SECTION 18

DIGITAL COMPUTER SYSTEMS

Murray J. Haims, Stephen C. Choolfaian, Daniel Rosich, Richard E. Matick, William C. McGee, Benton D. Moldow, Robert A. Myers, George C. Stierhoff, Claude E. Walston

No other invention in the twentieth century has impacted virtually every aspect of our life like the digital computer. The hardware along with software have forever changed how we work with everything from the automobile to our checking accounts. It goes to the deepest parts of our oceans and planet to well beyond our solar system. In spite of how pervasive and sophisticated they are, they remain fairly simple and straight forward devices (although anyone who works with them regularly could easily argue with us on this point). An understanding of the material in this section should allow us to work with many hardware and software challenges.

Computers are organized into basic activities. This is referred to as the architecture of the computer as well as the software. The first part to look at is data processing. Data are merely the representation of something in the real world such as money by their binary equivalent. After we look at the rest of the architecture of the computer we will look at how these can be varied in the design process to produce different types of computer activities.

Next we look at how software can control the interactions of the hardware to produce the desired results. Keep in mind that even though there are a number of different software packages that we use, they all do essentially the same things and all that is different is really just the syntax of the specific application.

Future trends in this field will essentially focus on two areas. The first will be in the area of how we interact with computer and input and output data. We already have the ability to use handwritten interaction and voice interaction. It is in the area of verbal interaction with the computer where we will most likely see the end of the keyboard as we know it. Current voice interactive software can achieve accuracies of 97 percent with training. This compares favorably with keyboard input and is decidedly faster for everyone but the most skilled typist.

The other area will be in the basic architecture of both the hardware and software. Increasingly we are developing portions of the computer into parallel processors that will enable a major shift in the way software will be developed. Eventually we will have computers that come from a manufacturer with a defined hardware and software architecture. Once the computer is turned on it will begin reconfiguring its hardware and software to adapt to the needs of the user. C.A.

In This Section:

Section Bibliography:

Aiken, A. H., and G. M. Hopper "The automatic sequence controlled calculator," *Elec. Eng.,* 1948, Vol. 65, p. 384.

Babbage, C. "Passages from the Life of a Philosopher," Longmans, 1864.

Babbage, H. P. "Babbage's Calculating Engines," Spon, 1889.

Bjorner, D., E. F. Codd, K. L. Deckert, and I. L. Traiger "The GAMMA-O *n*-ary relational data base interface specification of objects and operations," *Research Report RJ1200*, IBM Research Division, 1973.

Black, V. D. "Data Communications and Distributed Networks," 2nd ed., Prentice Hall, 1987.

Boole, G. "The Mathematical Analysis of Logic," 1847, reprinted Blackwell, 1951.

Brainerd, J. G., and T. K. Sharpless "The ENIAC," *Elec. Eng.,* February 1948, pp. 163–172.

Brooks, F. P., Jr. "The Mythical Man-Month," Addison-Wesley, 1975.

Codd, E. F. "A data base sublanguage founded on the relational calculus," *Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control*, Association for Computing Machinery, 1971.

Cypser, R. J. "Communications Architecture for Distributed Systems," Addison-Wesley, 1978.

Deitel, H. M. "Operating Systems," 2nd ed., Addison-Wesley, 1990.

Dijkstra, E. W. *Commun. Ass. Comput. Mach.*, 1968, Vol. II, No. 3, p. 341.

Dijkstra, E. W. "A Discipline of Programming," Prentice Hall, 1976.

Enslow, P. H., Jr. "Multiprocessor organization—a survey," *Comput. Surv.,* March 1977, Vol. 9, No. 1, pp. 103–129.

Feldman, J. M., and C. T. Retter "Computer Architecture, A Designer's Text Based on a Generic RISC," McGraw-Hill, 1994.

Flynn, M. J. "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, September 1972, Vol. C-21, No. 9, pp. 948–960.

Gear, C. W. "Computer Organization and Programming," 2nd ed., McGraw-Hill, 1978.

Gilmore, C. M. "Microprocessor Principles and Applications," McGraw-Hill, 1989.

Godman, J. E. "Applied Data Communication," Wiley, 1995.

Hamming, W. R. "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, 1947, Vol. 29.

Hancock, L., and M. Krieger "The C Primer," McGraw-Hill, 1986.

Hellerman, H. "Digital Computer System Principles," 2nd ed., McGraw-Hill, 1973.

Horowitz, E., and S. Sani "Fundamentals of Data Structures," Computer Science Press, 1976.

Lam, S. L. "Principles of Communication and Networking Protocols," Computer Society Press, 1984.

Mano, M. M. "Computer System Architecture," 2nd ed., Prentice Hall, 1982.

Mano, M. M. "Digital Logic and Computer Design," Prentice Hall, 1979.

Morris, D. C. "Relational Systems Development," McGraw-Hill, 1987.

O'Connor, P. J. "Digital and Microprocessor Technology," 2nd ed., Prentice Hall, 1989.

Smith, J. T. "Getting the Most from TURBO PASCAL," McGraw-Hill, 1988.

Stallings, W., and R. Van Slyke "Business Data Communications," 2nd ed., Macmillan, 1994.

Van de Goor, A. J. "Computer Design and Architecture," Addison-Wesley, 1985.

Wegner, P. "Programming Languages, Information Structures and Machine Organization," McGraw-Hill, 1968.

Whitten, J. L., L. D. Bentley, and V. M. Barlow "Systems Analysis and Design Methods," 3rd ed., Irwin, 1994.

CHAPTER 18.1
COMPUTER ORGANIZATION

PRINCIPLES OF DATA PROCESSING

Memory, Processing, and Control Units

The basic subsystems in a computer are the input and output sections, the store, arithmetic logic unit (processing unit), and the control section. Each unit is described in detail in this chapter. Generally, a computer operates as follows: An external device such as a disc file delivers a program and data to specific locations in the computer store. Control is then transferred to the stored program, which manipulates the data and the program itself to generate the output. These output data are delivered to a device, such as a CD-ROM/RW, DVD, printer, or display, where the information is used in accordance with the purpose of the digital manipulation.

Historical Background

There has been a line of development of mechanical calculator components, beginning with Babbage in the early 1800s and leading to a variety of mechanical desk and larger mechanical calculators. Another line of development has used relays as computing circuit elements. Today's computers have benefited from these lines of development, but especially they are based on electronic components, the vacuum tube, and the transistor. The transistor, first described by Shockley, Bardeen, and Brattain in 1974, began a line of development that is today characterized by the miniaturization and low-power operation of very large-scale integration (VLSI).

VLSI permits the interconnection of large numbers of computing elements by means of microscopic layered structures on a semiconductor substrate (usually silicon) or *chip* sometimes as small as $^1/_4$ in. square. Since the entire arithmetic and logic circuit of a computer can be built on a single chip (microprocessor), computers incorporating VLSI are often called *minicomputers* or *microcomputers.*

Binary Numbers

Most transistors display random variations of their operating parameters over relatively wide limits. Similarly, passive circuit elements experience a considerable degree of variation, and noise and power-supply variations, and so forth, limit the accuracy with which quantities can be represented. As a result, the preferred method is to use each circuit in the manner of an on-off switch, and representation of quantities in a computer is thus almost always on a *binary* basis. Figure 18.1.1 shows the binary numbers equivalent to the decimal numbers between 1 and 10. Figure 18.1.2 shows the addition of binary 6 to binary 3 to obtain binary 9.

The process of addition can be dissected into digital, logical, or boolean operations upon the binary digits, or bits (b). For example, a first step in the procedure for addition is to form the so-called EXCLUSIVE-OR addition between bits in each column. This function of two binary numbers is expressed in Fig. 18.1.3*a* in tabular form. This table is called a *truth table.* In Fig. 18.1.3*b* is the table used to generate the *carries* of a binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

FIGURE 18.1.1 Decimal and binary numbers between 0 and 10.

Decimal	Binary
6	110
+3	+ 11
9	1001

FIGURE 18.1.2 Addition of 6 and 3 in decimal and binary.

	0	1
0	0	1
1	1	10

Binary addition table excluding the carry (*a*)

	0	1
0	0	0
1	0	1

Binary addition carry generation (*b*)

FIGURE 18.1.3 Addition tables for decimal and binary numbers. The binary addition table (*a*) is called the EXCLUSIVE-OR or module-2 truth table; the carry table (*b*) performs the AND or *intersection* operation.

bit from one column to another. This latter function of two binary numbers is variously called the AND function, *intersection*, or *product*. The entries at each intersection in each table are the result of the combination of two binary numbers in the respective row and column. Figure 18.1.3 also shows the decimal addition tables. They illustrate the relative simplicity of the binary number system.

The names truth table and logical function arise from the fact that such manipulations were first developed in the *sentential calculus*, a subsection of the calculus of logic, dealing with the truth or falsity of combinations of true or false sentences.

Binary Encoding. Information in a digital processing machine is not restricted to numerical information since a different specific numeric code can be assigned to each letter of the alphabet. For example, A in the *EBCDIC code* (see next paragraph) is given by the binary sequence 11000010. When alphanumeric information is specified, such a code sequence represents the symbol A, but in the numeric context the same entry is the binary number equal to decimal 194.

Computer Codes

Alphanumeric information is stored in a computer via coded binary bits. Some of the more useful codes are:

ASCII (American Standard Code for Information Interchange), a seven-level alphanumeric code comprising 32 control characters, an uppercase and lowercase alphabet, numerals, and 34 special characters (Fig. 18.1.4). This code is used in personal computers and non-IBM machines. "A" in ASCII is given by the 7-bit binary codes 1000001.

BCDIC (binary-coded decimal interchange code), a six-level alphanumeric code that provides alphabetic (caps), numeric, and 28 special characters.

Binary code, a representation of numbers in which the only two digits used are 0 and 1, and each position's value is twice that of its right-hand neighbor (with the rightmost place having a value of 1).

Binary-coded decimal (BCD) code, in which the first ten 4-bit (hexadecimal) codes are used for the decimal digits, and each nibble represents one decimal place. Codes A through F are never used.

EBCDIC (expanded BCD interchange code), an eight-level alphanumeric code comprising control codes, an uppercase and lowercase alphabet, numerals, and special characters (Fig. 18.1.5). This code is used in IBM mainframe computers.

Gray code, a binary code that does not follow the positional notation of true binary code. Only one bit changes from any Gray number to the next.

Hexadecimal byte code, a two-digit hexadecimal number for each byte, with values ranging from 00 to FF.

Hexadecimal code, a base-16 number code that uses the letters A, B, C, D, E, and F as one-digit numbers for 10 through 15.

Hollerith code, a 12-level binary code used on punchcards to represent alphanumeric characters. Holes on the punchcard are ones, unpunched positions are zeros.

"back" 5 bits	00000	00001	00010	00011	00100	00101	00110	00111	01000	01001	01010	01011	01100	01101	01110	01111	10000	10001	10010	10011	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101	11110	11111
"front" 2 bits																																
00	Control code for typewriter control which don't print anything																															
01	b	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	/	]	^	←
11	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	¦	l	¦	~	DEL

FIGURE 18.1.4 American Standard Code for Information Interchange (ASCII).

Most computers are designed to work internally in binary fashion but at each juncture of input or output to translate the codes, either by programming or by hardware, so as to accept and offer decimal numeric information. Such systems are complicated, and failures in the coding and decoding system can prevent interaction with the program.

Communication needs lead to a compromise between the human requirement for a decimal system and the machine requirement for binary. An example is the use of the base-16 (hexadecimal) system, which is relatively amendable to human recognition and manipulation. The binary code is broken into 4-bit groups. Each 4-bit group is represented by a decimal numeral or letter, as indicated in Fig. 18.1.6. In this case, 0011 is three, and 1010 is ten or A.

Error-Correction Codes

Though the circuits in modern digital systems have reached degrees of reliability undreamed of in the relatively recent past, errors can still arise. Hence it is desirable to detect and, if possible, correct such errors. It is

Some characters of the expanded B.C.D. interchange code (EBCDIC)

No #	Numeric 1	Numeric 2	Numeric 3	Numeric 4	Numeric 5	Numeric 6	Numeric 7	Numeric 8	Numeric 9	Zone bits	Binary code
No zone Ø	1	2	3	4	5	6	7	8	9	111	$D \cdot C \cdot B \cdot A$
12 zone &	A	B	C	D	E	F	G	H	I	1100	$D \cdot C \cdot \bar{B} \cdot \bar{A}$
11 zone –	J	K	L	M	N	O	P	Q	R	1101	$D \cdot C \cdot \bar{B} \cdot A$
0 zones 0	/	S	T	U	V	W	X	Y	Z	1110	$D \cdot C \cdot B \cdot \bar{A}$
12 • 0 zones	a	b	c	d	e	f	g	h	i	1000	$D \cdot \bar{C} \cdot \bar{B} \cdot \bar{A}$
12 • 11 zones	j	k	l	m	n	o	p	q	r	1001	$D \cdot \bar{C} \cdot \bar{B} \cdot A$
11 • 0 zone	?	s	t	u	v	w	x	y	z	1010	$D \cdot \bar{C} \cdot B \cdot \bar{A}$
Numeric bits	0001	0010	0011	0100	0101	0110	0111	1000	1001		
Binary code	$\bar{8} \cdot \bar{4} \cdot \bar{2} \cdot 1$	$\bar{8} \cdot \bar{4} \cdot 2 \cdot \bar{1}$	$\bar{8} \cdot 4 \cdot \bar{2} \cdot \bar{1}$	$\bar{8} \cdot 4 \cdot \bar{2} \cdot \bar{1}$	$\bar{8} \cdot 4 \cdot \bar{2} \cdot 1$	$\bar{8} \cdot 4 \cdot 2 \cdot \bar{1}$	$\bar{8} \cdot 4 \cdot 2 \cdot 1$	$8 \cdot \bar{4} \cdot \bar{2} \cdot \bar{1}$	$8 \cdot \bar{4} \cdot \bar{2} \cdot 1$		

FIGURE 18.1.5 Hollerith/EBCDIC code chart.

