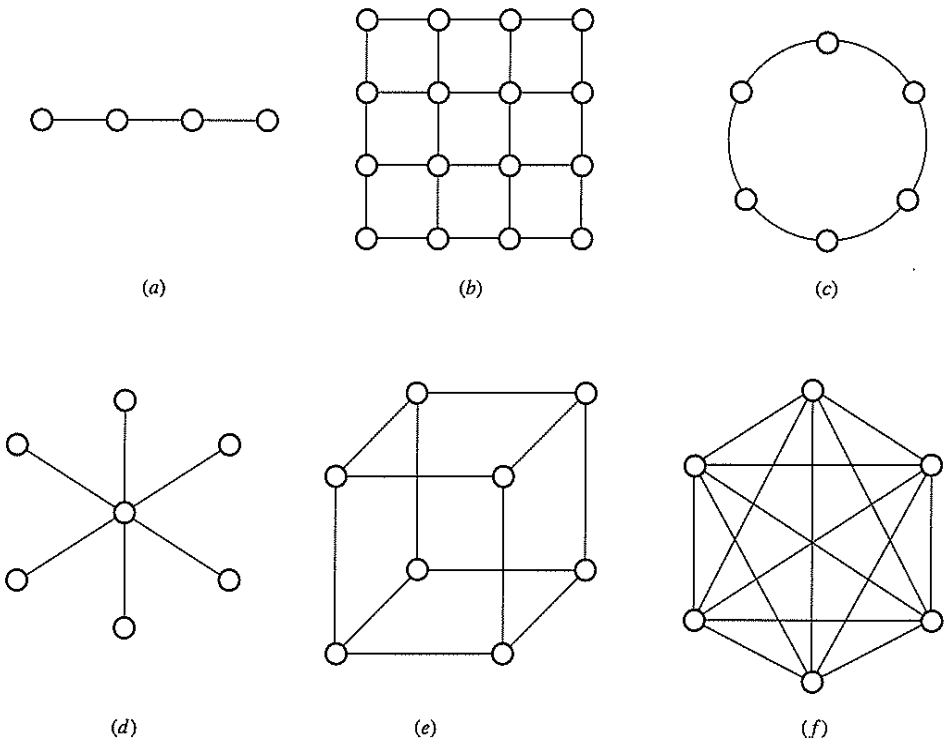


Figure 7.11
Interconnection structures: (*a*) linear; (*b*) mesh; (*c*) ring; (*d*) star; (*e*) hypercube; (*f*) complete.

Interconnection structure	Number of edges (buses)	Maximum node degree	Maximum internode distance
Linear	$n-1$	2	$n-1$
Ring	n	2	$n/2$
Mesh ($n^{0.5} \times n^{0.5}$)	$2(n-n^{0.5})$	4	$2(n^{0.5}-1)$
Star	$n-1$	$n-1$	2
Hypercube ($n=2^k$)	$(n/2)\log_2 n$	$\log_2 n$	$\log_2 n$
Complete	$n(n-1)/2$	$n-1$	1

Figure 7.12
Comparison of the interconnection structures of Figure 7.11 assuming each contains n nodes.

and supervises the use of the bus by the other units, the bus slaves. In many cases the CPU is the bus master, while the memory and IO interface circuits are slaves; IO controllers also serve as bus masters, however. Only a master can initiate data transfers, although slaves can request them. Both master and slave participate equally in the data-transfer process after it is initiated.

Basic features. Buses are distinguished by the way in which data transfers over the bus are timed. In *synchronous* buses each item is transferred during a time slot (clock cycle) known to both the source and destination units. Therefore, the bus interface circuits of both units are in step, or synchronized. Synchronization can be achieved by connecting both units to a common clock source, which is feasible only over very short distances. The rising or falling edge of the clock signal, which is one of the bus's control signals, determines when other bus signals attain stable (valid) states. Alternatively, each bus unit can be driven by separate clock signals of approximately the same frequency. Synchronization signals must then be transmitted periodically between the communicating devices in order to keep their clocks in step with each other.

Synchronous communication has the disadvantage that data-transfer rates are largely determined by the slowest units in the system, so some devices may not be able to communicate at their maximum rate. An alternative approach widely used in both local and (especially) long-distance communications is *asynchronous* timing, in which each item being transferred is accompanied by a control signal that indicates its presence to the destination unit. The destination can respond with another control signal to acknowledge receipt of the item. Because each device can generate bus-control signals at its own rate, data-transfer speed varies with the inherent speed of the communicating devices. This flexibility is achieved at the cost of more complex bus-control circuitry. In local communication where a clock signal is present, data transmission can be asynchronous in the sense that the number of clock periods between bus events (signal changes) can be indeterminate, while the events themselves are synchronized by the clock.

A unit is selected for connection to the main bus in two ways. The bus master can initiate the selection of a slave unit U in response to an instruction in a program or a condition occurring in the system that requires the services of U. Alternatively,

U itself can request access to the shared bus by sending a bus-request signal to the bus master. In each case the master unit must perform a specific sequence of actions to establish a logical connection between *U* and the bus. If several units can generate requests for bus access simultaneously, the bus master needs a way to select one of the units; this selection process is called *bus arbitration*. The CSMA/CD collision avoidance technique used by the Ethernet (Example 6.1) is an example of an arbitration process for LANs.

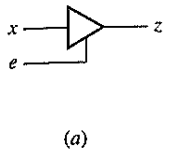
Bus lines fall into three functional groups: data, address, and control lines. The data lines transmit all bits of an n-bit word in parallel. They consist of either two sets of n unidirectional lines or a single set of n bidirectional lines. The data-bus width n is usually a multiple of eight, with $n = 8$, 16, 32, or 64 being common values. The address lines identify a unit to participate in a data transfer. Sometimes the same lines transfer addresses as well as data, a method termed *data-address multiplexing*. This method decreases the cost of the bus, along with the number of external connections (pins) of the units attached to the bus. A computer's system bus usually contains separate address and data lines, but its IO buses often do not; see Figure 7.4, for instance. This difference stems from the fact that an address accompanies each data-word transfer over the system bus, whereas data transfers via an IO bus tend to involve long blocks of consecutive words and need only the starting address of the block. That address can be sent at the start of the data transfer. The control lines convey timing signals and status information about the units attached to the bus; they also identify the type of information present on the data-address lines.

Bus interfacing. A significant contributor to the cost of a bus is the number and type of circuits required to transfer signals to and from the bus. A bus line represents a signal path with potentially very large fan-in and fan-out. Consequently, buffer circuits called bus *drivers* and *receivers* are needed to transfer signals to and from the bus, respectively.

A special transistor circuit technology called *tristate* logic is often used in bus design. It is characterized by the presence of three signal values 0, 1, and Z, where the third value Z is the *high-impedance state*. The binary values 0 and 1 have their usual interpretation, and correspond to two specific electrical states of a line, such as 0 volts and 3.3 volts. The high-impedance state Z, on the other hand, denotes the state of a line that is electrically disconnected from all voltage sources, that is, an open-circuited or floating line. Figures 7.13*a* and *b* define a *tristate buffer,* which serves as a bus-line driver. The inputs x and e are ordinary binary signals that take the values 0 and 1; the output z, however, can take all three values 0, 1, and Z. The tristate buffer (and every other tristate device) has a special input line e called *output enable,* which when set to 0 disables the output line z by changing it to the high-impedance state Z. When $e = 1$, the circuit becomes an ordinary noninverting buffer with $z = x$. Figures 7.13*c* and *d* show equivalent circuits corresponding to the buffer in the enabled and disabled states.

Tristate logic circuits have two big advantages in the design of shared buses:

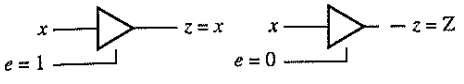
- They greatly increase the fan-in and fan-out limits of bus lines, permitting very large numbers of devices to be attached to the same line.
- They support bidirectional transmission over the bus by allowing the same bus connection to serve as an input port and as an output port at different times.



(a)

Inputs x	Inputs e	Outputs z
0	1	0
1	1	1
0	0	Z
1	0	Z

(b)



(c) (d)

Figure 7.13
Tristate buffer: (*a*) logic symbol; (*b*) truth table; (*c*) equivalent circuit when enabled; (*d*) equivalent circuit when disabled.

Figure 7.14 shows how we use tristate logic to interface two units U_1 and U_2 to a set of bidirectional bus lines. If $e_1 = 1$ and $e_2 = 0$, then U_1 controls or *drives* the bus lines in question; information is transferred over the bus from U_1 to U_2, in effect making $x_{2,i} = z_{1,i}$ for all i. Conversely, if $e_1 = 0$ and $e_2 = 1$, then U_2 drives the bus and information is transferred in the opposite direction from U_2 to U_1, making $x_{1,i} = z_{2,i}$ for all i. If $e_1 = e_2 = 0$, then the outputs of both U_1 and U_2 are logically disconnected from the bus and impose only a minuscule electrical load on it. The combination $e_1 = e_2 = 1$ is invalid, because it applies two different signals simultaneously to each bus line making the line's state indeterminate. Proper operation of the bus requires that at most one driver connected to each bus line be enabled at any time.

The bus lines that can be driven by a particular bus unit U, that is, used by U to send data to other units, depend on U's function in the system. Bus masters have the ability to drive most bus lines, including certain lines that slave units cannot

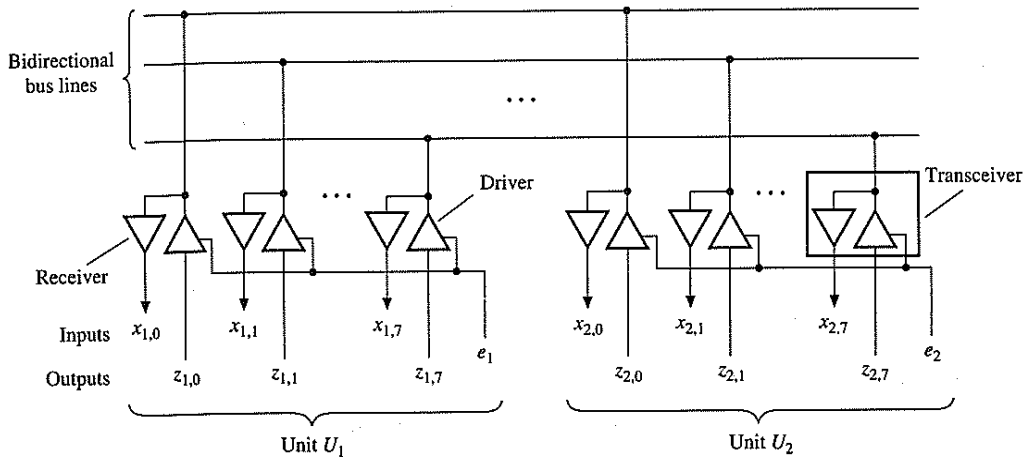


Figure 7.14
Use of tristate logic for bus interfacing.

drive. For example, a CPU can drive all data, address, and most control lines of a system bus. A main-memory unit, which is a bus slave, can drive the data lines but not the address lines, since it only needs to receive information from the address lines.

Timing. The details of some typical data transfers over a bus are shown in Figure 7.15 by means of *timing diagrams*. The *CLOCK* signal of period *T* serves as a timing reference, making this type of transfer synchronous. In this example, the 0-to-1 transition of *CLOCK*, that is, its rising edge, determines when other bus signals are recognized. All active signals must be set up with their new values before the *CLOCK* signal rises. Signal changes are expected to propagate along the bus to their destinations before the next 0-to-1 transition of *CLOCK*.

Figure 7.15 also illustrates some typical signal exchanges between slave and master units; these exchanges follow certain ordering rules called the *bus protocol*. Consider the read operation depicted in Figure 7.15*a*. Communication begins when

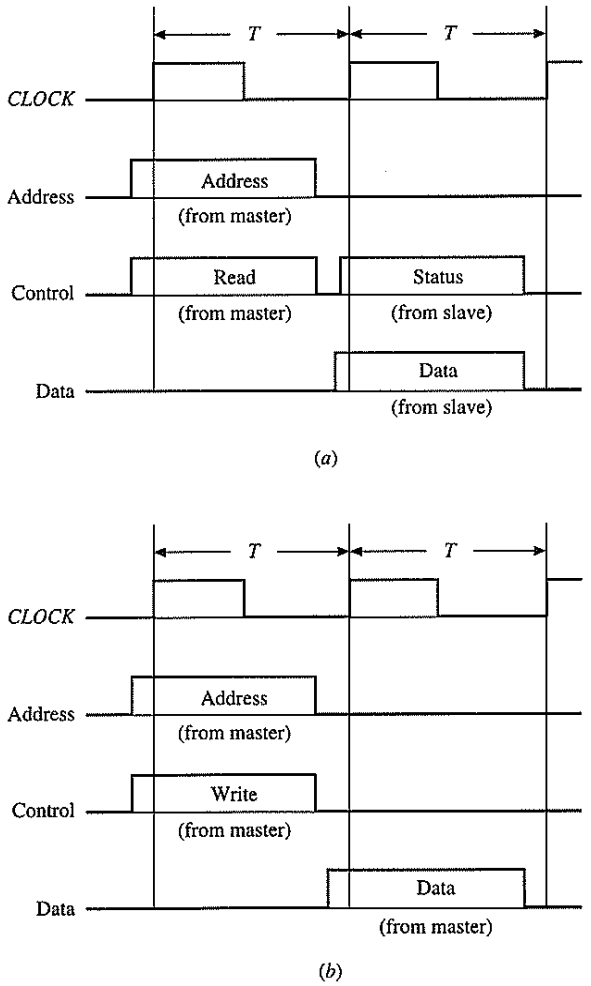


Figure 7.15
Synchronous data transfers:
(*a*) read and (*b*) write.

the bus master places one or more predetermined signals on the control lines specifying the desired bus transaction, for instance, read from memory (load) or read from IO device (input). At the same time, the master places the address of the desired (part of the) slave on the bus's address lines. All potential slave units then examine the active control and address signals. The slave with an address matching that on the bus responds in the next clock cycle by placing the requested data word on the bus's data lines; it can also optionally place status information, for example, (no) error occurred, on certain control lines. A synchronous write operation is similar except that the bus master rather than the slave is the data source; see Figure 7.15*b*. Note that both edges of *CLOCK* can be used as reference points in a bus transaction, and the read or write transactions of Figure 7.15 can be designed to take place during one clock cycle of period 2*T*.