Names of 4 bit groups

Digit name	*4-bit binary*	*Digit symbol*	*Digit name*	*4-bit binary*	*Digit symbol*
Zero	0000	0	Eight	1000	8
One	0001	1	Nine	1001	9
Two	0010	2	Ten	1010	A
Three	0011	3	Eleven	1011	B
Four	0100	4	Twelve	1100	C
Five	0101	5	Thirteen	1101	D
Six	0110	6	Fourteen	1110	E
Seven	0111	7	Fifteen	1111	F

FIGURE 18.1.6 Hexadecimal code.

possible by appropriate selection of binary codes to detect errors. For example, if a 6-b code is used, a seventh bit can be added to maintain the number of 1 bits in the group of 7 as an odd number. When any group of 7 with an even number of 1s is found by appropriate circuits in the machine, an error is detected. Such a procedure is known as *parity checking.* Although these error-control coding schemes were originally developed for noisy transmission channels, they are also applicable to storage devices in data-processing systems.

Boolean Functions

Figure 18.1.7 illustrates truth tables for functions of one, two, and three binary variables. Each x entry in each table can be either 0 or 1. Hence for one variable x four functions $f(x)$ can be formed; for two variables x_1 and x_2, 16 functions $f(x_1, x_2)$ exist; for three variables, 256 functions, and so on. In general, if $f(x_1, \ldots, x_n)$ is a function of n binary variables, 2^{2n} such functions exist.

For functions of one variable, the most important is the *inverse function*, defined in Fig. 18.1.8. This is called NOT A or $\bar{A}$, where A is the binary variable. Also illustrated in Fig 18.1.8 are the two most important functions of two binary variables, the AND (*product* or *intersection*) and the OR (*sum* or *union*). If A and B are the two variables, the AND function is usually represented as AB and the OR as $A + B$.

Figure 18.1.9 shows how the products AB, $\bar{A}B$, $A\bar{B}$ and $\bar{A}\bar{B}$ are summed to yield any function of two binary variables. Each of these products has only one 1 in the four positions in their respective truth tables, so that appropriate sums can generate any function of two binary variables. This concept can be expanded to functions of more than two variables, i.e., any function of n binary variables can be expanded into a sum of products of the variables and their negatives. This is the general theorem of boolean algebra. Such a sum is called the *standard sum* or the *disjunctive normal form.*

The fact that any binary function can be so realized implies that mechanical or electrical simulations of the AND, OR, and NOT functions of binary variables can be used to represent any binary function whatever.

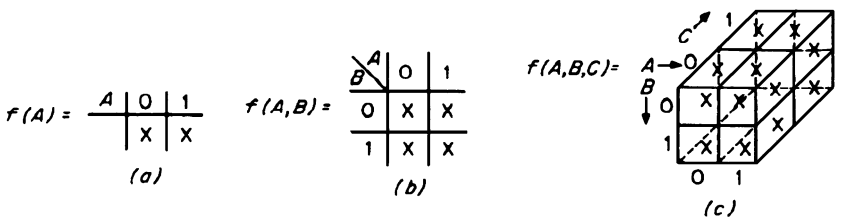

FIGURE 18.1.7 Binary functions of (*a*) one, (*b*) two, and (*c*) three binary variables.

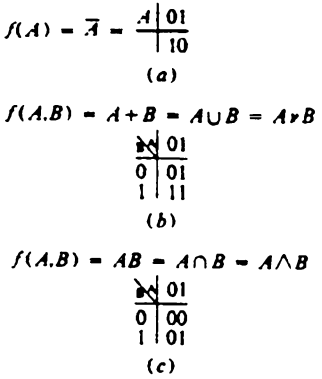

FIGURE 18.1.8 Significant functions of one and two binary variables: (*a*) the negation (NOT) function of one binary variable; (*b*) the OR function of two binary variables; (*c*) the AND function of two binary variables.

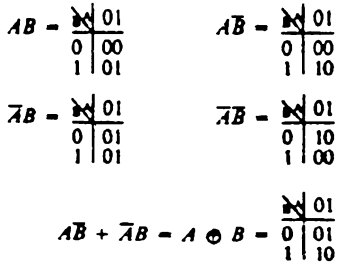

FIGURE 18.1.9 The four products of two binary variables (top). The realization of the EXCLUSIVE-OR function is shown below.

Electronic Realization of Logical Functions

Logical functions can be realized using electronic circuits. Figure 18.1.10 illustrates the realization of the OR function and Fig. 18.1.11 illustrates the realization of the AND circuit using diodes. Each input lead is associated with a boolean variable, and the upper level of voltage represents the logical 1 for that variable; in the OR circuit, any input gives rise to an output. Thus for a three-variable input, the output is $A + B + C$. With the AND function no output is realized unless all inputs are positive; the output function generated is ABC.

The inverse function (NOT) of a boolean variable cannot be readily realized with diodes. The circuit shown in Fig 18.1.12 uses the inverting property of a grounded-emitter transistor amplifier to perform the inverse function. Also shown in Fig. 18.1.12 is an example of how the OR function and the NOT function are combined to form the NOT-OR (NOR) function. In this case, since the transistor circuit provides both voltage and current gain, the signal-amplitude loss associated with transmission through the diode can be compensated, so that successive levels of logic circuits can be interconnected to form complex switching nets.

Figure 18.1.13 illustrates the realization of the EXCLUSIVE-OR function. Note that the variables are represented by the wiring of interconnected circuit blocks, while the function is realized by the circuit blocks themselves.

Levels of Operation in Data Processing

A detailed sequence of operations is generally required in a data-processing system to realize even simple operations. For example, in carrying out addition, a machine typically performs the following sequence of operations:

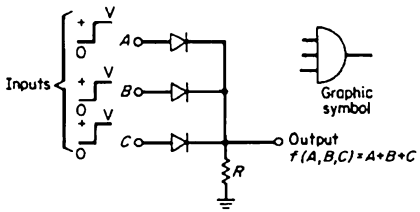

FIGURE 18.1.10 Diode realization of an OR circuit. A positive input on any line produces an output.

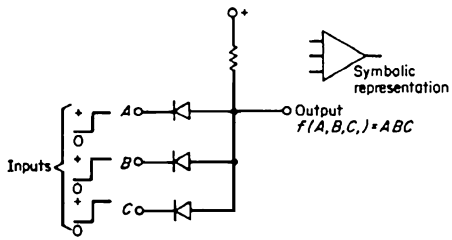

FIGURE 18.1.11 Diode realization of an AND circuit. All inputs must be positive to produce an output.

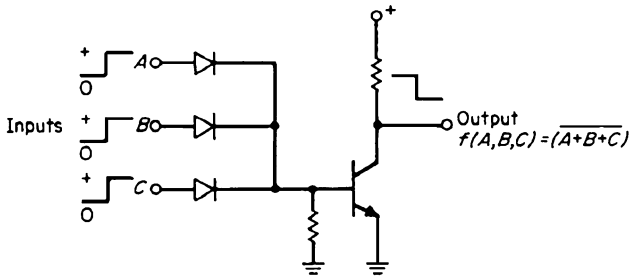

FIGURE 18.1.12 Use of a transistor circuit for inverting a function. The circuit shown forms the NOT-OR (NOR) of the inputs.

1. Fetch a number from a specific location in storage.
   - ***a.*** Decode the address of the program instruction to activate suitable memory lines. Such decoding is accomplished by activating appropriate AND and OR gates to apply voltage to the lines in storage specified by the instruction address.
   - ***b.*** Sequence storage to withdraw the information and place it in a storage output register.
   - ***c.*** Transmit information from the storage output register into the appropriate ALU.

2. Withdraw a number from storage and add it to the number in the ALU. These operations break down into:
   - ***a.*** Decode the instruction address, activate storage lines, and transmit the information to the ALU input for addition.
   - ***b.*** Form the EXCLUSIVE-OR of the second number with the number in the ALU to form the sum less the carry. Form the AND of the two numbers to develop the first-level carry.
   - ***c.*** Form the second-level EXCLUSIVE-OR sum.
   - ***d.*** AND the first-level carry with the first-level EXCLUSIVE-OR sum to form the second-level carry.
   - ***e.*** Generate the third-level EXCLUSIVE-OR by forming the EXCLUSIVE-OR of the second-level carry with the second-level EXCLUSIVE-OR sum, AND the second-level carries with the second-level EXCLUSIVE-OR for the third-level carry and so forth until no more carries are generated.

3. Store the result of the addition into a specified location in storage.

This sequence illustrates two basic types of operation in a data-processing machine. Operations denoted above by numbers are of specific interest to the programmer, since they are concerned with the data stored and the operations performed thereupon. The second level, denoted above by letters, are operations at the logical-circuit level within the machine. These operations depend on the particular configurations of circuits and other hardware in the machine at hand.

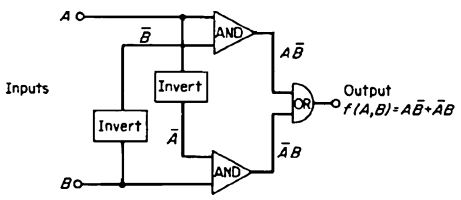

FIGURE 18.1.13 Circuit realization of the EXCLUSIVE-OR function.

If only the higher-level (numbered) instructions are used, some flexibility in machine operation is lost. For example, only an add operation is possible at the higher level. At the lower-level (lettered) operations the AND or EXCLUSIVE-OR of the data words can be formed and placed in storage.

The organization of current digital computers follows the lines of these two divisions (numbered and lettered, above). The *macroinstruction set* associated with each machine can be manipulated by the programmer. These instructions are usually implemented in a numerical code. For example, the instruction "load ALU" might be 01 in binary. "Add ALU" might be given by 10 and "store ALU" by 11. Similarly, each instruction has an associated storage address to provide source data. The microinstruction set comprises a series of suboperation that is combined in various sequences to realize a given macroinstruction.

Two methods of realizing the sequence of suboperations specified by the operations portion of the instruction have been used in machine design. In one such method a direct decoding of the information from the instruction occurs when it is placed in an instruction address register. Specific clock sequences turn on the successively required lines that have been wired in place to realize the action sought.

An alternative for actuating a subprogram is to store a number of information bits, called *microinstructions*, that are successively directed to the appropriate control circuits to activate selectively and sequentially individual wires to gate sequential actions for the realization of the requisite instruction.

The first method of computer design is called *hard-wired*, and the second is the *microprogrammed.* The microprogram essentially specifies a sequence of operations at the individual circuit level to specify the operations performed by macroinstructions. Microprogramming is preferred in modern computer designs.

Types of Computer Systems

There is a wide variety of computer-system arrangements, depending on the type of application. One type of installation is that associated with *batch processing.* A computer in a central job location receives programs from many different sources and runs the programs sequentially at high speed. An overall supervisory program, called an *operating system*, controls the sequence of programs, rejecting any that are improperly coded and completing those which are correct.

Another type of system, the *time-shared system*, provides access to the computer from a number of remote input-output stations. The computer scans each remote station at high speed and accepts or delivers information to that location as required by the operator or by its internal program. Thus a small terminal can gain access to a large high-speed system.

Still another type of installation, the *microcomputer*, involves an individual small computer that, though limited in power, is dedicated to the service of a single operator. Such applications vary from those associated with a small business, with limited computational requirements, to an individual engaged in scientific operations.

Other computers are used for dedicated control of complex industrial processes. These are individual, once-programmed units that perform a *real-time* operation in *systems control*, with sensing elements that provide the inputs. Highly complex interrelated systems have been developed in which individual computers communicate with and control each other in an overall major *systems network*. Among the first of such systems was the SAGE network, developed in the 1950s for defense against missile or aircraft attack.

Computers that are interconnected to share workload or problems are said to form a *multiprocessing system.* A computer system arranged so that more than program can be executed simultaneously is said to be *multiprogrammed.*

Interactive systems allow users to communicate directly with the computer and have the computer respond. The development of these systems parallels that of the keyboard and of the video display. The systems are used commercially (e. g., airline reservations) and scientifically (users input data at their terminals or telephones and get a response to their input). The *terminal*, a widely used input/output device, has a keyboard and a visual display. A terminal may be *dumb*, which means that it has no computing power, or *smart*, which indicates computing capabilities, such as those provided by a personal computer. *Client–server* systems are interactive systems where the data are at a remote computer called a server.

Internal Organization of Digital Computers

The internal organization of a data-processing system is called the *system architecture.* Such matters as the minimum addressable field in memory, the interrelations between data and instruction word size, the instruction format and length or lengths, parallel or serial (by bit or set of bits) ALU organization, decimal or binary internal organization, and so forth, are typical questions for the system architect. The answers depend heavily on the application for which the computer is intended.

Two broad classes of computer systems are *general-purpose* and *special-purpose* types. Most systems are in the general-purpose class. They are used for business and scientific purposes. General-purpose computers of varying computer power and memory size can be grouped, sharing a common architecture. These are said to constitute a *computer family.*

A computer scientifically designed for, and dedicated to the control of, say, a complex refinery process is an example of a special-purpose system.

A number of design methods have been adopted to increase the speed and functional range for a small increase in cost. For example, in the instruction sequence, the next cell in storage is likely to be the location of the next instruction. Since an instruction can usually be executed in a time that is short compared with storage access, the store is divided into subsections. Instructions are called from each subsection independently at high speed and put into a queue for execution. This type of operation is called *look-ahead.* If the instructions are not sequential, the queue is destroyed and a new queue put in its place.

Since instructions and data tend to be clustered together in storage, it is advantageous to provide a small, high-speed store (local store) to work with a larger, slower-speed, lower-cost unit. If the programs in the local store need information from the larger store, a least-used piece of the local store reverts to the larger store and a batch of data surrounding the information sought is automatically brought into the high-speed unit. This arrangement is called a *hierarchical memory* and the high-speed store is often called a *cache.*

NUMBER SYSTEMS, ARITHMETIC, AND CODES

Representation of Numbers

A set of codes and names for numbers that meets the requirements of expendability and convenience of operation can be obtained using the following power series:

$$N = A_n X^n + A_{n-1} X^{n-1} + \cdots + A_1 X + A_0 + A_{-1} X^{-1} + \cdots + A_{-m} X^{-m} \tag{1}$$

Here the number is represented by the sum of the powers of an integer X, each having a coefficient A_i. A_i may be an integer equal to or greater than zero and less than X. In the decimal system, X equals 10 and the coefficients A_i range from 0 to 9.

Note that Eq. (1) can be used to represent X^{m+n+1} numbers ranging between 0 and $X^{n+1} - X^m$ with an accuracy limited by X_m. Thus m and n must be of reasonable size to be useful in most applications. A useful property of the power series is the fact that its multiplication by X^k can be viewed as a shift of the coefficients of any given term by the number of positions specified by the value of k. These results are independent of the choice of X in the series representation.

There is little reason to write the value of the number in the form shown in Eq. (1) since complete information on the value can be readily deduced from the coefficient A_i. Thus, a number can be represented merely by a sequence of the values of the coefficients. To determine the value of the implied exponents on X, it is customary to mark the position of the X_0 term by a period immediately to the right of its coefficient. The power series for a number represented in the decimal system ($X = 10$) and its normal decimal notation are

$$3 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} = 3{,}024.62 \tag{2}$$

The value of X is called the *radix* or *base* of the number system. Where ambiguity might arise, a subscript to indicate the radix is attached to the low-order digit, as in $1000_2 = 8_{10} = 10_8$ (1000 binary equals 8 decimal equals 10 octal). The power series for a number in base 2 and its representation in binary notation is

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 11011.011 \tag{3}$$

Number-System Conversions

Since computer systems, in general, use number systems other than base 10, conversion from one system to another must be carried out frequently. Equation (4) shows the integer N represented by a power series in base 10 and base 2

$$\sum_{i=0}^{n} A_i 10^i = \sum_{j=0}^{m} B_j 2^j \tag{4}$$

The problem is to find the correlation between the coefficients A_i and B_j. In the binary series, if N is divisible by 2, then B_0 must be 0. Similarly, if N is divisible by 4, B_1 must be 0, and so forth. Thus if the decimal coefficients A_i are given, successive divisions of the decimal number by 2 will yield the binary number, the binary digits depending on the value of the remainder of each successive division. This process is shown in Fig. 18.1.14.

The conversion of a binary integer to a decimal integer is

$$\begin{aligned} 100011011 &= 1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 \\ &\quad + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 \times 2^0 \\ &= 283 \end{aligned}$$

In the case of conversion of an integer in binary to an integer in decimal, the powers of 2 are written in decimal notation and a decimal sum is formed from the contribution of each term of the binary representation. For conversion from a binary fraction to a decimal fraction, a similar procedure is used since the value of terms as multiplied by the A_i can be added together in decimal form to form the decimal equivalent.

The conversion of a decimal fraction to a binary fraction is defined by

$$0.5764_{10} = A_{-1}2^{-1} + A_{-2}2^{-2} + \cdots + A_n 2^{-n} \tag{5}$$

To determine the values of the A_i, first multiply both sides of Eq. (27.5) by 2 to give

$$1.1528_{10} = A_{-1} + A_{-2}2^{-1} + \cdots + A_{-n}2^{n-1} \tag{6}$$

Since the position of the decimal point (more accurately called the *radix point*) is invariant, and since in a binary series each successive term is at most half of the maximum value of the preceding term, the leading 1 in the decimal number in Eq. (6) indicates that A_{-1} must have been 1. A second multiplication by 2 can similarly determine the coefficient A_{-2}. This process of conversion of a base-10 fraction to a base-2 fraction is illustrated in Fig. 18.1.15.

Conversion from binary integers to octal (base 8) and the reverse can be handled simply since the octal base is a power of 2. Binary to octal conversion consists of grouping the terms of a binary number in threes and replacing the value of each group with its octal representation. The process works on either side of a decimal point. The octal-to-binary conversion is handled by converting each octal digit, in order, to binary and retaining the ordering of the resulting groups of three bits.

Since there are not enough symbols in decimal notation to represent the 16 symbols required for the hexadecimal system, it is customary in the data processing field to use the first six letters of the alphabet to complete the set.

Conversions from decimal to octal or hexadecimal can proceed indirectly by first converting decimal to binary and then binary to octal or hexadecimal. Similarly, a reverse path from octal or hexadecimal to binary to decimal can be used. Direct conversions, however, between hexadecimal and octal and decimal exist and are widely used. In going from hexadecimal or octal to decimal, each term in the implied power series is expressed directly in decimal, and the result is summed. In converting from a decimal integer to either hexadecimal or octal, the decimal is divided by either 16 or 8, respectively, and the remainder becomes the next higher-order digit in the converted number. Examples of four common number representations are shown in Table 18.1.1.

Binary-Arithmetic Operations

Figure 18.1.16 shows an example of the addition of two binary numbers, 1001 and 1011 (9 and 11 in decimal). The rules for manipulation are similar to those in decimal arithmetic except that only the two symbols 1 and 0 are used and the addition and carry tables are greatly simplified.

Figure 18.1.17 shows an example of binary multiplication with a multiplication table. This process is also simple compared with that used in the decimal system. The rule for multiplication in binary is as follows: if a particular digit in the multiplier is 1, place the multiplicand in the product register; if 0, do nothing; in either case shift the product register to the right by one position; repeat the operations for the next digit of the multiplier.

Figure 18.1.18 shows an example of binary subtraction and the subtraction and borrow tables. The subtraction table is the same as the addition table, a feature unique to the binary system. The borrow operation is handled in

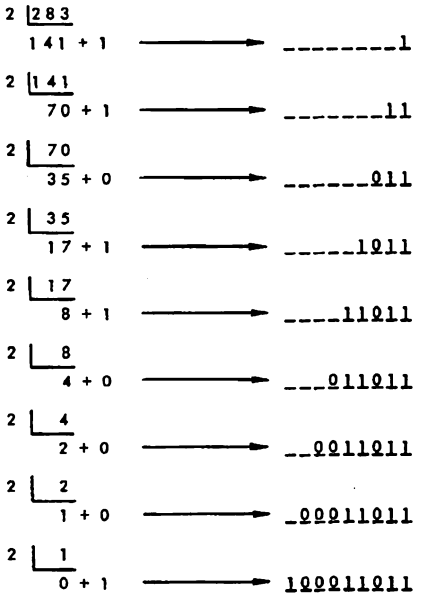

FIGURE 18.1.14 Conversion from a decimal to binary by repeated division of the decimal integer. At each division the remainder becomes the next higher-order binary digit.

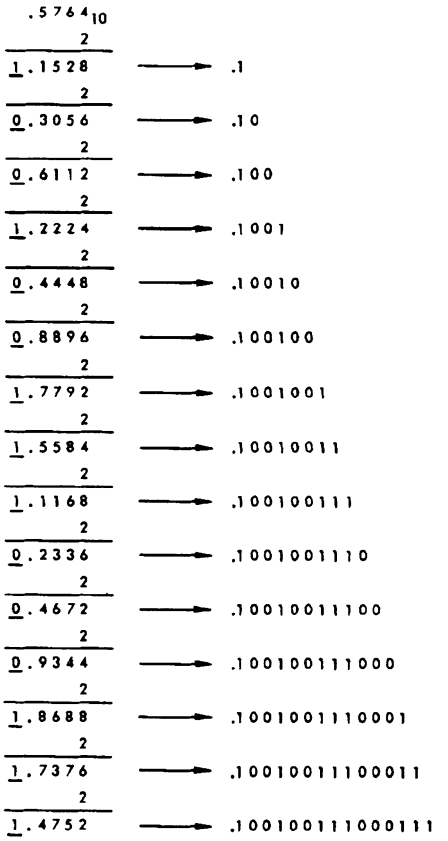

FIGURE 18.1.15 Conversion of a decimal fraction into a binary fraction. At each stage the number to the right of the decimal is multiplied by 2. The resulting number to the left of the decimal point is entered as the next available lower-order position of the binary fraction to the right of the binary radix point.

TABLE 18.1.1 Comparison of Decimal, Binary, Octal, and Hexadecimal Numbers

Decimal	Binary	Octal	Hexadecimal	Decimal	Binary	Octal	Hexadecimal
0	0	0	0	8	1000	10	8
1	1	1	1	9	1001	11	9
2	10	2	2	10	1010	12	A
3	11	3	3	11	1011	13	B
4	100	4	4	12	1100	14	C
5	101	5	5	13	1101	15	D
6	110	6	6	14	1110	16	E
7	111	7	7	15	1111	17	F

Binary		Decimal
1011	=	11
+ 1001	=	9
10100	=	20

FIGURE 18.1.16 Binary addition and corresponding decimal addition.

Binary	Decimal
1011	11
× 1001	× 9
1011	99
101100	
1100011	

×	0	1
0	0	0
1	0	1

FIGURE 18.1.17 Binary multiplication. The binary multiplication table is the AND function of two binary variables. The process of multiplication consists of merely replicating and adding the multiplicand, as shown, if a 1 is found in the multiplier. If 0 is found, a single 0 is entered and the next position to the left in the multiplier is taken up.

a fashion analogous to that in decimal. If a 1 is found in the preceding column of the subtrahend, it is borrowed, leaving a 0. If a 0 is found, an attempt is made to borrow from the next higher-order position, and so forth.

An example of binary division is

$$\begin{array}{r} 101 \\ 101\,\overline{)11110} \\ 101 \\ \hline 101 \\ 101 \\ \hline 0 \end{array} \qquad \begin{array}{r} 6 \\ 5\,\overline{)30} \end{array}$$

The procedure is as follows:

1. Compare the divisor with the leftmost bits of the dividend.
2. If the divisor is greater, enter a 0 in the quotient and shift the dividend and quotient to the left.
3. Try subtraction again.
4. When the subtraction yields a positive result, i.e., the divisor is less than the bits in the dividend, enter a 1 in the quotient and shift the dividend and the quotient left one position.
5. Return to step 1 and repeat.

Binary division, like binary multiplication, is considerably simpler than the decimal operation.

Subtraction by Complement Addition

If subtraction were performed by the usual method of borrowing from the next higher order, a separate subtraction circuit would be required. Subtraction can be performed, however, by the method of *adding complements*

Binary	Decimal
100110	38
1001	9
11101	29

$A-B$, less borrow			$A-B$, borrow		
⊘	01		⊘	01	
0	01		0	00	
1	10		1	10	

FIGURE 18.1.18 Binary subtraction and corresponding decimal subtraction. The subtraction table is the same as the addition table. The borrow operation is handled analogously to decimal subtraction.