The requirement that the slave respond immediately (in the next clock cycle) to the bus master is lifted by providing a control signal called an *acknowledge* signal *ACK*, as shown in Figure 7.16 for a read bus transaction. *ACK* is controlled by the slave unit and is not activated until the slave has completed its part of the data transfer. The master therefore waits until it has received the *ACK* signal for the current data-word transfer before initiating a new one. Thus an acknowledge signal allows a delay of one or more bus cycles, called *wait states,* to be inserted in a bus transaction to accommodate slow devices. Although *ACK* may be activated in any cycle, its changes are synchronized with those of *CLOCK*. This type of communication is often used between main memory and a CPU. By inserting a variable number of wait states and signaling with *ACK* when it is ready, a memory of essentially any speed can communicate with a faster CPU.

Purely asynchronous timing eliminates the bus's clock signal and replaces it with timing control signals like *ACK*, which are generated by the communicating units. These units are thus self-timed, and units with quite different data-transfer

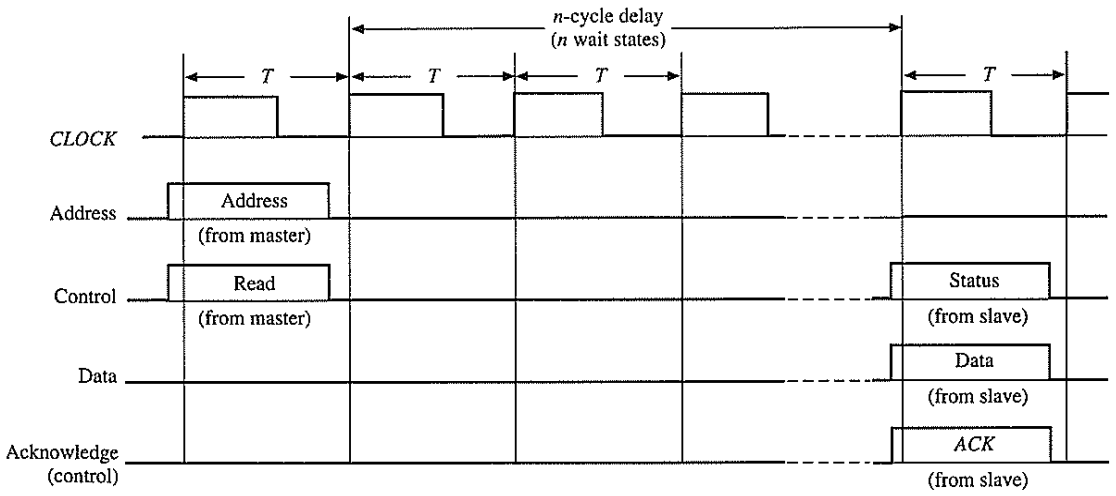


Figure 7.16
Synchronous data transfer (read) with wait states.

rates can communicate asynchronously. We distinguish two cases:

- *One-way* control in which one of the two communicating devices supplies all timing signals.
- *Two-way,* or *interlocked,* control in which both devices generate timing signals.

If one-way control is employed, a single signal controls each address or data transfer. This signal can be activated by the source and destination unit, either one of which can be the bus master. Figure 7.17*a* shows a source-initiated data transfer of this sort. The source places the data word on the data bus. After a brief delay the source activates the control line with the generic name *DATA READY*. The delay is to prevent the *DATA READY* signal from reaching the destination before the data word. Alternatively, the source can activate *DATA READY* and place data on the data bus at the same time. The destination unit must then insert a delay between its receipt of *DATA READY* and its reading of the data bus. The data lines and the *DATA READY* control line must remain in the active state long enough to allow the destination unit to copy the data from the data bus. Figure 7.17*b* shows a data transfer initiated by the destination unit. In this case the destination begins the data transfer by activating the control line *DATA REQUEST*. The source responds by placing the required word on the data lines. Again the data must remain active long enough for the destination unit to read it.

Often the *DATA READY/REQUEST* signals are used to load the data from the source unit to the bus or from the bus to the destination unit. Such control signals are called *strobe* signals and are said to strobe data to or from the bus. For example, the source may generate a data word asynchronously and place it in a buffer register connected to the bus data lines. A signal on *DATA REQUEST* activates the clock input line of the buffer, thereby "strobing" the data onto the bus; Figure 7.18 illustrates this process.

The disadvantage of one-way control is that it does not verify that the data transfer has been successfully completed. For example, in a source-initiated data transfer, the source unit receives no indication that the destination unit has actually received the data transmitted to it. If the destination unit is unexpectedly slow in responding to a *DATA READY* signal, the data may be lost. This problem is eliminated by introducing a second control line that allows the destination unit to send a reply signal to the source when it receives a *DATA READY* signal. This control line has the generic name *DATA ACKNOWLEDGE* or *ACK*. Figure 7.19*a*

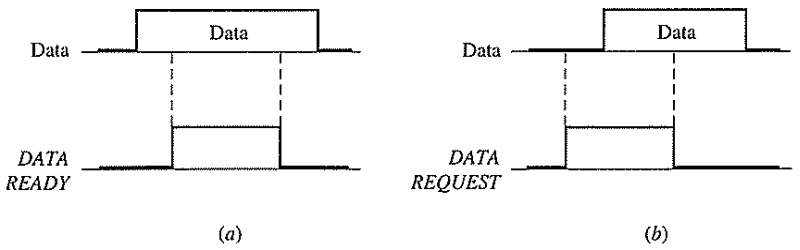


Figure 7.17
One-way asynchronous data-transfer timing: (*a*) source initiated and (*b*) destination initiated.

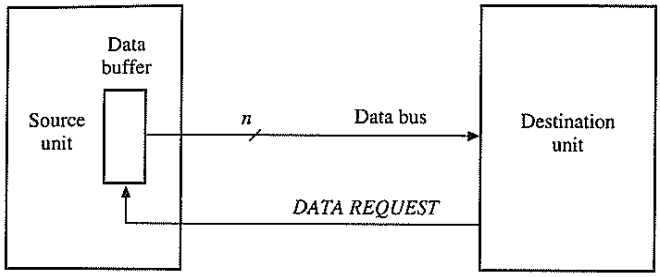


Figure 7.18
Use of a *DATA REQUEST* line to strobe data.

shows the exchange of signals, often called *handshaking,* that accompanies a source-controlled transfer in this case. The source unit maintains the data on the bus until it receives the *ACK* signal. The destination activates *ACK* after copying the data from the bus. This sequence allows delays of arbitrary length to occur during the data transfer. Figure 7.19*b* depicts a similar technique for destination-initiated communication. The source unit activates *ACK* to indicate that the requested data is available on the bus's data lines. The source maintains the data on the bus until the destination unit deactivates *DATA REQUEST,* an action that confirms successful receipt of the data at its destination. As Figure 7.19 demonstrates, a pair of control lines can perform the ready, request, and acknowledge functions for all types of asynchronous bus communications.

Bus arbitration. The possibility exists that several master or slave units connected to a shared bus will request access to the bus at the same time. A selection mechanism called *bus arbitration* is therefore required to enable the current mas-

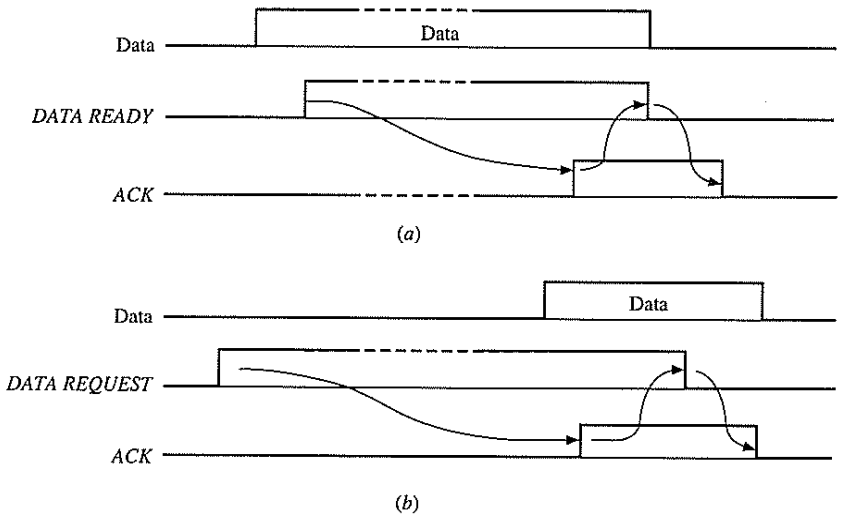


Figure 7.19
Asynchronous data transfer (handshaking): (*a*) source initiated and (*b*) destination initiated.

ter, which we will refer to as the bus controller, to decide among such competing requests. We discuss three representative arbitration schemes: daisy chaining, polling, and independent requesting. These methods differ in the number of control lines they require and in the speed with which the bus controller can respond to bus-access requests of different priorities. Some bus systems combine several distinct arbitration techniques.

Figure 7.20 illustrates *daisy-chaining* arbitration. This method involves three control signals to which we assign the generic names *BUS REQUEST*, *BUS GRANT*, and *BUS BUSY*. All the bus units are connected to the *BUS REQUEST* line. When activated, it merely serves to indicate that one or more units are requesting use of the bus. The bus controller responds to a *BUS REQUEST* signal only if *BUS BUSY* is inactive. This response takes the form of a signal placed on the *BUS GRANT* line. On receiving the *BUS GRANT* signal, a requesting unit enables its physical bus connections and activates *BUS BUSY* for the duration of its new bus activity.

The main distinguishing feature of daisy chaining is the way the *BUS GRANT* signal is distributed; it is connected serially from unit to unit as shown in Figure 7.20. When the first unit requesting access to the bus receives *BUS GRANT*, it blocks further propagation of that signal, activates *BUS BUSY*, and begins to use the bus. When a nonrequesting unit receives the *BUS GRANT* signal, it forwards the signal to the next unit. Thus if two units simultaneously request bus access, the one closer to the bus controller, that is, the one that receives *BUS GRANT* first, gains access to the bus. Selection priority is therefore determined by the order in which the units are linked (chained) by the *BUS GRANT* lines.

Daisy chaining requires very few control lines and embodies a simple, fixed arbitration scheme. It can be used with an essentially unlimited number of bus units. Since priority is wired in, a unit's priority cannot be changed under program control. If it generates bus requests at a sufficiently high rate, a high-priority unit like U_1 can lock out a low-priority device like U_n. A further difficulty with daisy chaining is its susceptibility to failures involving the *BUS GRANT* lines and their associated circuitry. If unit U_i is unable to propagate the *BUS GRANT* signal, then no U_j where $j > i$ can gain access to the bus.

The bus-arbitration scheme called *polling* replaces the *BUS GRANT* line of the daisy-chain method with a set of poll-count lines that are connected directly to all units on the bus, as depicted in Figure 7.21. As before, the units request access to the bus via a common *BUS REQUEST* line. In response to a signal on *BUS*

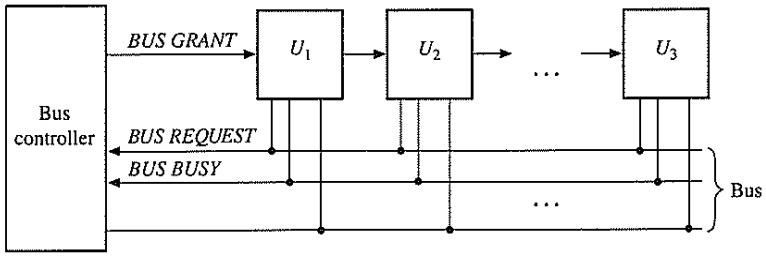


Figure 7.20
Bus arbitration using daisy chaining.

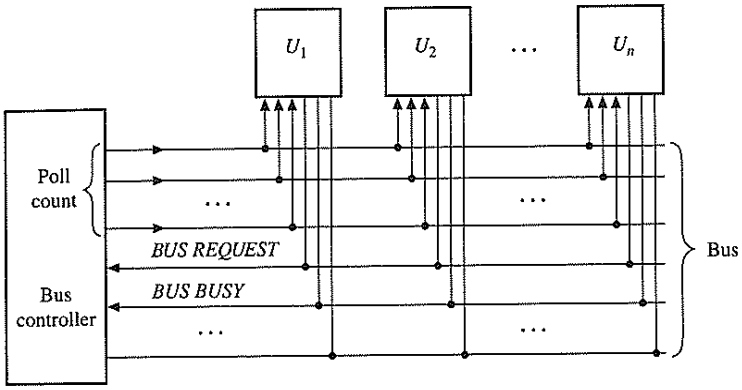


Figure 7.21
Bus arbitration using polling.

REQUEST, the bus controller proceeds to generate a sequence of numbers on the poll-count lines. Each unit compares these numbers, which may be thought of as unit addresses, to a unique address assigned to that unit. When a requesting unit U_i finds that its address matches the number on the poll-count lines, U_i activates *BUS BUSY*. The bus controller responds by terminating the polling process, and U_i connects to the bus.

The priority of a bus unit is determined by the position of its address in the polling sequence. This sequence can be programmed if the poll-count lines are connected to a programmable register; hence selection priority can be altered under software control. A further advantage of polling over daisy chaining is that in polling a failure in one unit need not affect the other units. This flexibility is achieved at the cost of more control lines (k poll-count lines instead of one *BUS GRANT* line). Also, the number of units that can share the bus is limited by the addressing capability of the poll-count lines.

The third arbitration technique, *independent requesting,* has separate *BUS REQUEST* and *BUS GRANT* lines for every unit sharing the bus. This approach,

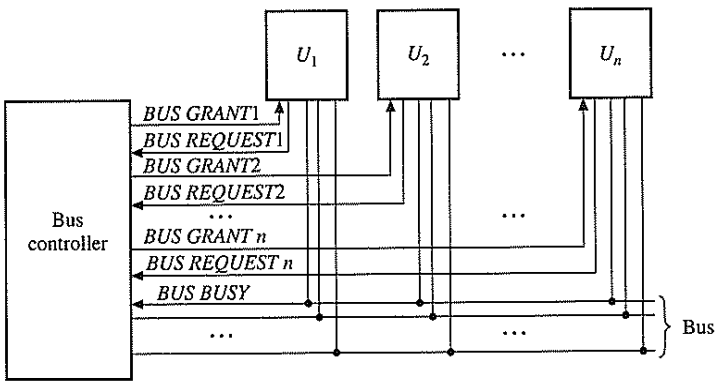


Figure 7.22
Bus arbitration using independent requesting.

which is depicted in Figure 7.22, provides the bus controller with immediate identification of all requesting units and enables it to respond rapidly to requests for bus access. The bus-control unit determines priority, which is programmable. The main drawback of bus control by independent requesting is the fact that $2n$ *BUS REQUEST* and *BUS GRANT* lines must be connected to the bus controller in order to control n devices. In contrast, daisy chaining requires two such lines, while polling requires approximately $\log_2 n$ lines.