(or adding 1's complements, as the method is also called). By this method, the *subtrahend*, i.e., the number that is to be subtracted, is *inverted*, changing the 0s to 1s and the 1s to 0s. Then the inverted subtrahend is added to the *minuend*, i.e., the number that is to be subtracted from, and an additional 1 is added to find the difference. As an example, consider the subtraction 1101 – 1001. The subtrahend (1001) is first inverted to form the complement (0110). The difference is formed by adding the minuend and the complement of the subtrahend (plus 0001) as follows: 1101 – 1001 = 1101 + 0110 (complement) + 0001 = (1)0011 + 0001 = 0100. Note that in subtraction by complement addition, a leading 1 (in parentheses) in the result (difference) must be suppressed, and that 1 must be added to obtain the result. The result of this operation can be verified by observing that the decimal equivalent of this operation is 13 – 9 = 3 + 1 = 4.

Floating-Point Numbers

In a computer having a fixed number of bits that define a word, the bits represent the maximum size of a numerical value. For example, if 40-bit positions are provided for a word, the maximum decimal number that can be represented is in the order of $1.009 \times 10.^{12}$ Though this number is large, it does not suffice for many applications, especially in science, where a greater range of magnitudes may be routinely encountered. To extend the range of values that can be handled, numbers are represented in floating-point notation. In floating point the most significant digits of the number are written with an assumed radix point immediately to the left of the highest-order digit. This number is called the *fraction.* The intended position of the radix point is identified by a second number, called the *characteristic*, which is appended to the fraction: The characteristic denotes the number of positions that the assumed radix point must be shifted to achieve the intended number. For example, the number 146.754 in floating point might be 146754.03 where 146754 would be equivalent to 0.146754 and the 0.03 would denote a shift of the decimal point three places to the right. In binary notation the number 11011.011 (27.375 in decimal) might be represented in floating point as 11011011.101 with the fraction again to the left of the decimal and the characteristic to the right.

With floating-point addition and subtraction, a shift register is required to align the radix points of the numbers. To perform multiplication or division, the fraction fields are appropriately multiplied or divided and the exponents summed or subtracted, respectively. As with fixed-point addition or subtraction, provision is usually made to detect an overflow condition in the characteristic fields. In some systems provision is made to note when an addition or subtraction occurs with such widely differing characteristics that justification destroys one of the two numbers (by shifting it out the end of a shift register).

Numeric and Alphanumeric Codes

The numeric codes used to represent numerical values previously discussed include the hexadecimal, octal, binary, and decimal codes. In many applications the need arises for the coding of nonnumeric as well as numeric information, and such coding must use the binary scheme. A code embracing numbers, alphabetic characters, and special symbols is known as an *alphanumeric code.*

A widely used code with its roots in the past is the telegraph code (the Baudot code). Other alphanumeric codes have been devised for special purposes. One of the most significant of these, because of its present use and its contribution to the design of other codes, is the Hollerith code, developed in the 1890s. Hollerith's equipment contributed to the development of electromechanical accounting machines that provided the foundation for electronic computers.

Another code of importance in the United States is the American Standard Code for Information Interchange (ASCII) (see Fig. 18.1.19). This code, developed by a committee of the American National Standards Institute (ANSI), has the advantage over most other codes of being *contiguous*, in the sense that the binary combination used to represent alphanumeric information is sequential. Hence alphabetic sorting can be easily accomplished by arithmetic manipulation of the code values.

Codes used for data transmission generally have both data characters and *control characters.* The latter perform control functions on the machine receiving information. In more sophisticated codes, such as

	000	001	010	011	100	101	110	111
0000	NULL	① DC_0	b	0	@	P		
0001	SOM	DC_1	!	1	A	Q		
0010	EOA	DC_2	"	2	B	R		
0011	EOM	DC_3	#	3	C	S		
0100	EOT	DC_4 (STOP)	$	4	D	T		
0101	WRU	ERR	%	5	E	U		
0110	RU	SYNC	&	6	F	V		
0111	BELL	LEM	'	7	G	W	Unassigned	
1000	FE_0	S_0	(	8	H	X		
1001	HT SK	S_1	)	9	I	Y		
1010	LF	S_2	*	:	J	Z		
1011	V TAB	S_3	+	;	K	[		
1100	FF	S_4	,	<	L	\		ACK
1101	CR	S_5	-	=	M	]		②
1110	SO	S_6	.	>	N	↑		ESC
1111	SI	S_7	/	?	O	←		DEL

Identification of control symbols and some graphics

NULL	Null/idle	V_{TAB}	Vertical tabulation	S_0–S_7	Separator (information)
SOM	Start of message	FF	Form feed	b	Word separator (space, normally nonprinting
EOA	End of address	CR	Carriage return	<	Less than
EOM	End of message	SO	Shift out	>	Greater than
EOT	End of transmission	SI	Shift in	↑	Up arrow (exponentation)
WRU	"Who are you?"	DC_0	Device control 1. Reserved for data link escape	←	Left arrow (implies/ replaced by)
RU	"Are you...?"			\	Reverse slant
BELL	Audible signal	DC_1–DC_3	Device control	ACK	Acknowledge
FE_0	Format effector	DC_4 (STOP)	Device control (stop)	②	Unassigned control
HT	Horizontal tabulation	ERR	Error	ESC	Escape
SK	Skip (punched card)	SYNC	Synchronous idle	DEL	Delete/idle
LF	Line feed	LEM	Logical end of media		

FIGURE 18.1.19 The ASCII code has a contiguous alphabet, so that numeric ordering permits alphabetic sorting.

ASCII, these control functions are greatly extended and hence are applicable to machines of different design.

Other Numeric Codes. Not all numeric information is represented by binary numbers. Other codes are also used for numeric information in special applications. Figure 18.1.20 shows a widely used code called the *reflected* or *Gray* code. It has the property that only 1 b is changed between any two successive values, irrespective of number size. This code is used in digital-to-analog systems since there is no need for propagation of carry integers in sequential counting as in a binary code.

Decimal	Gray code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

FIGURE 18.1.20 The Gray code, used in analog-to-digital encoding systems. There is only a 1-b change between any two successive integers.

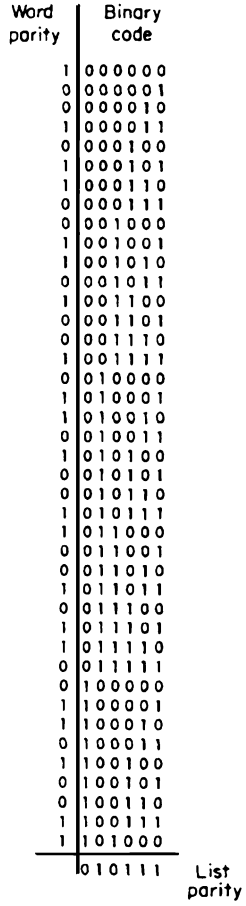

FIGURE 18.1.21 Two-dimensional parity checking, in which a single error can be corrected and a triple error detected.

Error Detection and Correction Codes

The integrity of data in a computer system is of paramount importance because the serial nature of computation tends to propagate errors. Internal data transmission between computer-system units takes place repeatedly and at high speed. Data may also be sent over wires to remote terminals, printers, and other such equipment. Because imperfections of transmission channels inevitably produce some erroneous data, means must be provided to detect and correct errors whenever they occur.

A basic procedure for error detection and correction is to design a code in which each word contains more bits than are needed to represent all the symbols used in a data set. If a bit sequence is found that is not among those assigned to the data symbols, an error is known to have occurred.

One such commonly used error-detection code is called the *parity check.* Suppose that 8 bits are used to represent data and that an additional bit is reserved as a check bit. A simple electronic circuit can determine whether an even or odd number of 1 bits is included in the eight bit positions. If an even number exists, a 1 can be inserted in the check position. If an odd number of 1s exist, the check position contains a 0. As a result all code words must contain an odd number of 1 bits. If a 9-b sequence is found to contain an even number of 1s, an error can be presumed.

Decimal digit	Position 1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	0	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1

Parity checks — 8, 4, 2, 1 code

FIGURE 18.1.22 A Hamming code. The parity bit in column 1 checks parity in columns 1, 3, 5, and 7; the bit in column 2 checks 2, 3, 6, and 7; and the bit in column 4 checks 4, 5, 6, and 7. The overlapping structure of the code permits the correction of a single error or the detection of single or double errors in any code word.

There are limitations in the use of the simple parity check as a mechanism for error detection, since in many transmission channels and storage systems there is a tendency for a failure to produce simultaneous errors in two adjacent positions. Such an error would not be detected since the parity of the code word would remain unchanged.

To increase the power of the parity check, a list of code words in a two-dimensional array can be used, as shown in Fig. 18.1.21. The code words in the horizontal dimension have a parity bit added, and the list in the vertical dimension also has an added parity bit, in each column. If one bit is in error, errors appear in both the row and the column. If simultaneous errors occur in two adjacent positions of a code word, no parity error will show up in that row, but the column checks will detect two errors. This code can detect any 3-b errors.

It is possible to design codes that can detect directly whether errors have occurred in two bit positions in a single code word. Figure 18.1.22 shows such a code, an example of a *Hamming* code. The code positions in columns 1, 2, and 4 are used to check the parity of the respective bit combinations. Two code words of a Hamming code must differ at three or more bit positions, and therefore any 2-b error patterns can be detected. The pattern formed by the particular parity bits that show errors indicates which bit is in error in the case of a single bit failure.

In general, if two code words must differ at D or more bit positions, the code can detect up to $D - 1$ bit errors. For $D = 2t + 1$, the code can detect $2t$ bit errors or correct t bit errors.

COMPUTER ORGANIZATION AND ARCHITECTURE

Introduction

Over the past 40 years great progress has been made as component technology has moved from vacuum tubes to solid-state devices to large-scale integration (LSI). This has been achieved as a result of increased understanding of semiconductor materials along with improvements in the fabrication processes. The result has been significant enhancement in the performance of the logic and memory components used in computer construction along with significant reductions in cost and size. Figure 18.1.23, for example, indicates the reduction in volume of main memory that has occurred in the last 40 years. The volume required to store 1 million characters has been reduced by a factor of 3 million in that period. Smilarly, during that same period, the system cost to execute a specific mix of instructions has also decreased by a factor of 5000. Integrated circuit manufacturers are able to incorporate millions of transistors in the area of a square inch. Large-scale integration (LSI) and VLSI have led to small-size, lower cost, large-memory, ultrafast computers ranging from the familiar PC to the high-performance, high-priced supercomputers. There is no reason to expect that this progress will not continue into the future as new technology improvements continue to occur.

These advances in component technology have also had a major impact on computer organization and its realization. Functions and features that were too expensive to be included in earlier designs are now feasible, and the trade-offs between software and hardware need to be reevaluated as hardware costs continue to decrease. New approaches to computer organization must also be considered as technology continues to improve. The advances in component technology also have had a major impact on such aspects of computer realization as packaging, coding, and power.

Basic Computer Organization

The basic organization of a digital computer is shown in the block diagram of Fig. 18.1.24. This structure was proposed in 1946 by von Neumann. It is a tribute to his genius that this design, which was intended for use in

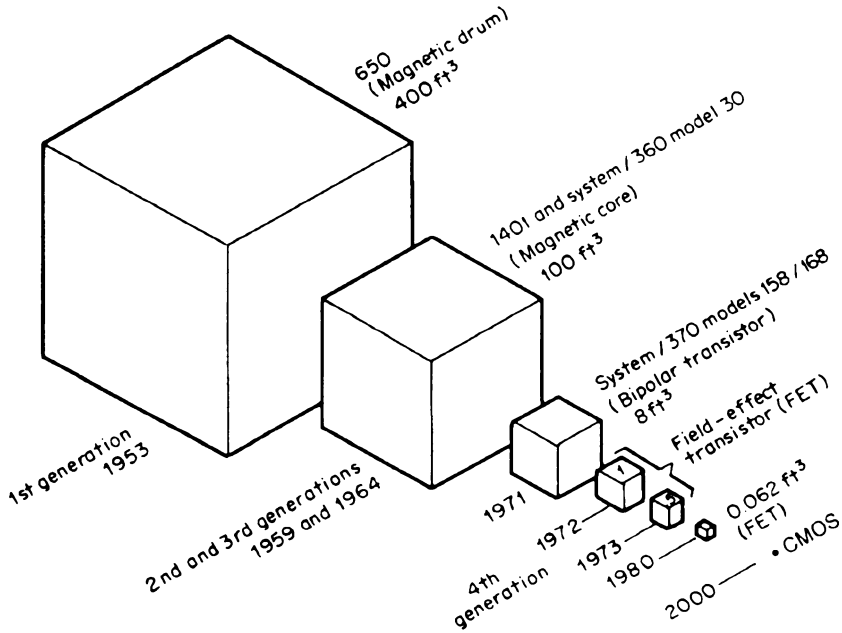

FIGURE 18.1.23 The reduction by a factor of 6400 in memory size from the first- to the fourth-generation families of IBM computers.

solving differential equations, has also been applicable in solving other types of problems in such diverse areas as business data processing and real-time control. Von Neumann recognized the value of maintaining both data and computer instructions in storage and in being able to modify instructions as well as data. He recognized the importance of branch or jump instructions to alter the sequence of control of computer execution. His contributions were so significant that the vast majority of computers in use today are based on his design and are called von Neumann computers.

The four basic elements of the digital computer are its *main storage, control unit, arithmetic-logic unit* (ALU), and *input/output* (I/O). These elements are interconnected as shown in Fig. 18.1.24. The ALU, or *processor*, when combined with the control unit, is referred to as the *central processing unit* (CPU).

Main storage provides the computer with directly addressable fast-access storage of data. The storage unit stores programs as well as input, output, and intermediate data. Both data and programs must be loaded into main storage from input devices before they can be processed.

The control unit is the controlling center of the computer. It supervises the flow of information between the various units. It contains the sequencing and processing controls for instruction decoding and execution and for handling interrupts. It controls the timing of the computer and provides other system-related functions.

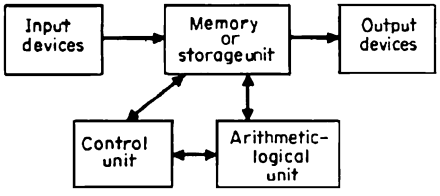

FIGURE 18.1.24 Block diagram of a digital computer illustrating the main elements of the von Neumann architecture.

The ALU carries out the processing tasks specified by the instruction set. It performs various arithmetic operations as well as logical operations and other data processing tasks.

Input/output devices, which permit the computer to interact with users and the external world, include such equipment as card readers and punches, magnetic-tape units, disc storage units, display devices, keyboard terminals, printers, teleprocessing devices, and sensor-based devices.

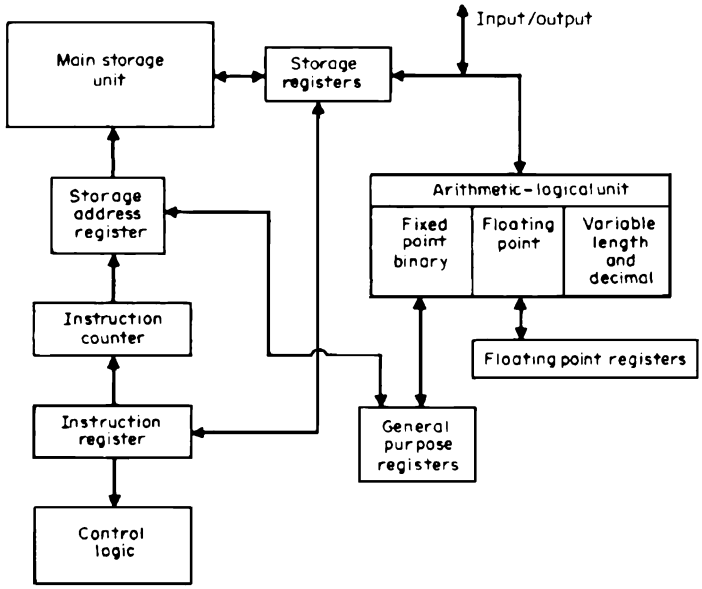

FIGURE 18.1.25 Basic structure of a digital computer showing a typical register organization and interconnections.

Detailed Computer Organization

The block diagram of Fig. 18.1.25 provides an overview of the basic structure of the digital computer. Computer systems are complex, and block diagrams cannot describe the computer in sufficient detail for most purposes. One is therefore forced to go to lower levels of description. There are at least five levels[66a] that can describe the implementation of a computer system:

1. Processor-memory-switch (block-diagram) level
2. Programming level (including the operating system)
3. Register-transfer level
4. Switching-circuits level
5. Circuit or realization level

Each of these levels is an abstraction of the levels beneath it. A number of computer-hardware description languages have been developed to represent the components used in each level along with their modes of combination and their behavior.

A *register* is a device capable of receiving information, holding it, and transferring it as directed by control circuits. The actual realization of registers can take a number of forms depending on the technology used. Registers store data temporarily during a program's execution. Some registers are accessible to the user through instructions.

Registers are found in every element of the computer system. They are an integral part of main storage, being used as storage registers to contain the information being transferred from memory (read) or into memory (write) as well as storage address registers (SAR) to hold the address of the location in storage involved in the information transfer. In the control unit, the instruction (or program) counter contains the storage address of the instruction to be executed while the instruction register holds the instruction being decoded and executed.

In the ALU internal registers are used to hold the operands and partial results while arithmetic and logical operations are being performed. Other ALU registers, called *general-purpose registers*, are used to accumulate

the results of arithmetic operations but can be used for other purposes such as indexing, addressing, counting looping operations, or subroutine linkage. In addition, floating-point registers may be provided to hold the operands and accumulate the results of arithmetic operations on floating-point numbers. Of all the registers mentioned, only the general-purpose and floating-point registers are accessible to program control and to the programmer. Figure 18.1.25 shows the primary registers and their interconnections in the basic digital computer.

At the register transfer level of abstraction one can describe a digital computer as a collection of registers between which data can be transferred. Logical operations can be applied to the data during the transfers. The sequencing and timing of the transfers are scheduled and controlled by logic circuits in the control unit.

The data transferred between registers within the computer consist of groups of binary digits. The number of bits in the group is determined by the computer architecture and in particular by the organization of its main storage. Main storage is structured into segments, called bytes, and each storage word is uniquely identified by a number, called its *address*, assigned to it. A byte consists of 8 binary bits and is the standard for describing memory elements. Many computers read groups of bytes (2, 4, 6, 8) from memory in one access. A group of bytes is referred to as a word, which can vary in length from one computer to another. A memory access is a sequence to read data from memory or store it into memory. The 1s and 0s can be interpreted in various ways. These bit patterns can be interpreted as: (1) a pure binary word, (2) a signed binary number, (3) a floating-point number, (4) a binary-coded decimal number, (5) data characters, or (6) an instruction word.

In a signed binary number the high-order (leftmost) bit indicates the sign of the number. If the bit is 0, the number is positive; a 1 indicates it is negative. Thus

0bbbbbbb represents a positive 7-bit number

1bbbbbbb represents a negative 7-bit number

A negative number is carried in 2's-complement (inverted) form. For example,

$$11111110_2 = 2^{10}$$

A binary-coded decimal code uses 4 b to represent a decimal digit. It uses the binary digit combinations for 0 to 9; combinations greater than 9 are not allowed. Thus

$$0101_2 = 5_{10} \quad 1010_2 = \text{illegal}$$

The sign of the decimal number can be indicated in several ways. One technique uses the low-order bits to indicate the sign; for example,

bbbbbbbbbbbb1100 represents a positive 3-digit number

bbbbbbbbbbbb1011 represents a negative 3-digit number

Thus, $0001010100111011_2 = -153_{10}$.

For external communication, as well as text processing and other nonnumeric functions, the digital computer must be able to handle character sets. The byte has been accepted as the data unit for representing character codes. The two most common codes, described earlier, are the American Standard Code for Information Interchange (ASCII) and the Extended Binary-Coded Decimal Interchange Code (EBCDIC). The 16-b word 1100011110010110 coded in EBCDIC represents the two-letter word "go." (See Figs. 18.1.4, 18.1.5, and 18.1.9).

The *instruction word* is composed of two major parts, an *operation part* and an *operand part.* The length of the instruction word is determined by the computer architecture. Some computers have a single format for their instruction words and thus a single length, whereas other computers have several formats and several different lengths. The operation part consists of the operation code that describes the particular operation to be performed by the computer as a result of executing that instruction. The operand part usually contains the addresses of the two operands involved in the operation. For example, the RR (register-to-register) instruction format in the System/370 is

Op Code	Reg 1	Reg 2
bbbbbbbb	bbbb	bbbb
0 7	8 11	12 15

The instruction 1AB4 (in hexadecimal), for example, instructs the computer to add the contents of register 11 to the contents of register 4 and to put the resulting sum in register 11, replacing its original contents. Most computers use the two-address instruction with three sections: The first section consists of the op-code and the second and third sections each contain an address of an operand. Different computers use these addresses differently. In some cases, the operand section is used as a modifier or to extend the instruction length. A single-operation computer has two sections—an op code and an operand, usually a memory address, with the accumulator the implied source or destination.

Two facts should be noted. First, the discussion thus far may have implied that digital computers can deal only with fixed-length words. That is true for some computers, but other families of computers can also deal with variable-length words. For these, the operand part of the instruction contains the address of the first digit or character of each variable-length word plus a measure of its length, i.e., the number of characters it contains. The second fact is that it is impossible to distinguish between the various data representations when they are stored. For example, there is nothing to indicate whether a word of memory contains a binary number or a binary-coded decimal (BCD) number. Programmers must make the distinction in the programs they develop and not attempt meaningless operations such as adding a binary number to a decimal number. The only way the computer distinguishes an instruction word from other data words is by the time when, as discussed in the next section, it is read from storage into the control unit.

Instruction Execution

The digital computer operates in a cyclic fashion. Each cycle is called a *machine cycle* and consists of two main subcycles, the *instruction* (I) *cycle* (sometimes called the *fetch cycle*) and the *execution* (E) *cycle.* During the machine cycle, the following basic steps occur in sequence (see Fig. 18.1.25):

1. The cycle begins with the I cycle:
   a. The contents of the instruction counter are transferred to the storage address register (SAR). (The instruction counter holds the address of the next instruction to be executed.)
   b. The specified word is transferred from storage to the instruction register. (The control unit assumes that this storage word is an instruction.)
   c. The contents of the instruction register are decoded by logical circuits in the control unit. This identifies the type of operation to be performed and the locations of the operands to be used in the operation.

2. At this point, the E cycle begins:
   a. The specified computer operation is performed using the designated operands, and the result is transferred to the location indicated by the instruction.
   b. The instruction counter is advanced to the address of the next instruction in the sequence. (If a branch, or change in execution control sequence, is to occur, the contents of the instruction counter are replaced by an address as directed by the instruction currently being executed.)

3. At this point, the I cycle is repeated.

To indicate in more specific terms what happens in the CPU during instruction execution it is necessary to go to the switching level of description. The following paragraphs describe of the operations of the ALU and the control section in more detail.

Arithmetic Logic Unit

The ALU performs arithmetic and logical operations between two operands, such as OR, AND, EXCLUSIVE-OR, ADD, MULTIPLY, SUBTRACT, or DIVIDE. The unit may also perform operations such as INVERT on only one operand, and it tests for minus or zero and forms a complement.

Adders and multipliers are at the heart of the ALU. In Fig. 18.1.26 one bit position of an ALU is shown as part of a wider data path. One latch (part of a register *A*) feeds an AND circuit that is conditioned by a CONTROL *A*. The output feeds INPUT *A* of the adder circuits. One latch of register *B* is also ANDed with CONROL *B* and feeds the other input into the adder. *A* true-complement circuit is shown on the *B* line. This latter circuit has to do with subtraction and can be assumed to be a direct connection when adding. Each adder stage is a combinatorial circuit that accepts a carry from the stage representing the next lower digit in the binary number (assumed to be on the right). The collection of outputs from all adder stages is the sum. This sum is ANDed into register *D* by CONTROL *D*.

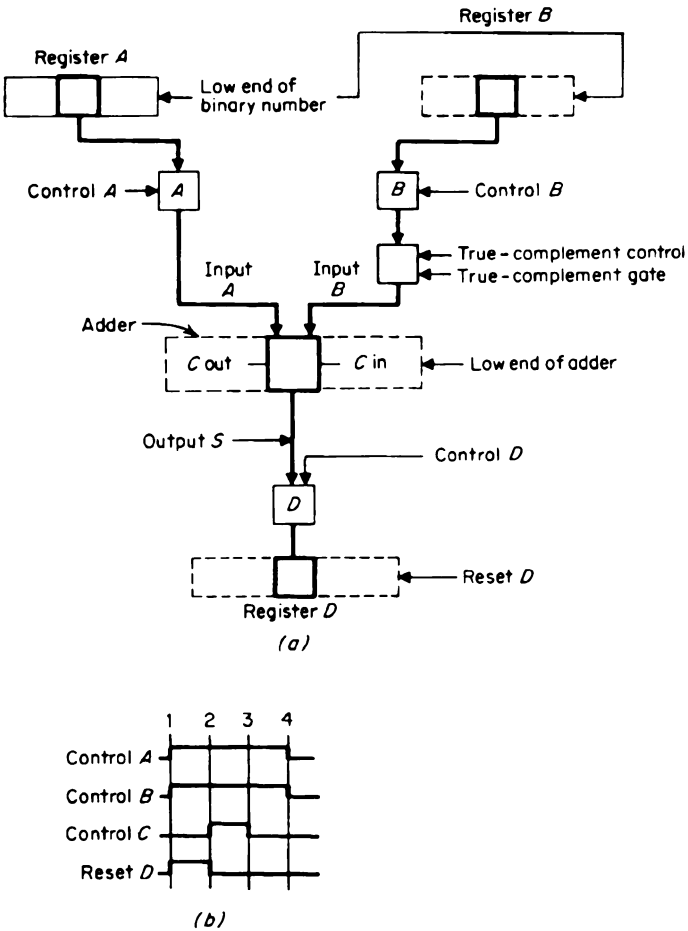

FIGURE 18.1.26 Basic addition logic. The heart of the ALU is the adder circuit shown in functional form in (*a*). The control section applies the appropriate time sequence of pulses (*b*) on the control to perform addition. Heavy lines indicate a repeat of circuitry in each bit position to form a machine adder.