EXAMPLE 7.2 THE PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS [SHANLEY AND ANDERSON 1995]. The PCI bus, often referred to as a "local" bus, was developed by Intel in the early 1990s and has since become a widely adopted standard for microprocessor-based computer products such as single-board microcomputers. Unlike some earlier standard buses, the PCI bus is designed to be easily interfaced with different microprocessor families, main memory, and a very wide range of IO devices. Many of the PCI bus's lines are optional, so it can be attached to bus units with as few as 47 pins and as many as 100. It can support either 32-bit or 64-bit data transfers. In version 2.1, the maximum clock rate is 66 MHz, which allows a data-transfer rate of up to 524 MB/s.

The PCI bus is basically intended for attaching IO devices to a computer, but it has many of the characteristics of a high-performance system bus. It can be configured as an IO bus as in Figure 7.3 so that the microprocessor can communicate with memory via its system bus while the PCI bus controller communicates independently with IO devices via the PCI bus. Figure 7.23 shows a different configuration in which the PCI bus has a more central role. Here the PCI bus is linked to the host CPU's "system" bus via a memory controller referred to as a *bridge*, which gives it direct access to the host's main memory. This arrangement, unlike that of Figure 7.3, allows CPU-cache

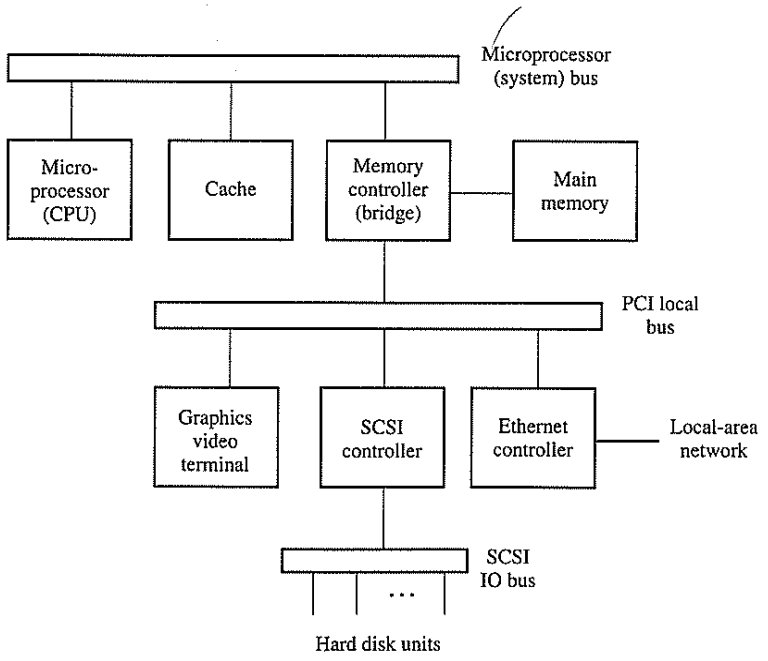


Figure 7.23
Computer system organized around a PCI bus.

and IO-memory transfers to take place simultaneously. High-speed devices such as video terminals and fast network controllers that have little need of the CPU are connected directly to the PCI bus. IO devices intended to conform with other bus standards such as SCSI or ISA can also be interfaced to the PCI bus via appropriate IO controllers, such as the SCSI bus controller in Figure 7.23.

Each PCI device is required to implement a set of registers called its configuration registers, whose format is defined in the PCI bus specification. When the system is first powered up, all such registers are accessed by the system control software to determine which IO devices are currently attached to the PCI bus and their basic communication requirements.

Figure 7.24 summarizes the 100 lines that make up the PCI bus. On the left are the signals required to support basic data transfers using 32-bit or smaller words. On the right are optional lines that support 64-bit data transfers, interrupt control, and other, less-common functions. To reduce pin counts and the size of the connectors needed by PCI-compatible units, addresses and data are multiplexed over a common set of lines denoted *AD*. A typical bus transaction involves two phases: In the first phase, an address is sent over *AD*; in the second phase, one or more data words are sent over *AD*. The remaining lines of the bus perform various control functions, which are outlined below. All bus operations are timed by a clock signal, so the PCI bus is considered to be synchronous; however, ready and acknowledge signals are provided to allow slow devices to insert wait states. Most lines are tristate and are considered inactive in the Z and 0 states, unless they have an overbar, in which case they are inactive in the Z and 1 states.

The command/byte-enable lines perform different functions at different times. During the address-transmission phase, $\overline{C/BE}$ defines a bus command that the bus master uses to tell the bus slave the type of transaction required. The possible commands include memory read, memory write, IO read, IO write, interrupt acknowledge, and a few others. During the data-transmission phase, $\overline{C/BE}$ indicates which bytes of *AD* carry valid data. *PAR* specifies the parity of the 36 bits *AD*[31:0] and $\overline{C/BE}$ [3:0]; it serves the usual function of single-bit error detection. The lines designated basic interface control include $\overline{FRAME}$, which delimits a data-transfer transaction and therefore

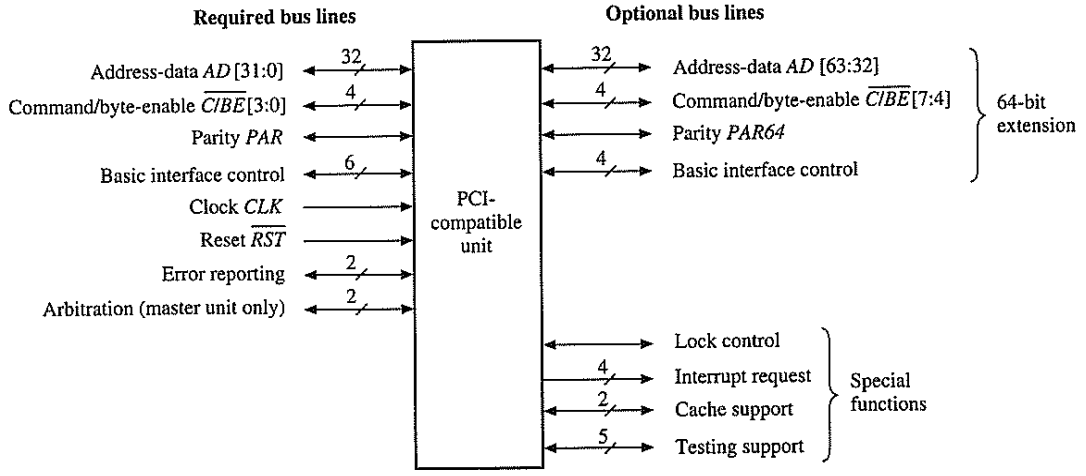


Figure 7.24
Signals of the PCI standard bus.

is active for the duration of the entire transaction; a pair of data-ready/acknowledge lines, $\overline{IRDY}$ (initiator ready) and $\overline{TRDY}$ (target ready), for use by the master and slave, respectively; and a $\overline{STOP}$ line that the slave uses to ask the master to halt the current transaction. The system clock signal *CLK* is responsible for synchronizing all bus transactions, while $\overline{RST}$ resets all bus-control registers attached to the PCI bus. The two error-reporting lines indicate parity errors and related problems.

A pair of lines $\overline{REQ}$ (bus request) and $\overline{GNT}$ (bus grant) control bus arbitration. The bus-arbitration method is not part of the PCI bus's specification, which requires only that the central bus controller receive a single request at a time on the $\overline{REQ}$ line and that all attached bus masters receive their fair share of access to the bus. The daisy-chaining method discussed earlier is easily implemented. Independent requesting can also be implemented without difficulty by means of priority-encoding logic that selects one of several active requests to forward to the PCI bus's $\overline{REQ}$ line.

Figure 7.25 shows a representative three-word data transfer from slave to master via the PCI bus; for example, an IO read operation. The transaction begins when the initiating master unit (which is assumed to already be in control of the bus) activates $\overline{FRAME}$ by setting it to 0 in clock cycle 1; as its name suggests, $\overline{FRAME}$ frames the entire data transfer sequence. The master then places an address and command word (IO read in our example) on the *AD* and $\overline{C/BE}$ lines, respectively; this information should be valid when clock cycle 2 begins. During cycle 2 all the units attached to the bus try to decode the address and command. In this instance an IO unit containing the current address will be successful and will prepare to communicate with the master. In the next cycle the master relinquishes control of *AD* and places valid byte-enable information on the $\overline{C/BE}$ lines for the remainder of the transaction. To avoid conflicts when the master stops driving the *AD* bus (and certain control lines) and the slave begins to do so, an idle, *turnaround* cycle—cycle 3 in Figure 7.25—must follow the address phase. The slave can transmit a sequence of data words via *AD*, beginning in cycle 4 at the maximum rate of one word per clock cycle. The two communicating units control the actual transfer rate via the $\overline{IRDY}$ and $\overline{TRDY}$ lines, which permit any number of wait states to be inserted after each data-transfer cycle.

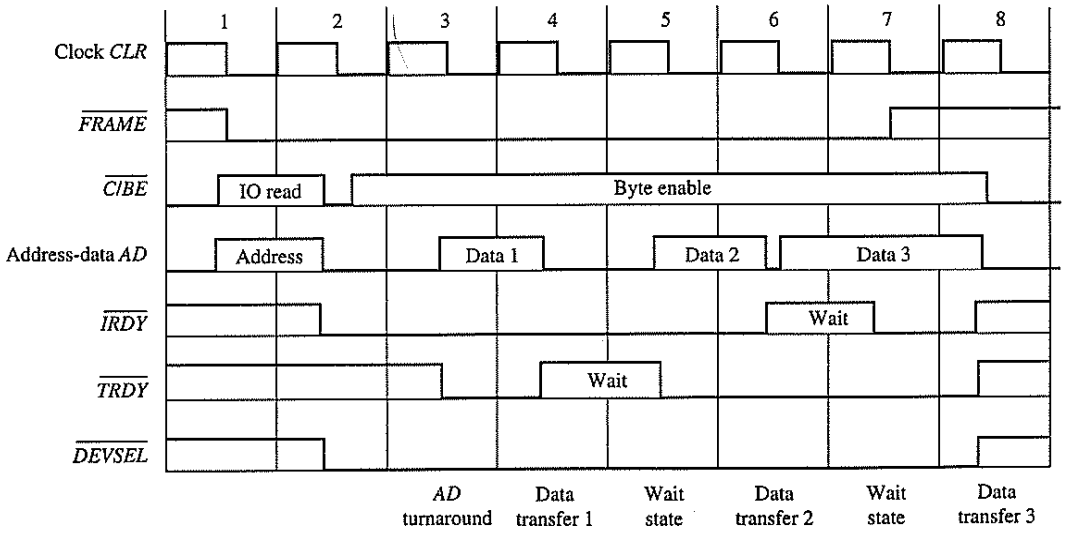


Figure 7.25
Data-transfer transaction (memory read) via the PCI bus.

Data transfer cannot begin until the master activates $\overline{IRDY}$ to indicate that it is ready to receive data; this occurs in cycle 2. The slave makes the data word 1 available and signals this fact by making $\overline{TRDY} = 0$ in cycle 3; the data transfer takes place in cycle 4. In this example the slave immediately deactivates its ready line ($\overline{TRDY} = 1$) making cycle 5 into a wait state; it then reactivates $\overline{TRDY}$ and places data word 2 on *AD* for transmission in cycle 6. The slave places data word 3 on *AD* for transmission in cycle 7. This time, however, the master decides to insert a wait state by making $\overline{IRDY} = 1$ for one clock cycle. Consequently, data word 3's transfer is delayed until cycle 8. The master deactivates $\overline{FRAME}$ in cycle 7 to signal that the following cycle marks the end of the bus transaction. The last control line $\overline{DEVSEL}$ (device select) shown in the figure is activated by the slave device in cycle 2 to indicate that the slave has successfully decoded the address and is the target of the current bus transaction. No data transfer can occur until $\overline{DEVSEL}$ is active, so this line serves to tell the master when a bus transaction cannot be completed due to a missing or faulty slave unit.

A write transaction (where the master is the data source rather than the slave) is very similar to that of Figure 7.25. No turnaround cycle is needed after the address-transfer phase, because the master continues to drive *AD* throughout the transaction.

7.2 IO AND SYSTEM CONTROL

The main data-processing functions of a computer involve its CPU and external (cache-main) memory M. The CPU fetches instructions and data from M, processes them, and eventually stores the results back in M. The other system components—secondary memory, user interface devices, and so on—constitute the input-output (IO) system. In this section we discuss the hardware and software needed to implement IO operations. We also discuss operating systems—the supervisory programs that manage a system's major resources including the CPU, main memory, and IO subsystems.

IO control methods. Input-output operations are distinguished by the extent to which the CPU is involved in their execution. (Unless otherwise stated, *IO operation* refers to a data transfer between an IO device and M, or between an IO device and the CPU.) If such operations are completely controlled by the CPU, that is, the CPU executes programs that initiate, direct, and terminate the IO operations, the computer is said to be using *programmed IO*. This type of IO control can be implemented with little or no special hardware, but causes the CPU to spend a lot of time performing relatively trivial IO-related functions. One such function is testing the status of IO devices to determine if they require servicing by the CPU.

A modest increase in hardware enables an IO device to transfer a block of information to or from M without CPU intervention. This task requires the IO device to generate memory addresses and transfer data to or from the bus (system or local) connecting it to M via its interface controller; in other words, the IO device must be able to act as a bus master. The CPU is still responsible for initiating each block transfer. The IO device interface controller can then carry out the transfer without further program execution by the CPU. The CPU and IO controller interact only when the CPU must yield control of the memory bus to the IO controller in response to requests from the latter. This level of IO control is called

direct memory access (DMA), and the IO device interface control circuit is called a *DMA controller*.

The DMA controller can also be provided with circuits enabling it to request service from the CPU, that is, execution of a specific program to service an IO device. This type of request is called an *interrupt*, and it frees the CPU from the task of periodically testing the status of IO devices. Unlike a DMA request, which merely requests temporary access to the system bus, an interrupt request causes the CPU to switch programs by saving its previous program state and transferring control to a new interrupt-handling program. After the interrupt has been serviced, the CPU can resume execution of the interrupted program. Most computers have DMA and interrupt facilities, which are supported by special DMA and interrupt control units.

A DMA controller has partial control of IO operations. Essentially complete control of IO operations can be relinquished by the CPU if an IO *processor* (IOP) is introduced. Like a DMA controller, an IOP has direct access to main memory and can interrupt the CPU; however, an IOP can also execute programs directly. These programs, called *IO programs*, may employ an instruction set different from the CPU's—one that is oriented toward IO operations. It is common for larger systems to use general-purpose microprocessors as IOPs. An IOP can perform several independent data transfers between main memory and one or more IO devices without recourse to the CPU. Usually the IOP is connected to the devices it controls by a separate bus system, the IO bus, as illustrated in Figure 7.3.

7.2.1 Programmed IO

First we examine programmed IO, a method included in every computer for controlling IO operations. It is most useful in small, low-speed systems where hardware costs must be minimized. Programmed IO requires that all IO operations be executed under the direct control of the CPU; in other words, every data-transfer operation involving an IO device requires the execution of an instruction by the CPU. Typically the transfer is between two programmable registers: one a CPU register and the other attached to the IO device. The IO device does not have direct access to main memory M. A data transfer from an IO device to M requires the CPU to execute several instructions, including an input instruction to transfer a word from the IO device to the CPU and a store instruction to transfer the word from the CPU to M. One or two additional instructions may be needed for address computation and data-word counting.

IO addressing. In systems employing programmed IO, the CPU, M, and IO devices usually communicate via the system bus. The address lines of the system bus that are used to select memory locations can also be used to select IO devices. An IO device is connected to the bus via an *IO port,* which, from the CPU's perspective, is an addressable data register, thus making it little different from a main-memory location.

A technique used in many machines, such as the Motorola 680X0 series, is to assign a part of the main-memory address space to IO ports. This technique is called *memory-mapped IO*. A memory-referencing instruction that causes data to

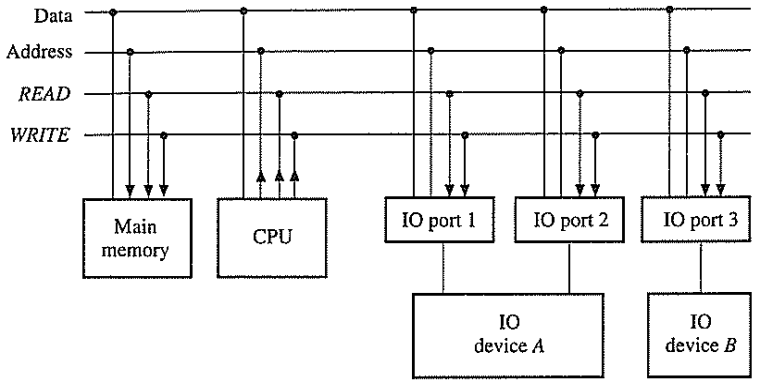


Figure 7.26
Programmed IO with shared memory and IO address space (memory-mapped IO).

be fetched from or stored at address X automatically becomes an IO instruction if X is made the address of an IO port. The usual memory load and store instructions are used to transfer data words to or from IO ports; no special IO instructions are needed. Figure 7.26 shows the essential structure of a computer with this type of IO addressing. The control lines *READ* and *WRITE*, which are activated by the CPU when processing a memory reference instruction, are used to initiate either a memory access cycle or an IO transfer.

In the organization shown in Figure 7.27, sometimes called *IO-mapped IO*, the memory and IO address spaces are separate. This scheme is used, for example, in the Intel 80X86 microprocessor series. A memory-referencing instruction activates the *READ M* or *WRITE M* control line which does not affect the IO devices. The CPU must execute separate IO instructions to activate the *READ IO* and *WRITE IO* lines, which cause a word to be transferred between the addressed IO port and the CPU. An IO device and a memory location can have the same address bit pattern without conflict. A minor modification of the circuit of Figure 7.27 can merge the memory and IO address spaces, if desired.

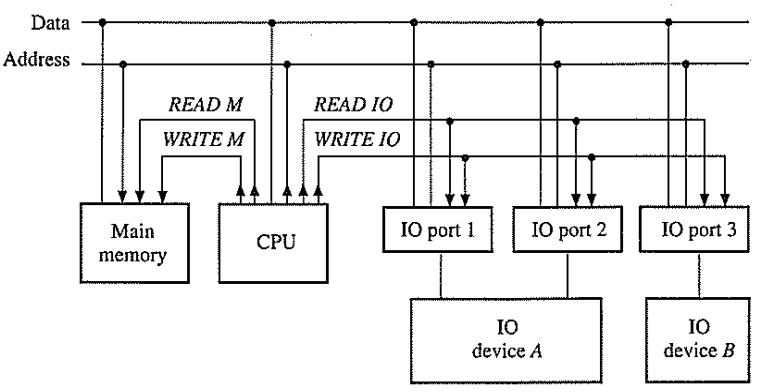


Figure 7.27
Programmed IO with separate memory and IO address spaces (IO-mapped IO).

"intelligent IO subsystem" [Intel 1996]. Figure 7.48 indicates the complexity of this IC. The IOP is built around the 80960 microprocessor, a member of the i960 family of pipelined 32-bit RISCs. The 80960's instruction set is noteworthy for its fast implementation of call and return instructions, its high-performance integer ALU, and its large register file. The core processor also contains a 4KB two-way set-associative I-cache and a 2KB direct-mapped D-cache. To provide quick response to interrupts, the i960 RP allows the programmer to permanently lock critical IO routines such as interrupt handlers in its I-cache.

The i960 RP IOP supports a pair of 32-bit PCI buses: a primary bus for connection to the host CPU and a secondary bus for IO devices. It also has a 32-bit internal "local" bus—the 80960's system bus—to which IO devices can be attached, as well as some specialized IO buses such as the I^2C (inter-integrated circuit) bus, a serial IO bus developed by Philips Semiconductor. Not surprisingly, this single-chip device has a very large number of IO pins—352 in all. The i960 RP IOP has controllers for three independent DMA channels, two dedicated to the primary PCI bus and one to the secondary PCI bus. It also has flexible controllers to support vectored interrupts, including the advanced programmable interrupt control (APIC) interface used by the Pentium and other Intel microprocessors.

7.2.4 Operating Systems

Except when it is dedicated to a single task, a computer is usually managed by a supervisory program called an operating system, which provides a uniform software interface for other system programs and for applications programs. In multiuser environments the operating system controls such shared resources as CPU time, memory space, IO devices, utility programs, and databases [Silberschatz 1994].

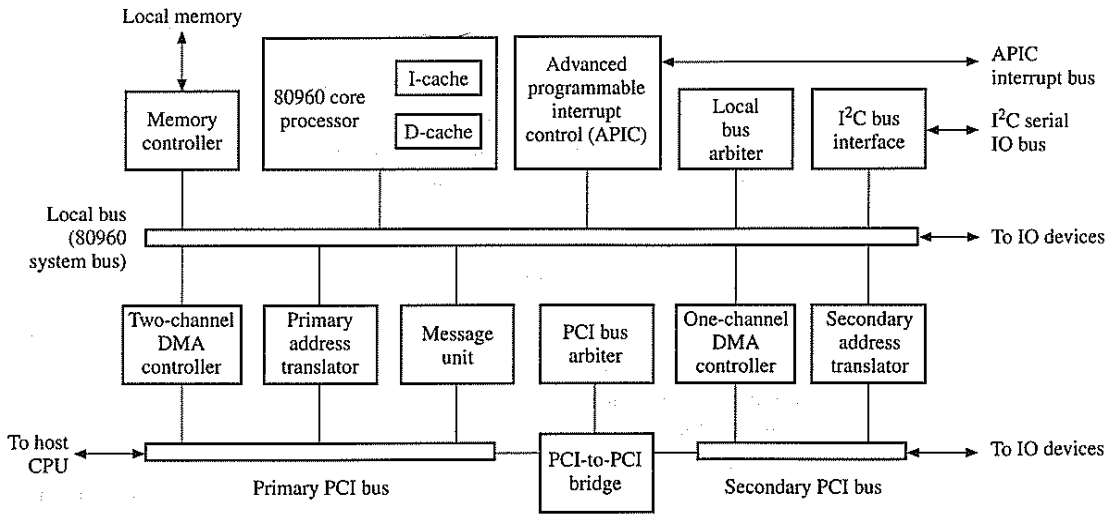


Figure 7.48
Structure of the Intel i960 RP input-output processor.

Introduction. Programs use a computer's resources in various, and often unpredictable, ways. Resource requirements also change dynamically during the execution of a single program. For example, programs often alternate between computations that use the CPU and IO operations that use IOPs and peripheral devices but do not require the CPU. If several programs are available for execution at the same time, then the computer's performance as measured by overall throughput can be improved by assigning one program to the CPU while others are assigned for execution by IOPs. The scheduling of CPU and IO processing is a typical function of an operating system. Another important shared resource is memory, both main and secondary, whose management is also typically an operating system task.

Several types of operating systems have evolved over the years. The earliest system control programs (batch monitors and spooling systems) were mainly concerned with reducing the time required for IO operations involving user programs. Modern operating systems attempt to manage a wide range of computer resources efficiently—not just IO devices. They provide textual or graphical interfaces that allow users to interact directly with the operating system by specifying the resources needed for a particular job. Current operating systems have their origins in several influential systems developed in the 1960s, such as IBM's OS/360, which became a de facto standard for mainframe computers. Early work at Manchester University (Atlas), MIT (Multics), and elsewhere led to the UNIX operating system, which was developed at Bell Laboratories in the mid-1970s and is now in wide use, especially in workstations.

Processes. The basic unit of computing managed by an operating system is a *process,* which is loosely defined as a program module in the course of execution. The resources needed by a process, including processors and memory space, are allocated to it dynamically during execution. Examples of processes are a procedure executed by a CPU and an IO program executed by an IOP. A process can be created in response to a user command to the operating system. Processes can also be created by other processes, for example, in response to interrupts. When no longer needed, a process (but not the underlying program) is deleted by the operating system, and the resources currently allocated to the process are released. While in existence, a process has three major states: ready, running, and blocked, as depicted in Figure 7.49*a*. In the ready state a process is waiting, perhaps in a queue with other processes, for the resources that it needs to enter the running or active state. A blocked process is waiting for some event to occur, such as completion of another process that provides it with input data. A transition from one process state to another is triggered by conditions such as interrupts and user instructions to the operating system.

Figure 7.49*b* shows the state behavior of a typical user process P in a system with an independent IOP. It is assumed that P runs on the CPU until an IO instruction is encountered, at which point the operating system intervenes and changes P from running to blocked. P can also be terminated by a timer-generated interrupt, which the operating system uses to limit the amount of time that any one process is assigned to the IOP. In this case P is returned to the ready state, where it remains until rescheduled for execution by the operating system via the CPU. A new process P' can now be created to run on an IOP and carry out the required IO opera-

tion. Completion of P' results in an IO interrupt that causes the CPU to transfer P from blocked to ready. At this point P' can be deleted if it is no longer needed by P or other active processes. As soon as the CPU is available to execute P, that is, when there are no CPU processes of higher priority ready for execution, P is transferred once more to the running state. It continues running until it encounters another IO instruction, exceeds its allocated time, or completes execution. In the last case a call is made to the operating system, which can then delete P.

Kernel. An operating system comprises many resource management programs, including processor scheduling routines, virtual memory management routines, and IO device control programs (device drivers). Common utility programs, such as compilers, text editors, and the like, often form part of the operating system. Thus operating systems tend to contain more software than can fit comfortably in main memory. The part of an operating system that resides more or less continuously in main memory and consists of its most frequently used parts is termed the *kernel* or *nucleus*. The other, less frequently used parts, such as file management routines, reside in secondary (disk) memory and are transferred to main memory when needed.

The kernel of an operating system is responsible for the creation, deletion, and state switching of the many processes that define a computer's behavior. The kernel performs its tasks by quickly responding to a steady flow of interrupt requests. These requests have many sources such as user-generated requests for operating system services; CPU-generated process time-outs; memory faults; IO operations; and hardware or software errors. The kernel achieves rapid response by briefly disabling other interrupts while responding to the current one and then dispatching or, if necessary, creating a system process to execute the appropriate interrupt-handling routine. The performance and reliability of the kernel can be improved by implementing its more basic functions in hardware or firmware.

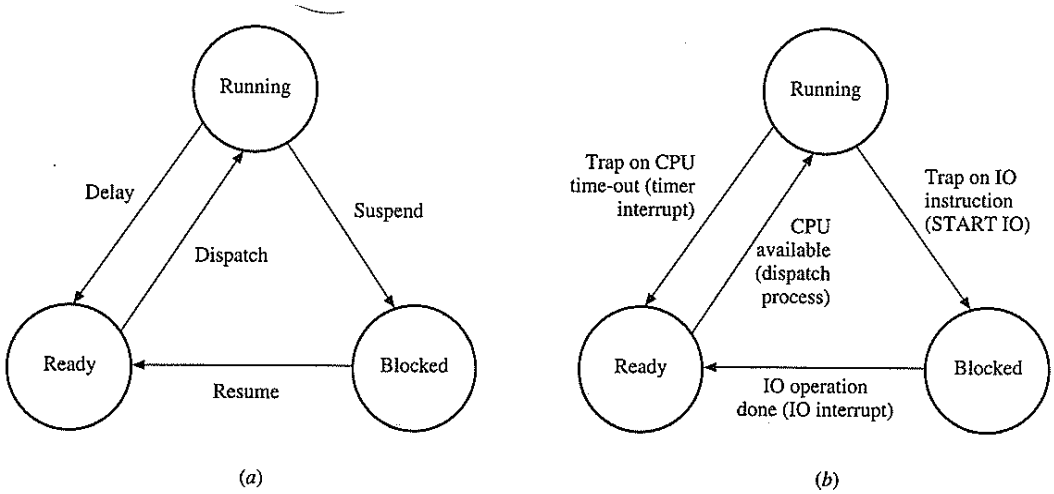


Figure 7.49
Process behavior: (*a*) general case and (*b*) CPU process in system with IOP.

The kernel keeps track of each process by means of a data segment called a *process control block* PCB, which defines the most recent execution state or context of the process. The PCB typically contains all the programmable registers associated with a process, including its program counter, stack pointers, status register, and general-purpose data and address registers. The PCB normally resides in main memory. When the process is about to be executed, its PCB is transferred to the corresponding processor registers. The transfer of control from one process to another (*context switching*) is implemented by saving the context of the old process in its PCB in memory and loading the PCB of the new process into the processor in its place.

Figure 7.50 shows the PCB used by the VMS operating system for the Digital VAX computer series. This PCB contains several stack pointers used by the operating system, the CPU's general registers, the program counter PC, and the program status word PSW. The latter stores CPU status (flag) bits and the interrupt priority level of the process. The last entries in the PCB specify the base address and length of two page tables: one for the user program and one for the user stack. Page tables play an essential role in the firmware-implemented address mapping that manages the VAX's virtual memory. Two VAX instructions SVPCTX (save process context) and LDPCTX (load process context) support context switching by transferring the complete PCB to and from memory, respectively.