All bit positions of each of the registers are gated by a single control line. If the gate is closed (control equal to 0), all outputs are 0s. If the gate is open (control equal to 1), the bit pattern appearing in the register is transmitted through the series of AND circuits to the input of the adders. Thus, a gate is a two-way AND circuit for each bit position. The diagram of Fig. 18.1.26 illustrates all positions of an *n*-position adder since all positions are identical. In such a case heavy lines, as shown, indicate that this one line represents a line in each bit position.

Binary Addition

At the outset of an addition, it is assumed that registers *A* and *B* (Fig. 18.1.26*a*) contain the addends. An addition is performed by pulsing the control lines with signals originating in the control section of the CPU. Time is assumed to be metered into fixed intervals by an oscillator (clock) in the control section. These time slots are numbered for easier identification (Fig. 18.1.26*b*). At time 1 the inputs are gated to the adder, and the adders begin to compute the sum. At the same time, register *D* is *reset* (all latches set to 0). At time 2 the outputs of the adders have reached steady state, and control line *D* is raised, permitting those bit positions for which the sum was 1 to set the corresponding latches in register *D*. Between times 2 and 3, the result is latched up in register *D*, and at time 3, control *D* is lowered. Only after the result is locked into *D* and cannot change, may control *A* and *B* be lowered. If they were lowered earlier, the change might propagate through the adder to produce an incorrect answer.

The length of the pulses depends on the circuits used. The times from 2 to 3 and 3 to 4 are usually equal to a few logic delays (time to propagate through an AND, OR, or INVERTER). The time from 1 to 2 depends on the length of the adder and is proportional to the number of positions in a parallel adder because of potential carry-propagation times. This delay can be reduced by *carry look-ahead* (some-times called *carry bypass*, or *carry anticipation*).

Binary Subtraction

Subtraction can be accomplished using the operation of addition, by forming the complement of a number. Negative numbers are represented throughout the system in complement form. To subtract two numbers such as *B* from *A*, a set of logic elements may be put into the line shown in Fig. 18.1.26*a* an input *B*. Using 2's complement, the sign of a number is changed by complementing each bit and adding 1 to the result. The inversion of the bit is performed by the logic element interposed on the input *B* line in Fig. 18.1.26*a* known as a *true-complement* (T/C) *gate.* This unit gates the unmodified bit if the control is 0 and inverts each bit if the control is 1. The boolean equation for the output of the T/C gate is

$$\text{Output} = \overline{\text{T/C}} \cdot B + \text{T/C} \cdot \overline{B}$$

The T/C gate is a series of EXCLUSIVE-ORS with one leg, common to all bit positions, connected to the T/C control line. The other leg of each EXCLUSIVE-OR is connected to one bit of the circuit containing the number to be complemented.

The T/C gate produces the 1's complement; a 1 must be added in the low-bit position to produce the true complement. The low stage of an adder may be designed to have an input for a carry-in, designed to accommodate the 1 bit automatically produced from the high-order position of the T/C gate. Such a logical interconnection accomplishes the required 1 input for a true-complement system when a positive number *B* is subtracted from a positive number and is called an *end-around carry.* Consistency of operation is obtained by entering the appropriate high-order T/C gate into the low-order carry position.

Decimal Addition

In some systems the internal organization of a computer is such that decimal representations are used in arithmetic operations. In BCD, a conventional binary adder can be used to add decimal digits with a small amount of additional hardware. Adding two 4-b binary numbers produces a 4- or 5-b binary result. When two BCD

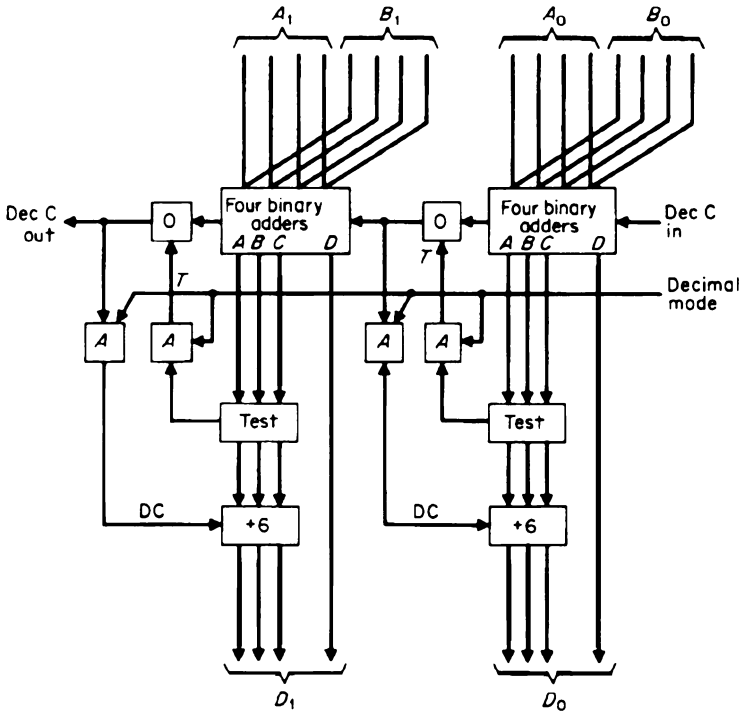

FIGURE 18.1.27 Binary adder with decimal-add feature. If the input of the binary add is 1010 or greater, the addition of a binary 6 produces the correct result by modulo-10 addition.

numbers are added, the result is correct if it lies in the range 0 to 9. If the result is greater than 9, that is the resulting bit pattern is 1010 to 10010, the answer must be adjusted by adding 6 to that group of 4 b, this number being the difference between the desired base (10) and the actual base ($2^4 = 16$). The binary carry from a block of 4 b must also be adjusted to generate the appropriate decimal carry.

The circuits to accomplish decimal addition are shown in Fig. 18.1.27. A test circuit generates an output if the binary sum is 1010 or greater. This output causes a 6 to be added into the sum and also is ORed with the original binary carry to produce a decimal carry. The added circuits needed to perform decimal additions with a binary adder represent one-half to two-thirds of the circuits of the original binary adder.

Decimal Subtraction

In most computers that provide for decimal operation, decimal numbers are stored with a sign and magnitude. To perform subtraction the true complement is formed by subtracting each decimal digit in the number from 9 and adding back 1. Once the complement is formed, addition produces the desired difference.

In a machine that provides for decimal arithmetic a *decimal true-complement* switch may be incorporated in each group of four BCD bits to form the complement. As with binary, provision must be made for the addition of an appropriate low-order digit and for the occurrence of overflows.

Shifting

All computers have shift instructions in their instruction repertoire. Shifting is required, for example, for multiply and divide operations.

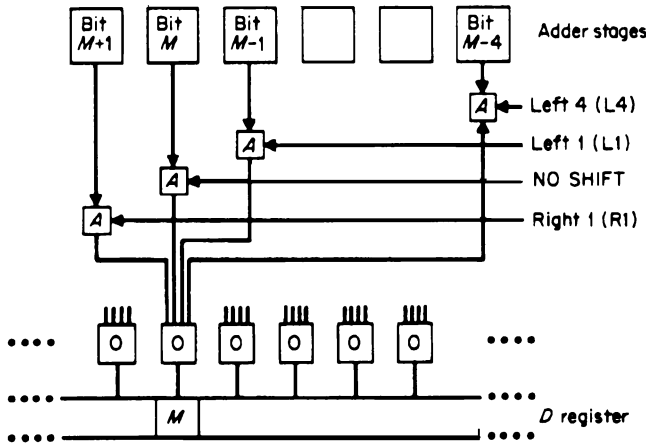

FIGURE 18.1.28 Shift gates. In many systems shifting is accomplished in conjunction with the output of the adder, i.e., position shifts can be accomplished with only one circuit delay.

The minimum is a shift of one position left or right, but most computers have circuits permitting a shift of one or more bits at a time, that is, 1, 4, or 8.

In shifting, a problem arises with the bit(s) shifted out at the end of the register and the new (open) bit position(s). Those shifted out at one end can be inserted in the other end (referred to as *end-around shift* or *circulate*), or they can be discarded or placed in a special register. The newly created vacancies can be filled with all 0s, all 1s, or from another special register.

In a typical computer, shifting is not performed in a shift register but with a set of gates that follow the adders. The outputs of the shift gates are connected to the output register, with an offset by providing a separate gate for each distance and direction of shift. One bit position of the output register can therefore be fed from several adder outputs, but only one gate is opened at a time, as in Fig. 18.1.28. The pattern shown is repeated for every bit position.

Multiplication

Figure 18.1.29 shows a possible data-flow system for multiplication, i.e., the data flow of the adder shown in Fig. 18.1.26 with the addition of register C to hold the multiplier and the shift registers as shown in Fig. 18.1.28. An extra register E holds any extra bits that might be generated in the process of multiplication. Register E in the particular system shown also receives the contents of C after transmission through C's shift register.

Computers also have an extension of the shift instruction called *rotate.* A shift usually occurs as a result, but the leading bit of the shift right or left is rotated to the opposite end of the register. This feature can be used for bit detection without losing data.

The process of binary multiplication involves decoding successive bits in the multiplier, starting with its lowest-order position. If the bit is a 1, the multiplier is added into an accumulating sum; if 0, no addition takes place. In either case the sum is moved one position to the right and the next higher-order bit in the multiplier considered.

In Fig. 18.1.29 the multiplier is stored in register C, the multiplicand in A, and all others are reset to zero. If the low-order position of C is a 1, the contents of B and C are added and shifted one position to the right into register D. After the addition, register C is shifted one position to the right and stored in register E. If the low-order bit in C had been a zero, only register B would have been gated into the adders; i.e., the addition of the contents of A would have been suppressed, but subsequent operations would have remained the same.

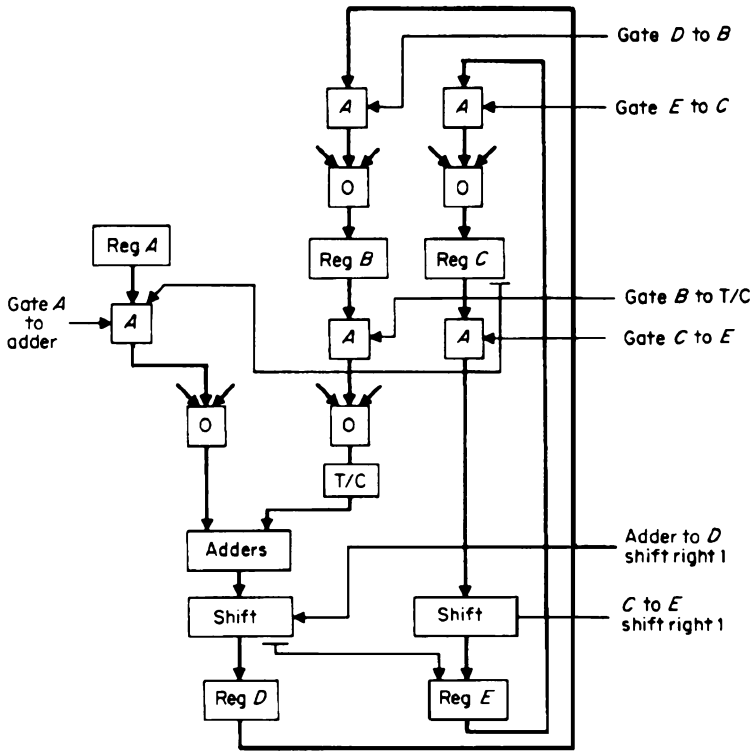

FIGURE 18.1.29 Data flow for multiplication.

Each add and shift operation subsequent to an add may generate low-order bits that may be shifted into register *E*, since as the contents of *C* are shifted to the right, unused positions become successively available. After the add and shift cycles, registers *D* and *E* are transferred into *B* and *C*, respectively, and the process is repeated until all positions of the multiplier are used. The content of *D* and *E* is the product.

Division

To provide for the division of two numbers the functions of the registers in Fig. 18.1.29 must be rearranged as shown in Fig. 18.1.30. A gating loop is provided from the shift register to register *A*. Initially the divisor is placed in register *B* and the dividend in *A*. The T/C gate is used to subtract *B* from *A*, and if the result is zero or positive, a 1 is placed in the low-order position of register *E*. If the result is negative, a 0 is placed in *E* and the input from *B* ungated to reset to the contents of *A*. The shift register then shifts the output of the adder one position to the right and gates it back to *A*. *E* is gated through *C* and shifted on to the right. The whole process is repeated until the dividend is exhausted. *E* contains the quotient and *A* the remainder.