An operating system supervises a potentially large set of processes that function asynchronously and concurrently. Many of the more subtle problems in designing an operating system are due to attempts by concurrent processes to use shared resources in undesirable or improperly synchronized ways. Next we consider two basic problems in concurrency control—mutual exclusion and deadlock—and their solutions.

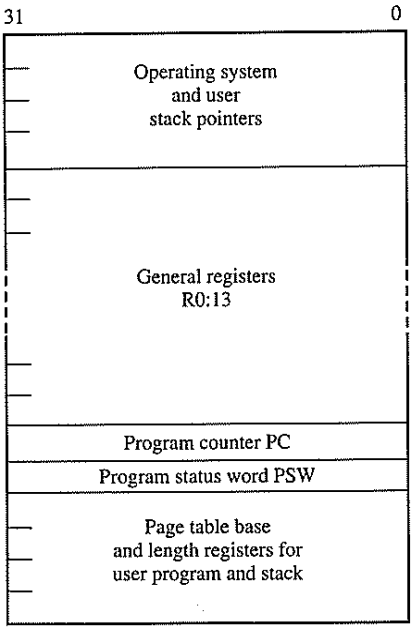


Figure 7.50
Process control block PCB for the VMS operating system.

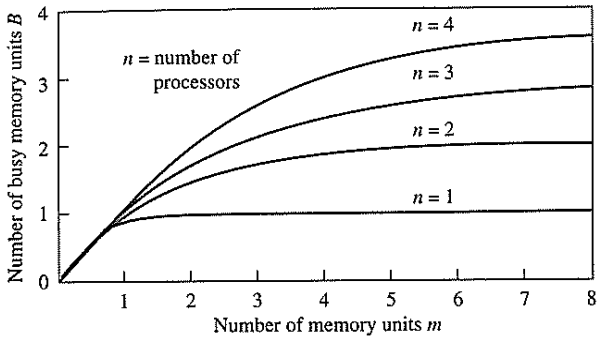


Figure 7.62
Performance of a shared-memory multiprocessor.

similar interconnection network, as in Figure 7.60*b*. All programs and data used by the processors are stored in the m-unit global memory and are accessed via the crossbar network. It is reasonable to assume that the instruction or data bandwidth b of a processor P_j is proportional to the rate at which P_j accesses memory. The latter is, in turn, proportional to the average number of busy memory units B. Suppose further that the probability of any processor P_j generating a request to M_i is $1/m$; in other words, the memory requests are distributed uniformly. Hence the probability that M_i is idle, and therefore free to respond to memory requests, is $(1 - 1/m)^n$; the probability p_i that M_i is busy is $1 - (1 - 1/m)^n$. If M_i is busy when a new request for access to it is received, that request is not serviced until M_i becomes free again. The average number of busy memory units B is therefore given by

$$B = \sum_{i=1}^{m} p_i = m\left[1 - \left(1 - \frac{1}{m}\right)^n\right] \tag{7.17}$$

As might be expected, if m is fixed and n approaches infinity ($n \to \infty$), then $B \to m$. Similarly, if n is fixed and $m \to \infty$, then Equation (7.17) implies $B \to n$; that is, all processors become busy. Figure 7.62 plots B against m for some small values of n. From this analysis we see that we can improve the performance of a multiprocessor by placing information that is frequently accessed by P_j in a local memory assigned to P_j while limiting the use of global memory to the storage of infrequently shared programs and data.

7.3.2 Multiprocessors

A *multiprocessor* is an MIMD computer containing two or more CPUs that cooperate on common computational tasks. Multiprocessors are distinguished from multicomputers and computer networks, which are systems with multiple CPUs operating largely independently on separate tasks. The various processors making up a multiprocessor typically share resources such as communication facilities, IO devices, program libraries and databases and are controlled by a common operating system.

Motivation. The main reasons for including multiple CPUs in a computer system are to improve performance and reliability. Performance is improved either by distributing the computation of a large task among several CPUs or by performing many small tasks in parallel using separate CPUs. A multiprocessor with n identical processors can, in principle, provide n times the performance of a comparable SISD system or *uniprocessor*. A major goal, therefore, in designing an n-CPU multiprocessor is to achieve a speedup $S(n)$ as close to n as possible. By enabling such resources as secondary memory to be shared, a multiprocessor can reduce overall system costs. Many multiprocessors also have the advantage of *scalability;* that is, the system size can be increased incrementally by adding processors to meet growing computation needs. Scalability is facilitated by making all CPUs identical and allowing each to execute either operating system (kernel) or user code; multiprocessors with these properties are said to be *symmetric.* Finally, system reliability is improved by the fact that the failure of one CPU need not cause the entire system to fail. The functions of the faulty CPU can be taken over by the other CPUs; consequently, multiprocessors enable fault tolerance to be incorporated into the system.

As discussed earlier, multiprocessors are classified by the organization of their memory systems (distributed memory and shared memory) and by their interconnection networks (dynamic or static). Shared-memory and distributed-memory multiprocessors are sometimes referred to as *tightly coupled* and *loosely coupled,* respectively, reflecting the speed and ease with which they can interact on common tasks. Multiprocessors are also classified by the number of processors they contain: *Massively parallel* machines can contain thousands of processors. Most multiprocessors, however, are *modestly parallel,* containing from 2 to about 30 processors; such multiprocessors have existed since the 1960s. The relative success of multiprocessors with a few CPUs stems from the difficulty of programming large numbers of CPUs to cooperate efficiently. The lack of standard, widely used languages and application packages for parallel programming has been a major obstacle to wider use of multiprocessors.

Shared-bus systems. Most commercial multiprocessors have been built around a single shared system bus B because of B's relative simplicity and low cost. The CPUs, memory, and IO units are attached directly to B and time-share its communication facilities. Only one pair of units can use B at a time, either for CPU-memory or IO-memory communication. The memory units and IO devices on B are global to all the processors; hence single-bus multiprocessors are of the shared-memory class. If the access time to the shared memory is the same for each processor, the multiprocessor is said to be of the *uniform-memory access (UMA)* type.

The global bus B is clearly a communication bottleneck in shared-bus multiprocessors, leading to contention and delay whenever two or more units request access to main memory. In practice, memory contention limits to about 30 the number of CPUs that can be included in the system without an unacceptable degradation in performance. Figure 7.63 shows that a single-bus multiprocessor's performance can be improved by supplying each CPU with a local bus. The local bus is connected to a local memory unit that contains part of the shared address space; it can also support a local IO subsystem, as illustrated in Figure 7.63. This system configuration removes most of the routine memory traffic from B so that it can be

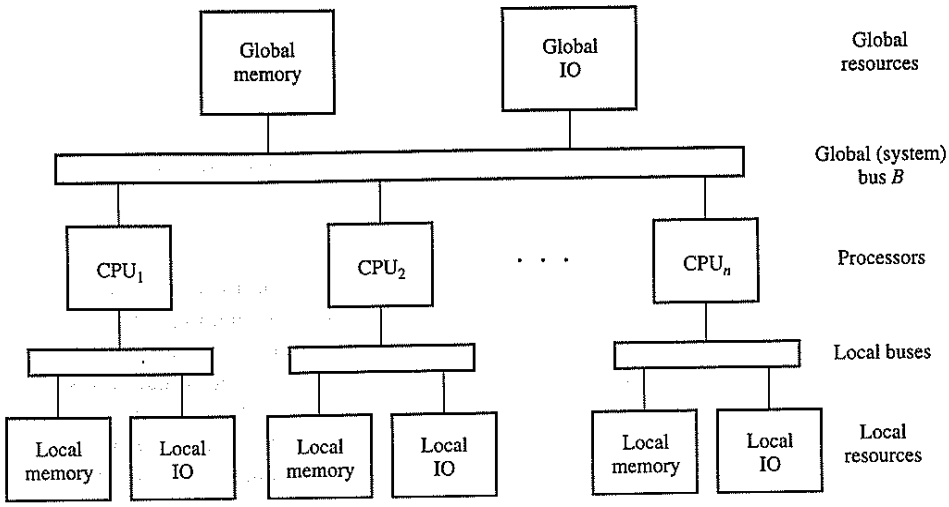


Figure 7.63
Shared-bus multiprocessor with global and local resources.

reserved primarily for interprocessor communication. Many microprocessor families can be configured as multiprocessors in this way. The Intel Pentium, for example, was designed for use in shared-bus multiprocessors, with standard buses like the PCI bus serving as local buses.

Despite its relative simplicity, the shared-bus architecture exhibits some of the basic synchronization problems common to all multiprocessors. Consider the situation in which two CPUs share a region R of global memory where mutual exclusion (section 7.2.3) applies; that is, only one processor should have access to the shared region at a time. Access to R is conveniently controlled by a semaphore (flag) F that indicates whether R is currently being used by some other process ($F = 1$) or is available for use by a new process ($F = 0$). Before it attempts to access R, a CPU first reads F, which must be stored in global memory. If $F = 0$, the CPU then changes F to 1 and proceeds to use R. If it finds that F is already 1, then it does not attempt to use R. The mutual exclusion requirement can be violated if it is possible for two CPUs to independently access the semaphore at the same time and find $F = 0$. This violation can occur if a second processor CPU_2 can read F after the first processor CPU_1 has read it, but before CPU_1 has changed F to 1. The problem lies in the fact that semaphore flag test-and-set instructions issued by the CPUs can be broken down into interleaved bus cycles as follows:

Global bus cycle	Action
i	CPU_1 fetches semaphore $F = 0$.
$i+1$	CPU_2 fetches semaphore $F = 0$.
$i+2$	CPU_1 sets F to 1.
$i+3$	CPU_2 sets F to 1.

At time $i + 4$, both CPU_1 and CPU_2 assume they have exclusive control over the critical region R, with potentially catastrophic consequences. A solution to this

problem, which is discussed in section 7.3.1, is to allow semaphore test-and-set instructions to have exclusive control of the system bus while they are being executed. Such instructions lock the bus until their execution is complete, thereby delaying any test-and-set instructions awaiting execution by other CPUs until the first CPU has safely set the semaphore to busy.

EXAMPLE 7.8 THE SEQUENT SYMMETRY SHARED-BUS MULTIPROCESSOR [SEQUENT 1996]. The Symmetry multiprocessor series is built around a high-speed shared bus, multiple CPUs from the Intel 80X86/Pentium series, and the UNIX operating system. Symmetry multiprocessors can be variously characterized as MIMD, shared memory, tightly coupled, scalable, symmetric, and UMA. They are typically used in applications such as on-line transaction processing characterized by heavy computation requirements and a need for high reliability.

The Symmetry 5000 system introduced in 1995 has the general organization depicted in Figure 7.64. It contains from 2 to 30 Pentium CPUs each with a 2MB cache; there are no other local memories. The CPUs are packaged two per circuit board; the system can be expanded to the maximum allowed by adding CPU boards. The main-memory system is also packaged in circuit boards that facilitate modular expansion. The memory is interleaved (section 6.1.2) to increase performance, and an error-correcting code improves reliability. The IO subsystem includes one or more high-speed IO processors designed to communicate with magnetic disk and tape memories via high-speed SCSI buses. Additional, slower IO controllers support other IO devices, as well as various standard external interfaces and communication protocols such as Ethernet.

A key component of the Symmetry 5000 is its proprietary system bus that links all processors, memory units, and IO controllers. This Highly Scalable Bus (HSB) contains a 64-bit data-address bus designed to transmit (in multiplexed mode) 64-bit

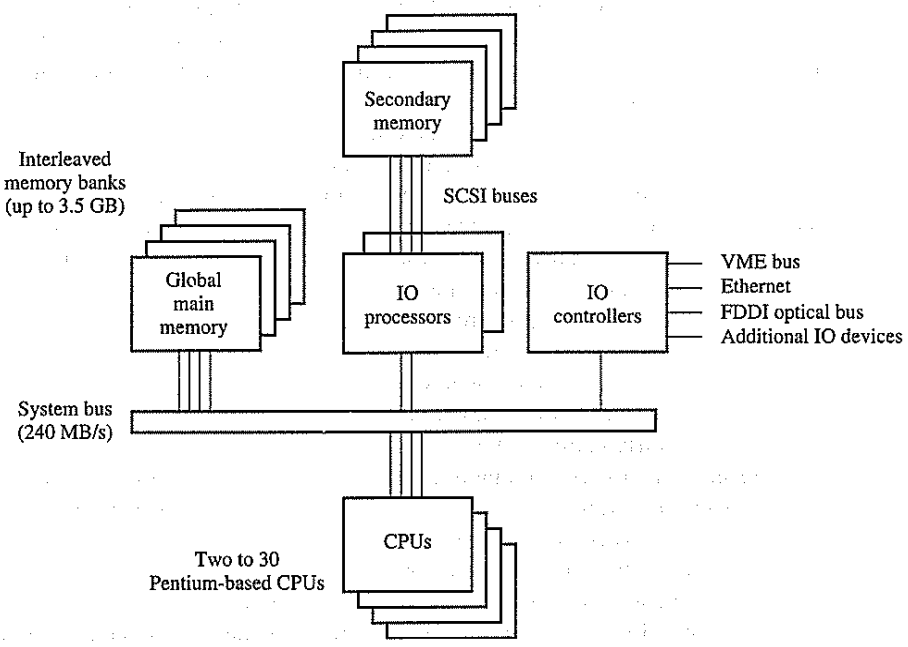


Figure 7.64
Organization of the Sequent Symmetry 5000 multiprocessor.

data words and 32-bit addresses. It has an unusual "pipelined" data transmission mode that supports a simplified form of package switching, which allows memory and IO data to be transmitted in bursts whose transmission can be overlapped. The HSB was designed for a maximum data bandwidth of 240 MB/s.

The Symmetry's operating system, DYNIX, is a version of UNIX with enhancements to support multiprocessing. Each CPU acts like a uniprocessor that is executing independently under UNIX supervision; it executes processes from a list that all the CPUs share. Interrupt signals are generated at periodic intervals to force the CPUs to examine the list of waiting processes and schedule a high-priority process for execution. This approach forces all CPUs to share the system's workload. To avoid conflicts among CPUs when executing kernel routines stored in the shared memory, a semaphore mechanism of the kind discussed earlier enforces mutual exclusion.

Cache coherence. In shared-bus multiprocessors like the Symmetry, caches play a vital role in reducing the contention for the shared system bus. Without caches, connecting more than two or three CPUs to the same bus might be impractical. Typically, each CPU has a private one- or two-level cache, which forms a local memory and allows the CPU to access data and instructions without using the system bus. With an independent cache in each CPU, the possibility exists for two or more caches to contain different (inconsistent) versions of the same information at the same time; this is the *cache-coherence problem.* This problem is alleviated, but not solved, by using write-through, which, as discussed in section 6.3, causes both the cache and main (global) memory to be updated whenever a memory write operation occurs. Suppose, for example, that one CPU updates variable X in both its cache and the global memory. If another CPU then changes X, the new value of X will be written into main memory, but the two caches will contain different values for X. Subsequent reads from these caches can lead to inconsistent results. Thus to ensure coherence we need a mechanism that informs each cache about changes to shared information stored in other caches.

We can solve the cache-coherence problem with either hardware or software. One software-based solution is to mark (tag) information during program compilation as either cacheable or noncacheable. All writable shared items are marked as noncacheable, meaning they can be accessed directly only from main memory. A write-through policy that requires a processor to mark a shared cache item X as invalid, or to be deallocated, whenever the processor writes into X can then ensure cache coherence. When the processor references X again, it is forced to bypass the cache and access main memory, thereby always acquiring the most recent version of X. This approach can significantly degrade system performance, however. Invalidation also forces the removal of needed data from the cache, thus increasing its miss ratio, which, in turn, increases the main-memory traffic.

Hardware-based methods of maintaining cache coherence offer the advantages of higher speed and program transparency, but they tend to be expensive. One possible approach is for a processor to broadcast its write operations to all caches and the global memory via the shared bus. Every cache controller in the system then examines its assigned addresses to see if the broadcast item is presently allocated to it. If it is, the cache block (line) in question is either updated or marked as dirty (modified). The drawback of this technique is that every cache write forces all caches to check the broadcast data, making the caches unavailable for normal processing.

A related, but less costly, hardware-based method known as *cache snooping* equips each CPU with circuitry to continuously monitor or "snoop" on system-bus activity in order to detect references by other processors to memory addresses currently in its cache. The CPU can also signal other CPUs that it has a copy of the referenced item and, when necessary, modify or delay the other CPUs' main-memory accesses. If CPU_2 attempts to read (write) memory data with an address that is currently assigned to CPU_1's cache, CPU_1 detects this attempt in what is called a *snoop read (write) hit* by CPU_1. On making a snoop hit, CPU_1 determines whether actual or potential incoherence exists and then takes appropriate steps to eliminate it. The following courses of action are typical:

- Suppose that CPU_1 makes a snoop read hit when its cache copy of the requested item is dirty and it has not yet updated main memory—this situation can occur only when the write-back policy is used. CPU_1 signals CPU_2 to suspend its read request while CPU_1 updates main memory by writing back the block containing the requested word. Then CPU_1 signals CPU_2 to complete its memory read operation.
- If CPU_1 makes a snoop write hit, it knows that its own cache copy of the requested item is about to become dirty. It therefore marks that copy as dirty. Hence the next time CPU_1 tries to read the item in question, a cache miss occurs that forces CPU_1 to read a valid copy from main memory.

An alternative response to a snoop write hit by CPU_1 is for CPU_1 to capture the new data on the system bus as CPU_2 writes it to global memory. CPU_1 can then use the captured data to update its cache.

EXAMPLE 7.9 THE MESI CACHE COHERENCY PROTOCOL [MOTOROLA 1994; ANDERSON AND SHANLEY 1995]. To maintain consistency in a multiprocessor, or in a uniprocessor with independent IO processors, a cache controller must keep careful track of the state of each cache block (line) under its control. It does so by attaching a few state bits to every block stored in the cache data memory and processing the states according to some coherence algorithm or *protocol*, as it is often called. Microprocessors such as the Pentium and some PowerPC models employ a standard cache coherence protocol based on the following four states:

- *M* (*modified*): The block has been modified or "dirtied" by a recent write hit to the cache.
- *E* (*exclusive*): The block is "clean," that is, the same as the copy in main memory, and no other processor has a copy.
- *S* (*shared*): The block is clean, but other processors may have a copy.
- *I* (*invalid*): The data in the block is not valid.

A cache-control algorithm using these states is known as the *MESI coherence protocol* for obvious reasons. Figure 7.65 gives a slightly simplified version of the MESI protocol, which shows how the states of a cache block change in response to various read and write conditions, assuming that a write-back policy and a cache-snooping mechanism are used. We also assume a one-level cache, although this protocol works equally well with multiple cache levels.

First consider the effect of read operations on the state of a cache block. Read hits to the block leave its state unchanged. Read misses, however, are not so simple. When a processor P_1 first tries to read the (empty) cache, the cache controller changes all block states to *I* (invalid) and forwards the read request to main memory. Thus *I* acts like a reset state that triggers a block transfer to the cache; the incoming block's state is

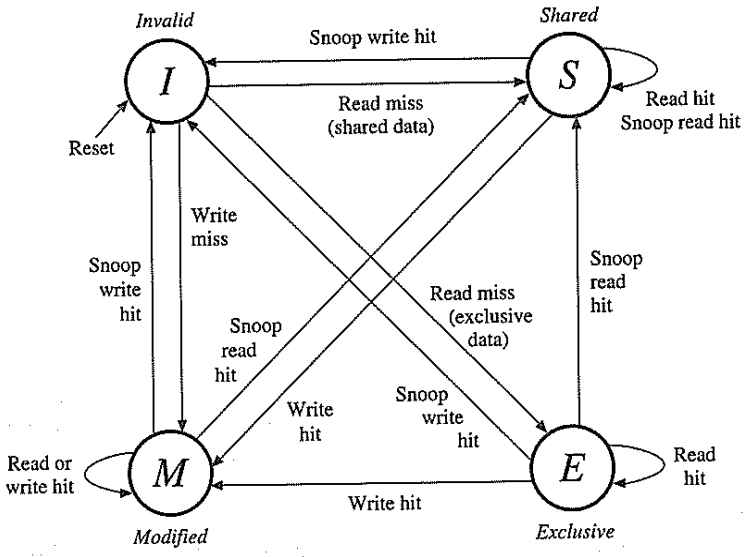


Figure 7.65
State-transition graph (simplified) for a cache block using the MESI coherence protocol.

set to *E* (exclusive) if no other processor has a copy of the same block. (An initial write also brings into the cache a block whose state is marked *E*.) If during P_1's read operation, a snooping processor P_2 signals via the shared bus that its cache has a clean copy of the same block, in which case no incoherence exists, the state of the block in P_1's cache is set to *S* (shared) instead of *E*. If, on the other hand, P_2 signals that its cache has a dirty (modified) copy of the same block, the caches are no longer coherent. To resolve this incoherence, the signal from P_2 causes P_1 to postpone its memory read and to relinquish the system bus. P_2 then assumes the role of bus master and writes its modified block back to main memory. P_2 also changes the state of its copy of the cache block from *E* to *S* because it now knows that the block in question is shared. This state change is specified by the transition from *E* to *S* marked "snoop read hit" on the right side of Figure 7.65. Finally, the first processor P_1 repeats its main-memory read request and obtains a clean copy of the block, which it marks as *S*.

Now consider the cache block's state when P_1 addresses a write hit to it. If the target block is in either of the clean states *S* or *E*, the block's state changes to *M* (modified or dirty). In the *S* case P_1 signals the other processors that it is writing to a shared block; they respond by marking their copies of the shared block *I* (invalid). The modified cache block remains in the *M* state in P_1 during subsequent reads and writes to it, unless P_1's own snooping detects read or write hits addressed to the same block in other caches.

A write miss by P_1 triggers a memory read operation that replaces the target block in the cache, where it is eventually marked *M*. If some other processor P_2 has a clean (*S* or *E*) copy of the same block, P_2 changes the state of its copy to *I*. If P_2 has a dirty (*M*) copy of the block in question, P_2 sends a signal to this effect to P_1, causing the latter to delay its memory read. P_2 then takes control of the system bus and writes its modified block to main memory; P_2 also changes the state of its cache copy from *I*, since it knows that the copy of the shared block in main memory is about to be changed by P_1. Control of the bus is then returned to P_1, which completes its block transfer.

Message-passing computers. As developments in VLSI technology during the 1980s ushered in powerful one-chip microprocessors and memory (RAM) chips with capacities in the multimegabit range, it has become feasible to build massively parallel multiprocessors, with hundreds or thousands of processors. Multiprocessor architectures with distributed memory systems, where interprocessor communication is by message-passing, avoid most of the contention problems inherent in the use of single shared memories and buses. Such computers can provide extremely high performance, but they also pose problems in algorithm and program design that are far from being satisfactorily solved.

Various static and dynamic interconnection structures have been proposed for massively parallel multiprocessors. Static structures like hypercubes and trees are easier to build and control when many processors are involved. Dedicated buses or IO communication lines typically serve as interprocessor links. Neighboring processors can then interact at the maximum possible rate, with little interference from other processors. Interconnection networks are selected to trade hardware cost for communication speed in some class of applications. The hypercube structure achieves a good balance between these parameters. Consequently, it has been used in several commercial computers of the massively parallel type [Hayes and Mudge 1989].

An n-dimensional hypercube computer is characterized by the presence of 2^n nodes, each consisting of a processor and its local memory. Each processor P_i has direct links to n other processors (its *neighbors*); these links form the edges of the hypercube. A set of 2^n distinct n-bit binary addresses can be assigned to the processors in such a way that P_i's address differs from each of its neighbors in exactly 1 bit; Figure 7.60*c* illustrates hypercube addressing for $n = 3$. Hypercubes have several attractive features:

- A hypercube can be expanded or scaled up while maintaining a good balance between the number of nodes and the cost of internode communication. As n is incremented by one, the number of nodes doubles, but the node degree and the maximum internode distance both increase only by one (from n to $n + 1$).
- A hypercube is homogeneous in that the system appears the same when viewed from any of its nodes. This feature simplifies programming because all nodes can execute the same programs on different data when collaborating on a common task.
- We can embed other useful interconnection structures, such as rings and meshes, efficiently in the hypercube. We say that (graph) G is *embeddable* in H if and only if every node in G can be mapped into a distinct node in H such that all nodes that are neighbors in G are also neighbors in H. In other words, G is embeddable in H if we can find an exact (isomorphic) copy of G inside H.
- A large hypercube can support multiple concurrent users with each user program assigned to a private embedded hypercube or *subcube* that is disjoint from other users' subcubes. For example, in a four-dimensional hypercube (Figure 7.66*b*), four-node subcubes can be assigned to two users, and an eight-node subcube can be assigned to a third user.

Embeddability can be used to compare different interconnection structures for multiprocessors. Let C_1 with (static) interconnection network N_1 and C_2 with interconnection network N_2 be computers employing similar processors. If N_1 is

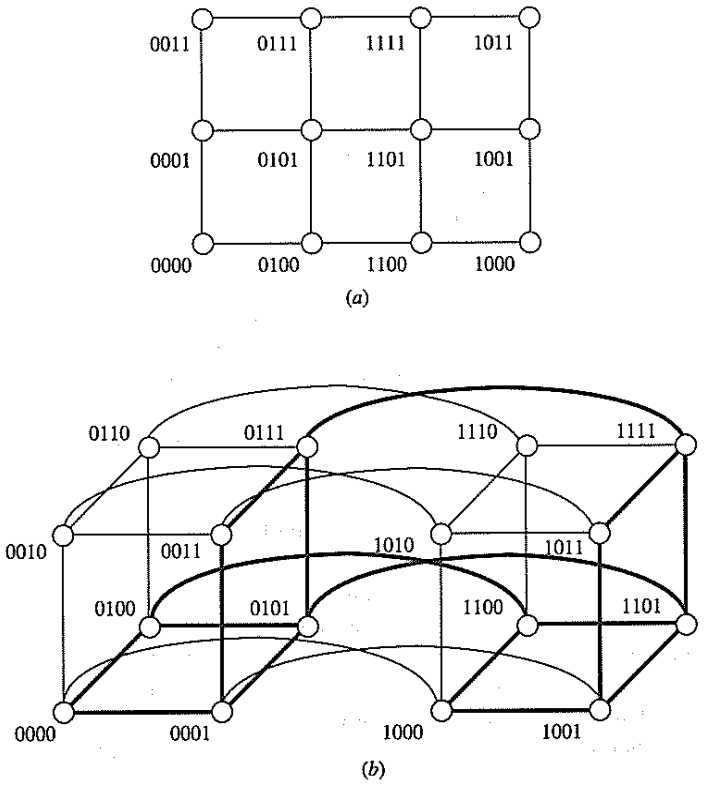


Figure 7.66
(*a*) A 3 × 4 mesh and (*b*) embedding the mesh in a four-dimensional hypercube.

embeddable in a sufficiently large version of N_2, then C_2 will be able to *embed* C_1. Therefore, any structure embeddable in C_1 is also embeddable in C_2, and C_2 is at least as powerful as C_1 from a static structural viewpoint. Referring to Figure 7.11, it is obvious that any k-node system can be embedded in a system of k or more nodes with the structure of a complete graph (Figure 7.11*f*). A sufficiently big mesh-structured system can embed any path or ring. It cannot, however, embed a hypercube, since for $n > 4$, every node of an n-dimensional hypercube has greater degree than every node of the mesh. A hypercube can embed both the ring and the star structures. Less obvious is the fact that a mesh can be embedded in a hypercube. An embedding of the 12-node 3 × 4 mesh into the 16-node four-dimensional hypercube appears in Figure 7.66. Heavy lines show the nodes and edges of the hypercube that correspond to those of the mesh.

EXAMPLE 7.10 THE nCUBE HYPERCUBE MULTIPROCESSOR [HAYES AND MUDGE 1989; nCUBE 1990]. Hypercube multiprocessors were proposed as early as 1962 at the University of Michigan, but the first working machine was not demonstrated until the completion of the six-dimensional (64-node) Cosmic Cube computer at Caltech in 1983. Influenced by this work, several commercial hypercube computers

were introduced in the mid-1980s, including Intel's iPSC series and the nCUBE (then written NCUBE) series developed by nCUBE Corp. The original nCUBE 1 family included hypercubes of various sizes up to a 10-dimensional (1024 node) machine. Subsequent nCUBE computers increased the number of nodes to $8192 = 2^{13}$.

An nCUBE processor node is equipped with a set of high-speed IO channels, each consisting of a serial input line and a serial output line. One channel connects to a host or front-end computer; the remaining channels connect the node to its neighbors in the hypercube. Processor-to-processor communication is implemented by transmitting messages between buffer areas in the local memories of communicating nodes. Each interprocessor link has both an address register pointing to its message buffer area and a count register indicating the number of bytes to be sent or received. Once a processor initiates a message transfer, the processor can continue with other tasks while the interprocessor message transfer proceeds as a DMA operation between the memories of the communicating nodes. A broadcasting instruction is also supported that allows the same data to be transmitted to all processors in the hypercube; see problem 7.38.