Floating-Point Operations

In some applications, it is convenient to represent numerical values in floating-point form. Such numbers consist of a *fraction* that represents the number's most significant digits, in a portion of the field of a word in the machine, and a *characteristic* in the remaining portion. The characteristic denotes the decimal-point position

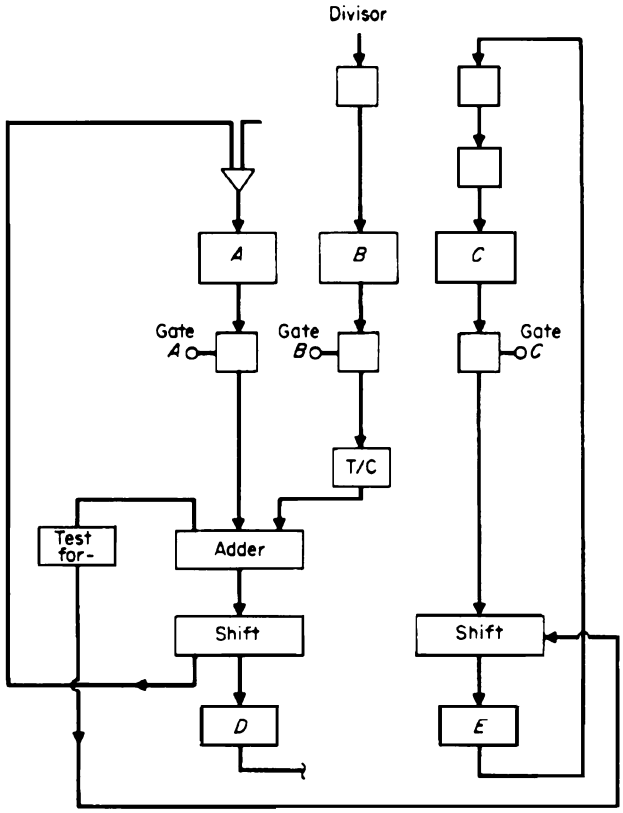

FIGURE 18.1.30 Data flow for division. *B* holds the divisor and *A* the dividend. A trial subtraction is made of the high-order bits of *B* from *A*, and if the results are 0 or positive, a 1 is entered into the lower-order bit of *E* and the result of the subtraction shifted left and reentered into *A* (with gate *A* closed). If the result had been negative the *B* gate would have closed to negate the subtraction, and a 0 would have been entered into *E*. The output of the adder would then have been shifted left one position and reentered in *A*.

relative to the assumed decimal point in the characteristic field. In floating-point addition and subtraction, justification of the fractions according to the contents of the characteristic field must take place before the operation is performed; i.e., the decimal points must be lined up.

In an ALU such as in Fig. 18.1.29, the operation proceeds in the following way. Two numbers *A* and *B* are placed in registers *A* and *B*, respectively. The control selection then gates only the characteristic fields into the adder, in a subtract mode of operation. The absolute difference is stored in an appropriate position of the control section. Controlled by the sign of the subtraction operation, the fraction of the least number is placed through the adder into the shift register. The control section then shifts the fraction the required number of positions, i.e., according to the stored difference of characteristic fields, and places the result back in the appropriate register. Addition or subtraction can then proceed according to the generating machine instruction.

This procedure is costly of machine time. Instead of using the ALU adder for the characteristic-field difference, provision can be made for subtraction of the characteristics in the control section. In such a case, only the characteristics *A* and *B* need be entered into registers *A* and *B* and the lesser number can be placed in *B* so that shifting can be accomplished by the circuits normally used in multiplication.

Control Section

The control section is the part of a CPU that initiates, executes, and monitors the system's operations. It contains clocking circuits and timing rings for opening and closing gates in the ALU. It fetches instructions from main storage, controls execution of these instructions, and maintains the address for the next instruction to be executed. The ALU also initiates I/O operations.

Basic Timing Circuits

Basic to any control unit is a continuously running oscillator. The speed of this oscillator depends on the type of computer (parallel or serial), the speed of the logic circuits, the type of gating used (the type of registers), and the number of logic levels between registers. The oscillator pulses are usually grouped to form the basic operating cycle of the computer, referred to as *machine cycles.* In this example four pulses are combined into such a group.

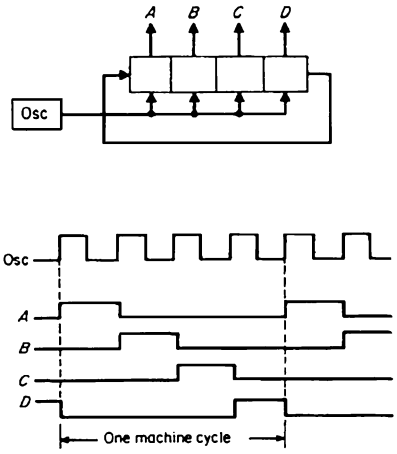

FIGURE 18.1.31 Oscillator and ring circuit. Many control functions can be performed by using an oscillator in conjunction with a timing ring that sequentially sends signals on separate lines, in synchronism with the oscillator.

In Fig. 18.1.31 an oscillator is shown that drives a four-stage ring. At any one time, only one stage is on. Suppose an addition is to be performed between registers *A* and *B* and the result is to be placed back in *B*. The addition circuitry described in Fig. 18.1.29 uses three registers, two for the addends (registers *A* and *B*), and one for temporary storage (register *D*). The operation to be performed is to add the content of register *A* to the content of register *B* and store the result in register *B*. Register *D* is required for temporary storage since if *B* were connected back on itself, an unstable situation would exist; i.e., its output (modified by *A*) would feed its input.

The operation of the four-stage clock ring in controlling the addition and transfer is shown in Fig. 18.1.32. Action is initiated by the coincidence of an ADD signal and clock pulse *A*, which starts the *add ring.* This latter ring has two stages and advances to the next stage upon occurrence of the next *A* clock pulse. The timing chart in Fig. 18.1.32 describes one sequence of actions required and the gates needed for the addition. The circuit diagram shows a realization of these gates. Each register is reset before it is reused, and all pulses are derived from the four basic clock pulses shown in Fig. 18.1.31. The add ring initiates the add by opening the gates between registers *A* and *B* and the adder. The add latch is then reset, *D* is reset, the ring stage transferred, and so forth. An ADD-FINISHED signal is furnished, and may be used elsewhere in the system.

In the timing diagram, it is assumed that the time required for transmission through the adder is about one machine cycle and that one clock cycle is sufficient for the signal to propagate through the necessary gating and set the information in the target register. These times must include the delay in the circuits and any signal-propagation delay.

Control of Instruction Execution

The following approach for the design of a control section is straightforward but extravagant of hardware. For each instruction in the computer a timing diagram is developed similar to the one shown for the add operation in the previous section. These timings are implemented in rings that vary in length, according to the complexity of the instruction. The concept is simple and has been widely used, but it is costly. To reduce cost, rings are used repeatedly within one instruction and/or by several instructions. Subtraction, for example, might use the ADD ring, except for an extra latch that might be set at ring time 1, clock time *A* (denoted 1.*A*) and reset at 2.*C*. This new latch feeds the T/C gates. Another latch that might be added to denote decimal arithmetic would also be set at 1.*A* time and reset at 2.*C* time. The addition of two latches and a few additional ANDS and ORS permits the elimination

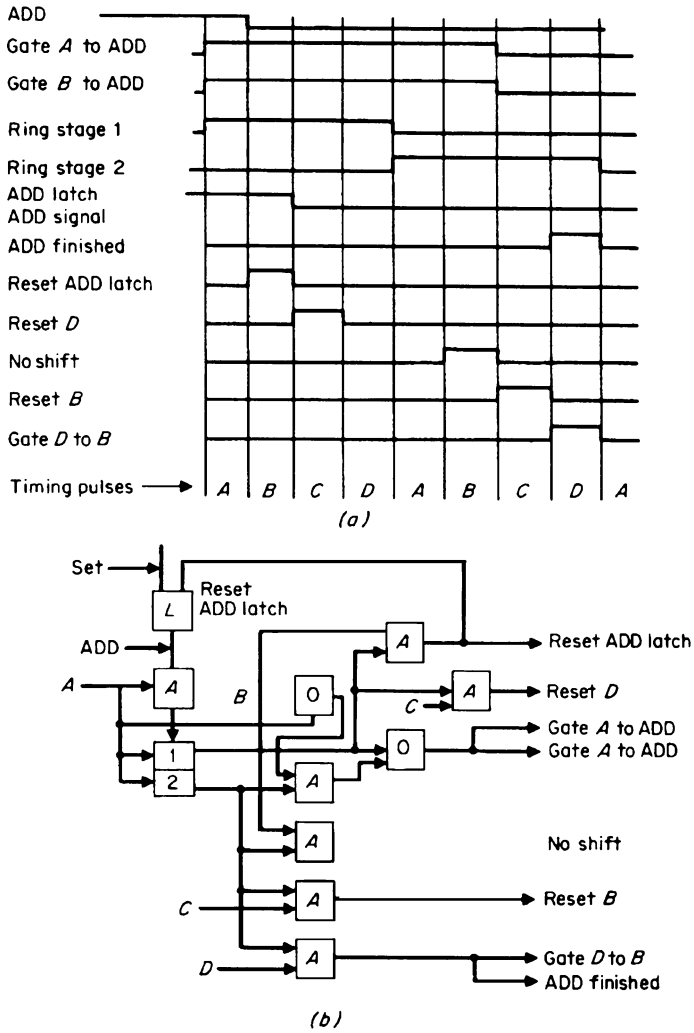

FIGURE 18.1.32 Control circuit (*b*) and timing chart (*a*) for addition. The timing ring shown in Fig. 18.1.31 to control the adder circuit shown in Fig. 18.1.29 with the help of additional switching circuits.

of three two-stage rings and associated logic. Further reductions in the number of required circuits can be achieved by considering the iterative nature of some instructions, such as multiplication, division, or multiple shifting.

Controls Using Counters (Multiplication)

To exemplify this approach multiplication is considered. Multiplication can be implemented as many add-shift cycles, one per digit in the multiplier (say N). The control for such an instruction can be implemented using one $2N + 1$ position ring, the first position initializing and the next $2N$ positions for N add-shift cycles (two positions are needed per add cycle). Such an approach requires not only an unnecessarily long ring but is relatively inflexible for other than N-b multipliers.

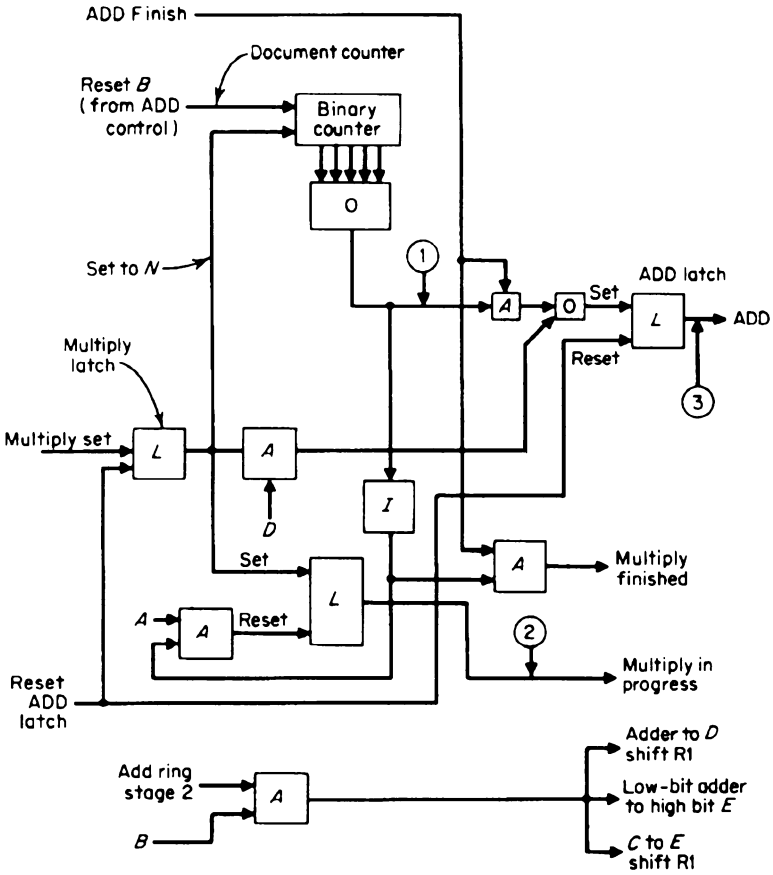

FIGURE 18.1.33 Control circuit for multiplication. Repetitive operations such as add and shift are combined in cyclical fashion. The timing diagram of Fig. 18.1.32*a* is assumed.

The alternate approach, requiring considerably less hardware, uses the basic operation in multiplication, an add and a shift. Therefore a multiply can be implemented by using the controls for the add-shift instruction, plus a binary counter, plus some assorted logic gates to control the gates unique to the multiply, as in Fig. 18.1.33. In the figure some of the less important controls needed for multiply have not been included to simplify the presentation. For example, the NO SHIFT signal for add must be conditioned with a signal that multiply is not in progress, and during multiplication an ADD-FINISHED signal should be ignored (nor start an I cycle), the reset *B* also resets *C*, the reset *D* also resets *E*, gate *D* to *B* also gates *E* to *C*, gating of *A* to the adder is conditioned on the last bit of *C*, and so forth.