First we consider interprocessor communication in an nCUBE 1 system. Assume that an n-dimensional subcube is assigned to the user and that the message source and destination nodes have the binary addresses $S = s_{n-1}\ldots s_1 s_0$ and $D = d_{n-1}\ldots d_1 d_0$, respectively. The EXCLUSIVE-OR function $R = S \oplus D = r_{n-1}\ldots r_1 r_0$, where $r_i = s_i \oplus d_i$ for $i = 0,1,\ldots,n-1$, controls the routing process. The values of i for which $r_i = 1$ indicate the dimensions of the hypercube to be traversed by a message en route from source to destination. The operating system kernel residing in each node that receives the message reads the destination address D (a field in the message header); computes $R = P \oplus D$, where P is the address of the current node; and scans R from left to right until it encounters some $r_j = 1$. Node P then forwards the message to the neighboring node P' whose address differs from P's in the jth bit. If $R = 0$, then $P = D$ and P recognizes itself as the destination node and proceeds to process the message. Thus in a six-node subcube of the nCUBE 1, a message being sent from node 7 to node 45 passes through nodes with the following sequence of addresses:

$$S = 000111 \rightarrow 100111 \rightarrow 101111 \rightarrow 101101 = D$$

This *store-and-forward* routing method sends each message along a shortest path so that the minimum number of intermediate nodes relay messages between the source and destination. In the nCUBE 2 computer, each node P contains a high-speed message-routing unit that allows messages for other units to pass though P without affecting P's ongoing operations; this approach largely eliminates the need to temporarily store messages in intermediate nodes.

A node of the nCUBE 2 consists of a full-custom 64-bit CPU on a single IC, plus a six-chip local memory. The CPU's architecture resembles that of the Digital VAX family; it has a CISC-style instruction set with fixed-point and floating-point arithmetic instructions and all the logic necessary for memory management and IO control. Its speedup features include a four-stage instruction pipeline, an I-cache and a D-cache, as well as the special message router noted already. The local memory size can range up to 64 MB per node, so an 8192-node system can have a distributed memory of 256 GB. With a modest clock rate of 20 MHz, each processor delivers about 2.4 MFLOPS (assuming 64-bit operations), implying a peak performance of around $2.4 \times 8192 = 19.7$ GFLOPS, so the nCUBE 2 was classed as a massively parallel "supercomputer."

The structure of an nCUBE 2 system is outlined in Figure 7.67. The hypercube array of processors H is packaged into printed-circuit boards, each of which contains a 64-node hypercube forming a subcube of H. Each processor has 14 communication channels, one of which connects to an IO subsystem, such as a "farm" of IO disks forming the system's secondary memory. Many IO channels to the hypercube array enable a large number of peripherals to operate in parallel to satisfy the nCUBE's

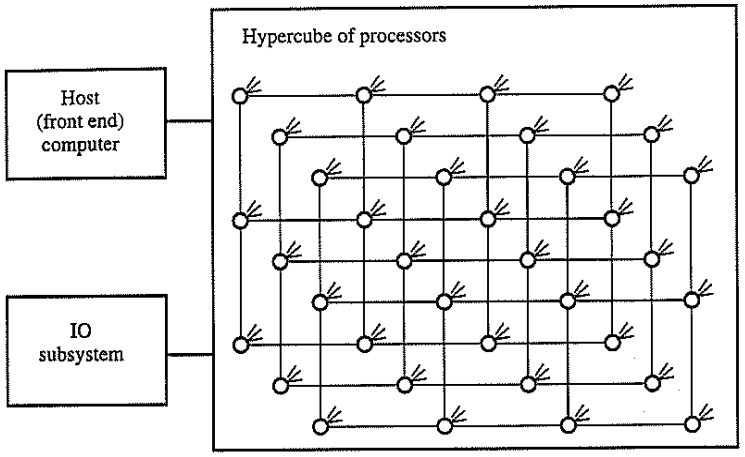


Figure 7.67
Organization of the nCUBE 2 hypercube multiprocessor.

massive computation ability. Each channel is controlled by the nCUBE processor used in the hypercube array. Disk storage capacity can exceed a terabyte (2^{40} bytes), making the nCUBE well suited to the management of very large databases.

The nCUBE operating system provides all the usual UNIX system management and programmer support functions (see Example 7.8). It treats a hypercube of processors as a device, which in the UNIX philosophy is a special type of file. Consequently, a hypercube of any size can be opened, closed, written into, and read from like any other UNIX file. This feature permits the operating system to allocate independent subcubes to different users so that one or two large applications or many small applications can share the processor hypercube concurrently.

Multistage interconnection networks. Dynamic interconnection networks for multiprocessors can be constructed from two-state switching elements of the kind depicted in Figure 7.68. Each switch S has a pair of input data buses X_1,X_2; a pair of output data buses Z_1,Z_2; and some control logic (not shown). All four buses are identical and can function as processor-processor or processor-memory links. S has two states determined by the control line c: a *through* or *direct* state T, as illustrated in Figure 7.68*b* where $Z_1 = X_1$ (Z_1 is connected to X_1) and $Z_2 = X_2$, and a *cross* state X where $Z_1 = X_2$ and $Z_2 = X_1$ (Figure 7.68*c*).

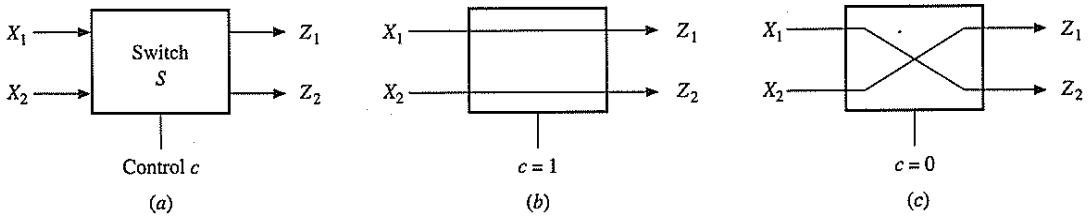


Figure 7.68
(*a*) Switching element; (*b*) through state T; (*c*) cross state X.

By using S as a building block, *multistage interconnection networks* (*MINs*) can be constructed for use in massively parallel computers [Siegel 1990]. Figure 7.69 shows a small MIN that has 12 switching elements arranged into three stages (columns) and is intended to provide dynamic connections among eight processors denoted P_{000}:P_{111}. By setting the control signals of the switching elements in various ways, many different interconnection patterns are possible. The processor-to-processor connections that are possible depend on the number of stages, the fixed connections linking the stages, and the settings of the switching elements. The particular MIN in Figure 7.69 is called an 8×8 *omega* network. A large version of this MIN was used in the experimental Cedar multiprocessor designed at the University of Illinois in the 1980s. We now examine the major characteristics of some typical MINs, concentrating on those designed for processor-to-processor communication.

An $N \times N$ MIN SN provides a flexible set of communication links between N processors, which are the sources and destinations of SN. Since the processors are identified by n-bit binary addresses, it is convenient to make $N = 2^n$. The processor-pairs that are connected to each other at any time by SN are determined by the states of the switching elements, each of which can be in either the through (T) or cross (X) state. Control logic associated with the MIN sets the switch states dynamically to satisfy interconnection requests from the processors. A particular MIN state is retained long enough to allow at least one package to be transferred through the network. The state then changes to match the source-destination requirements of the next set of packages, and so on. We assume that a processor can buffer or queue its outgoing packages until the MIN is ready to transfer them. The processors accept incoming packages as soon as they arrive.

A fundamental requirement of a MIN is that it be possible to connect every processor P_i to every other processor P_j using at least one configuration of the network; this feature is termed the *full-access* property. It is easy to show that the omega network of Figure 7.69 is a full-access network. Figure 7.70 shows the

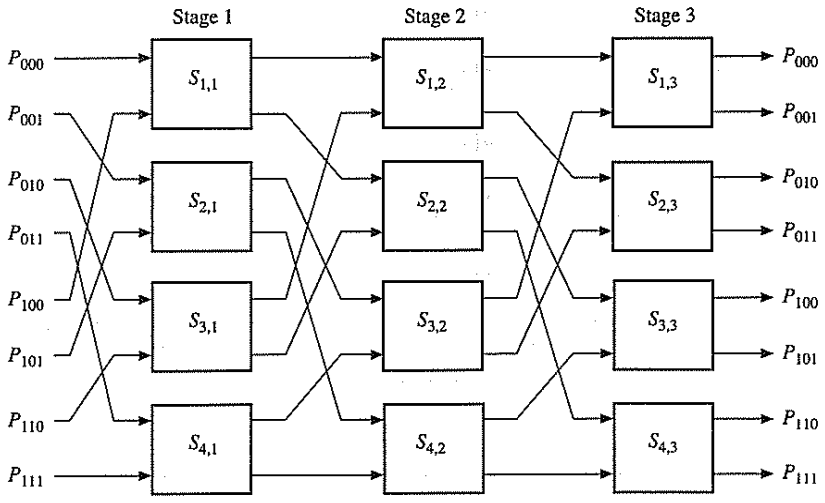


Figure 7.69
Three-stage 8×8 omega multistage interconnection network (MIN).

Destination	Stage 1	Stage 2	Stage 3
P_{001}	$S_{1,1} = T$	$S_{1,2} = T$	$S_{1,3} = X$
P_{010}	$S_{1,1} = T$	$S_{1,2} = X$	$S_{2,3} = T$
P_{011}	$S_{1,1} = T$	$S_{1,2} = X$	$S_{2,3} = X$
P_{100}	$S_{1,1} = X$	$S_{2,2} = T$	$S_{3,3} = T$
P_{101}	$S_{1,1} = X$	$S_{2,2} = T$	$S_{3,3} = X$
P_{110}	$S_{1,1} = X$	$S_{2,2} = X$	$S_{4,3} = T$
P_{111}	$S_{1,1} = X$	$S_{2,2} = X$	$S_{4,3} = X$

Figure 7.70
Switch settings of the three-stage omega network of Figure 7.69 to connect P_{000} to each of the other processors.

seven unique switch configurations needed to connect P_{000} to each of the other processors; here $S_{i,j}$ = T (X) indicates that switch i of stage j is set to the through (cross) state. A complete network configuration in which P_{000} is connected to P_{001} appears in Figure 7.71. In this state the network also connects P_{010}, P_{100}, and P_{110} to P_{011}, P_{101}, and P_{111}, respectively, thus providing simultaneous communication among four processor-pairs. Reducing the number of stages from three to two eliminates the full-access property.

Another useful property of a MIN is the ability to establish a connection between any pair of processors that are not using the network, without altering the switch settings already established to link other processors; this is the *nonblocking* property. The three-stage omega MIN of Figure 7.69 does not have this property and is therefore a *blocking* network. For example, suppose that P_{000} is already connected to P_{001}; this condition requires the top row of switches to be set to T T X, as specified in Figures 7.70 and 7.71. It is now impossible to connect P_{100} either to P_{010} or to P_{011}. The preexisting setting of $S_{1,1}$ creates a path from P_{100} through stage

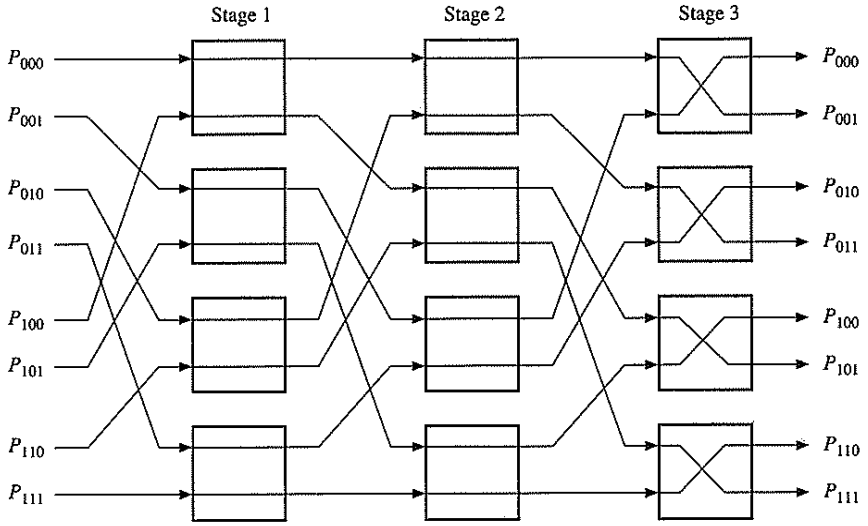


Figure 7.71
One state of the three-stage omega network.

1 to $S_{2,2}$. No links exist from $S_{2,2}$ to $S_{2,3}$, the third-stage switching element connected to P_{010} and P_{011}; hence $S_{2,2}$ cannot be set to forward data to P_{010} or P_{011}. This type of blocking causes communication delays similar to those occurring in a single-bus system when several processors attempt to use the system bus simultaneously. Nonblocking MINs require an excessive number of switches for most computer applications. An $N \times N$ crossbar switch is an example of a nonblocking network because it allows any idle row to be connected to any idle column. However, it contains N^2 complex crosspoint switches, whereas an $N \times N$ omega network contains only $(N/2) \log_2 N$ simpler 2×2 switches.

A few basic interstage wiring patterns characterize the most common MIN types proposed for multiprocessors. Each such pattern is a mapping ψ from a set of sources $\{S_i\}$ to a set of destinations $\{D_{\psi(i)}\}$ for $i = 0,1,\ldots,N-1$. Here S_i is the address of an output port of a processor or switching element, and $D_{\psi(i)}$ is the address of the input port to which S_i is wired. The *shuffle* pattern is defined by the following mapping:

$$\sigma(i) = 2i + \lfloor (2i)/N \rfloor \qquad \text{(modulo } N\text{)} \tag{7.18}$$

Here σ is the shuffle function illustrated by Figure 7.72*a* for $N = 8$. The name *shuffle* comes from the fact that the destination addresses 0, 1, 2, 3, 4, 5, 6, 7 can be mapped into (connected to) the source addresses 0, 4, 1, 5, 2, 6, 3, 7 by interleaving the first half 0, 1, 2, 3 of the address sequence with the second half 4, 5, 6, 7 in the manner of a perfectly shuffled deck of cards. Let each address i be represented by the corresponding n-bit binary number $b_{n-1}b_{n-2}\ldots b_0$. An equivalent definition to (7.18) is

$$\sigma(i) = b_{n-2}b_{n-3}\cdots b_0 b_{n-1} \tag{7.19}$$

indicating that the shuffle function corresponds to rotating the source address 1 bit to the left to determine the destination address. By following a shuffle connection with $N/2$ switching elements, each of which can exchange (cross) a pair of buses, we obtain the single-stage *shuffle-exchange* network, shown in Figure 7.72*b* for the case $N = 8$. The omega network of Figure 7.69 is built from $n = \log_2 N$ shuffle-exchange stages.

Another useful class of MINs is based on the *butterfly* connection depicted in Figure 7.73*a*. The 4×4 single-stage butterfly network appears in Figure 7.73*b*; note that the butterfly connection is placed after, rather than before, the $N/2$ switching elements. Consider an $N \times N$ multistage network with n stages $1, 2, \ldots, n$ and N port addresses $i = 0, 1, \ldots, N-1$, where, as before, $i = b_{n-1}b_{n-2}\cdots b_0$. The kth *butterfly function* β_k is defined as follows for $k = 1, 2, \ldots, n-1$:

$$\beta_k(b_{n-1}\cdots b_{k+1}b_k b_{k-1}\cdots b_1 b_0) = b_{n-1}\cdots b_{k+1}b_0 b_{k-1}\cdots b_1 b_k$$

Thus β_k interchanges bits 0 and k of the source address to obtain the destination address. For example, when $k = 1$ and $N = 4$, we obtain

$$\beta_1(00) = 00$$
$$\beta_1(01) = 10$$
$$\beta_1(10) = 01$$
$$\beta_1(11) = 11$$

corresponding to the interconnection pattern on Figure 7.73*a*.

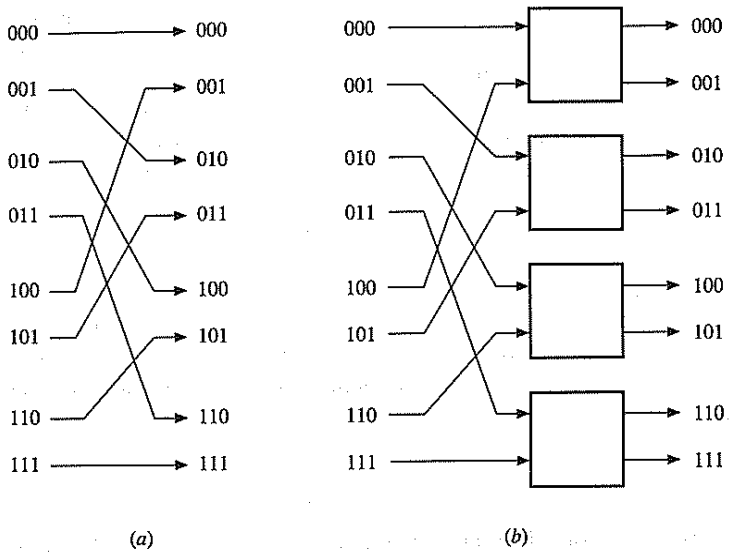


Figure 7.72
(*a*) Shuffle connection for $N = 8$ and (*b*) single-stage shuffle-exchange network.

The connection pattern defined by

$$\sigma^{-1}(i) = b_0 b_{n-2} b_{n-3} \cdots b_1 \tag{7.20}$$

is called the *inverse shuffle function* σ^{-1}. Equation (7.20) is the same as Equation (7.19), defining the shuffle function σ with the direction of the address bit rotation reversed. Figure 7.74 shows a 16×16 version of a MIN called the *indirect hypercube network*, which in the $N \times N$ case consists of $\log_2 N$ stages of $N/2$ switching elements; the wiring patterns following the stages are defined by $\beta_1, \beta_2, \ldots, \beta_{n-1}, \sigma^{-1}$. This MIN's name comes from the fact that it can easily simulate the connections of a static hypercube interconnection network; see problem 7.39.

Indirect hypercube and shuffle-exchange MINs have similar properties. Suppose that the directions of all the arrows in an $N \times N$ shuffle-exchange network are reversed, implying that the shuffle connection σ in each stage is replaced by σ^{-1}. The resulting $N \times N$ *inverse omega network* and the $N \times N$ indirect hypercube network are essentially the same MIN drawn in different ways. Consequently, for

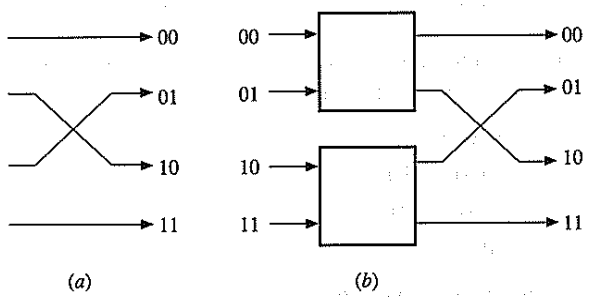


Figure 7.73
(*a*) Butterfly connection for $N = 4$ and (*b*) single-stage butterfly network.

each state of the indirect hypercube network, there is a state of the inverse omega network that connects the N processors in exactly the same way, and vice versa. This equivalence is not obvious and explains the many names under which this class of MINs appears in the literature (inverse omega, indirect binary n-cube, butterfly, and so forth).

Since an address contains $n = \log_2 N$ bits, at least $n = \log_2 N$ stages must be present for an $N \times N$ MIN to have the full-access property. With this number of stages, it is also easy to determine the switch settings needed to connect an arbitrary pair of processors, since each stage controls 1 bit (dimension) of the address space. We illustrate this for the indirect hypercube MIN of Figure 7.74. Suppose that a source processor with binary address $S = s_{n-1}s_{n-2}\cdots s_0$ is to be connected to a destination processor with address $D = d_{n-1}d_{n-2}\cdots d_0$. As in the static hypercube

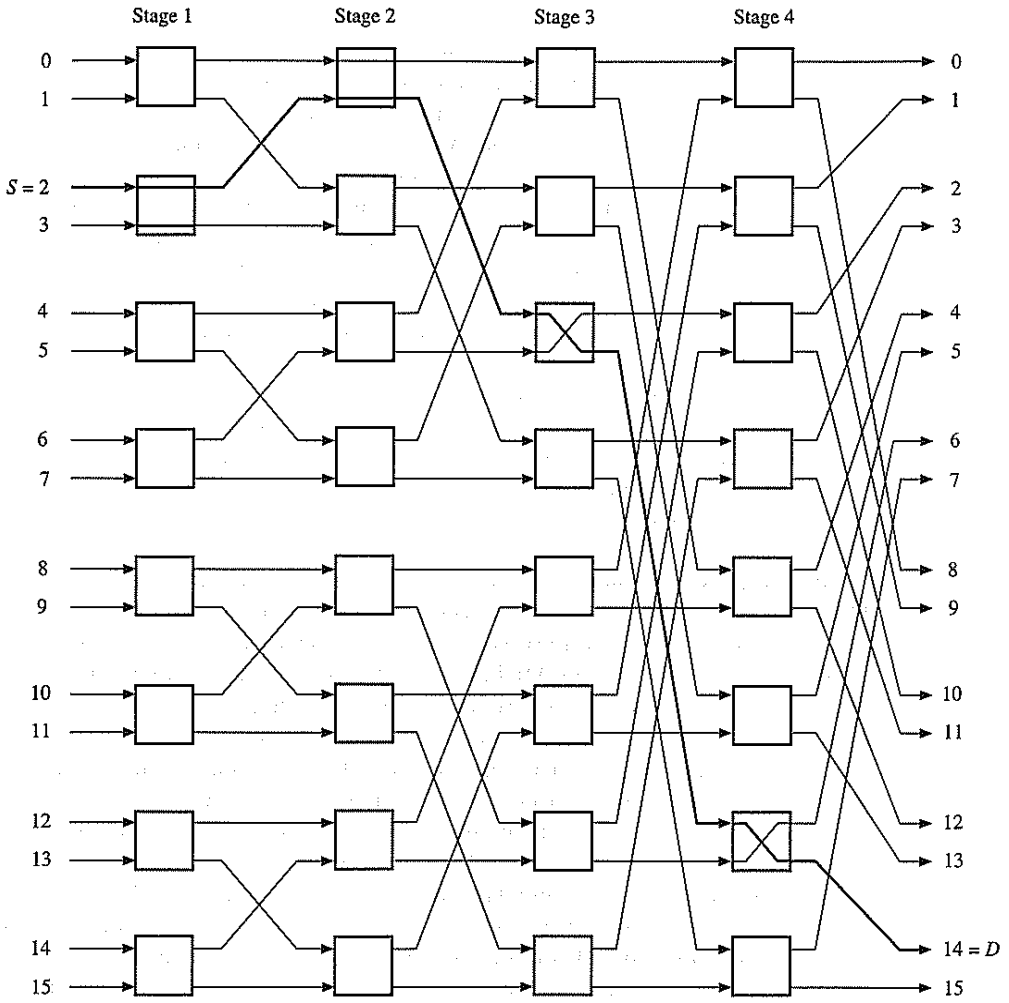


Figure 7.74
16 × 16 indirect hypercube network.

routing algorithm (Example 7.10), we compute $R = S \oplus D = r_{n-1}r_{n-2}\cdots r_0$, and use R to control the MIN's switch settings. If $r_i = 0$, then all the switches in stage $i + 1$ (assuming again that the stages are numbered 1, 2, ..., n) are set to the through (T) state; these switches are set to the cross (X) state if $r_i = 1$. For example, Figure 7.74 shows the switch settings to connect source $S = 2$ to destination $D = 14$. In this case $R = 0010 \oplus 1110 = 1100$, requiring two T and two X switch settings as indicated. The heavy lines in Figure 7.74 mark the path along which packages travel from S to D. If all switches are set to T, then $S = D$, so each processor is connected to itself via a path through $\log_2 N$ switches. Changing the state of the switch in stage $i + 1$ along this path from T to X connects the source processor to the destination processor that differs from it in the ith address bit. It follows that there is only one path through each of the foregoing ($\log_2 N$)-stage networks linking every source-destination pair.

The routing of packages through a MIN can be managed by a centralized controller attached to the network that examines all source-destination address pairs S, D generated by processors and sets the appropriate switching elements to the states specified by $R = S \oplus D$. An alternative is to attach R as a *routing tag* to each package to be transmitted from S to D and to use R to set the switching element states as the package passes through the MIN. When the package enters a switch $S_{j,i+1}$ in stage $i + 1$, $S_{j,i+1}$ examines the routing tag R using control logic built into the switch for this purpose. $S_{j,i+1}$ then sets its own state to T if $r_i = 0$, and to X if $r_i = 1$. Thus the centralized controller can be replaced by decentralized control logic distributed throughout the MIN. Each package determines its own path through the MIN and so can be viewed as *self-routing*. For example, to transmit a package from $S = 2$ to $D = 14$ in the four-stage MIN of Figure 7.74, the routing tag $R = r_3r_2r_1r_0 = 1100$ is appended to the package generated by the source processor P_2. The switch $S_{2,1}$ attached to P_2 in stage 1 inspects bit r_0 of R. Since $r_0 = 0$, switch $S_{2,1}$ sets itself to the through state T. This setting causes the package to be sent to the topmost switch $S_{1,2}$ in stage 2, which also sets its state to T, since $r_1 = 0$. The package proceeds to the final two stages, which set themselves to the cross state X, since $r_2 = r_3 = 1$.

The Butterfly computer developed by Bolt, Beranek and Newman Inc. around 1980 [Crowther et al. 1985] and its successor the TC2000 introduced in 1989 are examples of commercial multiprocessors based on MINs. They are shared-memory MIMD computers in which the MIN connects N processors to N memory units that form the shared memory. In the original Butterfly multiprocessor, the processors are based on the Motorola 680X0 series, and N ranges from 1 to 256. Every processor contains a microprogrammed coprocessor to handle virtual memory management, package transfer to and from the MIN, and related functions.

The Butterfly's MIN has single-chip 4×4 switching elements, each of which is obtained by cascading two copies of the basic butterfly network of Figure 7.72. Consequently, the processor-memory interconnection network is an $N \times N$ butterfly MIN composed of $\log_2 N$ stages of 2×2 switching elements. Data transmission through the network is by bit-serial packages, which can be transmitted at a rate of 32 Mb/s along any processor-memory path. Each package contains its destination address and is made self-routing in the manner described earlier by employing 2 bits of the destination address to determine the setting of each 4×4 switch through which the package passes. Should two packages attempt to use the same link in the MIN simultaneously, one is allowed to proceed and the other is retrans-

mitted after a short delay. This type of application-dependent contention increases the execution time of a typical program by only a few percent.

7.3.3 Fault Tolerance

Fault tolerance has been defined as "the ability of a system to execute specified algorithms correctly regardless of hardware failures and program errors" [Avizienis 1971]. It is of some concern in all computer systems, while in applications such as spacecraft control and telephone switching, fault tolerance is a major design goal [Siewiorek and Swarz 1992]. Most hardware failures have physical causes such as component wear or electromagnetic interference. The nature and frequency of these failures can be determined experimentally, which makes it possible to study the faults and their consequences using analytic or simulation models. Software faults are primarily due to algorithm or programming mistakes (design errors) and so are more difficult to deal with.

Redundancy. Fault tolerance is intimately associated with the concept of redundancy. When a component fails, its duties must be taken over by other, fault-free components of the system. If those components are intended to improve only the reliability of the system and do not significantly affect its computing performance, they are termed *redundant*. Redundancy can be introduced in several overlapping ways:

- *Hardware redundancy*: Multiple copies of critical hardware units.
- *Software redundancy*: Multiple versions of programs for critical operations.
- *Information redundancy:* Error-correcting or error-detecting codes.
- *Time redundancy:* Repeating or retrying critical operations.

The goal of these redundant design features is to prevent failures due to physical faults or design mistakes from producing *errors*, that is, data values or operating modes that lead to system failure. Information redundancy via coding methods is discussed in section 3.2.1. In this section, we examine the use of redundant hardware to achieve fault tolerance.

Two broad approaches, static and dynamic redundancy, have been identified for designing fault-tolerant systems. *Static redundancy* refers to the use of redundant hardware or software components, which form a permanent part of the system, to mask the error signals generated by faults. One form of static redundancy replaces a critical unit that generates a word X with $n \geq 3$ copies of that unit, configured to generate n independent copies of X in parallel. If the unit in question is a processor, then the resulting system is a type of multiprocessor. The n versions of X are applied to a circuit called a *voter*, which is designed to output the value of X appearing on the majority of its n input buses. Thus errors produced by any of the replicated units are masked by the voter, provided more than half of the units produce the correct X values at all times. A system of this type with n identical units and a voter is said to employ *n-modular redundancy* (*nMR*).

A frequently implemented version of nMR is *triple modular redundancy* (*TMR*), in which $n = 3$, as shown in Figure 7.75. In this case the behavior of the voter is defined by the logic equation

$$X = X_1X_2 + X_1X_3 + X_2X_3$$