In Fig. 18.1.33 the action is started by raising the MPY line that sets a *multiply* latch. This in turn sets the binary counter to the desired cycle count. Also set is the ADD-LATCH, and the addition cycles start. When the counter goes to 0, the ADD-LATCH is no longer set and the MULTIPLY-FINISHED signal is raised.

Microprogramming

The control of the E cycle, in the preceding descriptions, is performed by a set of sequential circuits designed into the machine. Once an execution is initiated, the clocking circuits complete the operation, through wired

paths in a specific manner. An alternative method of design for a control unit is *microprogramming.* The concept is not sharply defined but has as its objective implementation of the control section at low cost and with high flexibility.

In many cases it is desirable to design compatible computers, i.e., units with the same instruction set, with widely varying performance and cost. To provide a slower version at a lower cost, the width of the data path is reduced to lower circuit counts in the ALU. On the other hand, to operate with the same instruction set, the reduced data-path width usually implies more iterative operations in the control section, at added cost.

Considerable investment is required for programming development, and normally such systems run only on computers with identical instruction sets. If appropriate flexibility is provided, one computer can mimic the instruction set of another. Microprogramming provides for such operations. The process by which one system mimics another is called emulation.

Control lines are activated not by logic gates in conjunction with counters but by words in a storage system that are translated directly into electric signals. The words selected are under the control of the instruction decoder, but the sequence of words may be controlled by the words themselves by provisions of a field that does not directly control gates but specifies the location of the next word. The name given to these control words is *microinstruction* as opposed to machine instruction or instruction.

When two computers are designed for the same instruction set but with different data-path widths, the microinstruction sets of the two computers are radically different. For the small computer, the program for a given machine instruction is considerably longer than that for the large computer. The same instructions can be used in both computers. The difference in control-system cost between the two is not large. Although the microprogram is longer with the smaller computer, the difference is in the number of storage places provided, not in the control-section hardware.

The design of the sequence of microprogramming words is conceptually little different from other programming. The microprogram implementation, however, requires a thorough knowledge of a machine at the gate level. A *microinstruction counter* is used to remember the location of the next microinstruction, or a field can be provided to allow branching or specification of a next instruction.

A microprogram generally does not reside in main store but in a special unit called a *control store.* This may not be addressable by the user, or it may be a separate storage unit. In many cases, the microprogram is stored in a read-only store (ROS); i.e., it is written at the factory. ROS units are faster and may be cheaper per bit of storage and do not require reloading when power is applied to the computer. Alternatively the microprogram may be stored in medium that can be written into, called a *writable control store* (WCS). By reloading the WCS, entirely different macroinstruction can be implemented, using the same microinstruction set in a different microprogram. By such means emulation is achieved at minimal expense.

CPU Microprogramming

Figure 18.1.34 shows an ALU and Fig. 18.1.35 a control section with microprogram organization. The microinstructions embodied in these two units are shown in Table 18.1.2.

To simplify the program several provisions have been made:

1. Each microinstruction contains the control-store address of the next microinstruction. If omitted, the address in one microinstruction is the current address of the one written just below it. It may *not* be next in numeric sequence.
2. An asterisk at the beginning of a line in a program indicates a comment. This means that the entire line contains information about the program and does not translate into a microinstruction.
3. To simplify the drawing, a gate in a path is indicated by placing an X in the line and omitting from the drawing any control lines controlling these gates. It is also assumed that where two lines join, OR circuits are implied.
4. Rather than listing a numeric value for each field, a shorthand description for the desired action is invented. All actions not so described are assumed to be zero. For example, *A* to ADD implies that register *A* is gated to the adder, or T/C means raise T/C gate. These charges do not in any way modify the concept of microprogramming but make the result more readable.

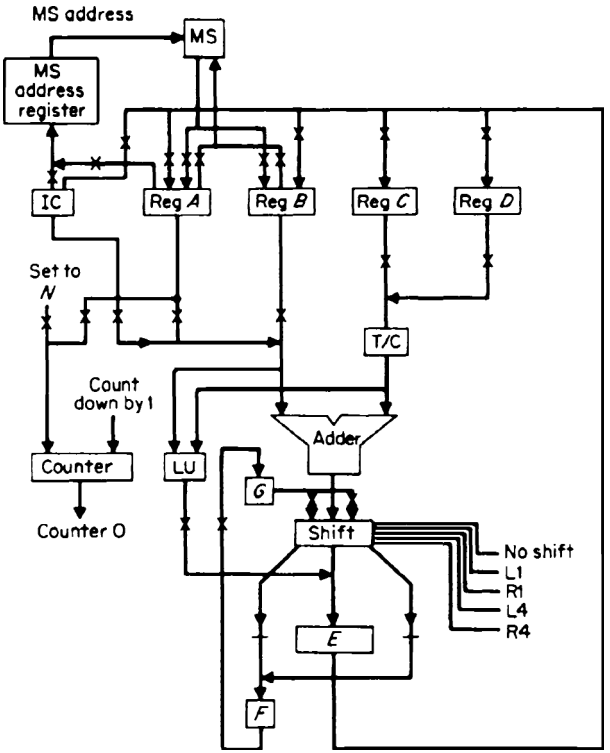

FIGURE 18.1.34 Microprogrammed ALU.

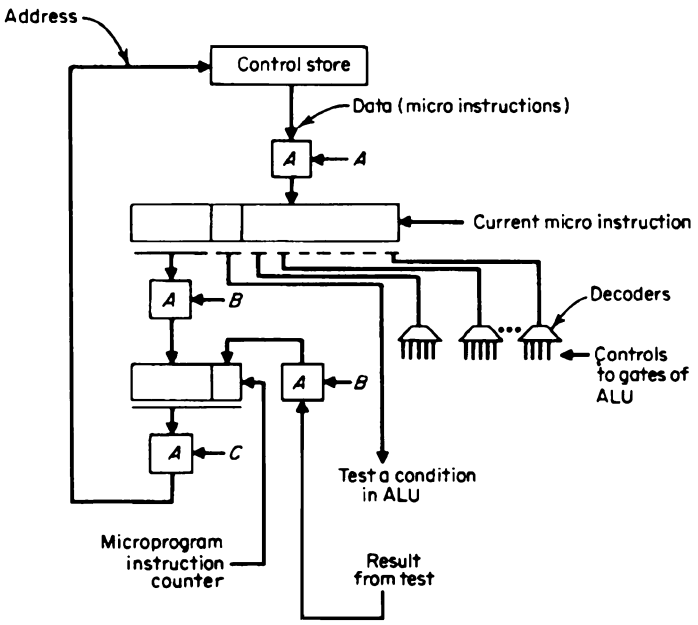

FIGURE 18.1.35 Microprogrammed control unit.

TABLE 18.1.2 Microinstructions Embodied in an ALU and a Control Section

Current address	Microinstruction	Comment
	•ADD	
51	B to ADD, C to ADD, NO-SHIFT	Add two operands
8	E to B	
9	A(2) to MS-ADR-REG, B to MS, GO TO 1	Store result branch to 1, next micro-instruction executed to be taken from control-store location 1
	•SUBTRACT	
52	B to ADD, C to ADD, NO-SHIFT T/C, GO TO 8	Go to 9, where result is stored
	•BRANCH (unconditional)	
53	A(3) to 1C, GO TO 1	This is the macroinstruction branch
	•MULTIPLY	
54	Set an N into counter, C to ADD, NO-SHIFT	Initialize
17	E to D, set C to 0s	
18	If last bit D = 1, then (B to ADD), C to ADD shift-R 1, 0 to input of high end of shifter, output of low end of shifter to F	Perform one add shift if last bit D = 1, only shift if last D = 0
19	E to C, F to G, COUNT down by 1	Increment counter
20	D to ADD, SHIFT-R1, G to input of high end of shifter	Shift D
21	E to D, if counter is not 0 then GO TO 17	Close loop
22	C to ADD	Store result in two MS locations
23	E to B, force 1 into C	
24	A(2) to ADD, C to ADD NO-SHIFT; A(2) to MS-ADR-REG B to MS	Store first half of result, increment result address
25	E to A(2)	
26	D to ADD, NO-SHIFT	
27	E to B	
28	A(2) to MS-ADR-REG, B to MS GOT TO 1	Store second half of result Branch to 1-fetch
	•SHIFT LEFT	
55	A(2) to COUNTER, C to ADD, NO-SHIFT	Operand 1 is number of bits operand 2 is to be shifted
10	If COUNTER = 0 then GO TO 9 E to B	Test if shift count was 0
11	B to ADD, SHIFT L1, COUNT down by 1	Shift loop
12	IF COUNT not 0, then GO TO 11 E to B	
13	GO TO 9	Completed shift

NOTE 1: (Location 7) This special test places the op-code bit pattern into the low part of the address for the next instruction, causing a branch to the appropriate microroutine for each op code. Branches are as follows:

Op code	Instruction	Address
1	ADD	51
2	SUBTRACT	52
3	BRANCH	53
4	MULTIPLY	54
5	SHIFT LEFT	55

NOTE 2: (Location 1) It is assumed that START button sets microinstruction counter to 1.

Instruction FETCH

In the preceding paragraphs the execution of instructions (E cycles) is discussed. In these cases, operations are initiated by setting an appropriate latch for the function to be performed. The signals that set the latch are in turn generated by circuits that interpret the information of the operation-code part of an instruction cycle (I cycle).

Whenever an instruction has completed execution (or when the computer operator presses the start button on the console), a *start-cycle* signal is generated at the next *A* time of the master-clock timing ring. This signal starts the I-cycle ring. The first action of this ring is to fetch the instruction to be executed from the main store by gating the instruction counter (IC) (sometimes called *program counter*) to the address lines of main storage and initiating a main-store cycle by pulsing a line called *start MS.*

These operations are illustrated in Fig. 18.1.36. At ring time 2, the instruction arrives back and is placed in the instruction register (IR). The instruction typically contains three main fields: the *operation code* (op code),

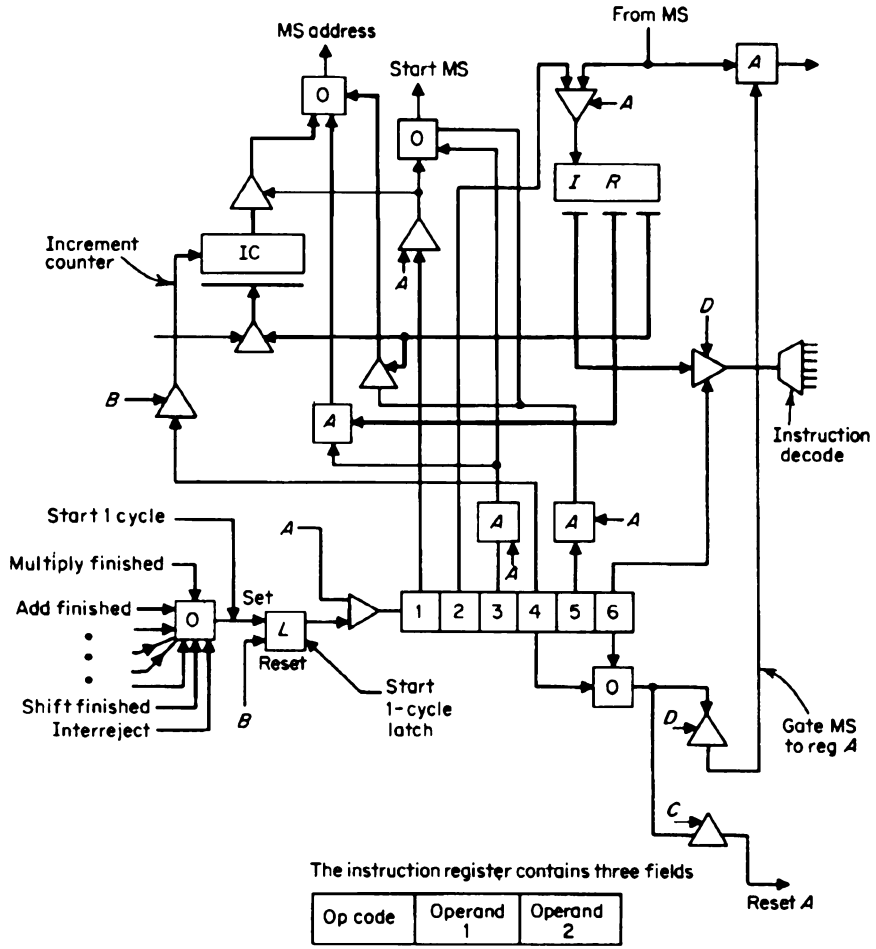

FIGURE 18.1.36 Implementation of equipment for an I cycle in a two-address machine. The instruction is first brought from main store and deposited in the instruction register. The operation proceeds by successively gating the information associated with two address fields into register *A*. The first is moved out of *A*, while the second is being sought from main store.

that determines what instructions is to be executed (ADD, SUBTRACT, MULTIPLY, DIVIDE, BRANCH), and the two addresses of the two operands participating in the operation. For certain classes of instruction, the operands must be delivered to appropriate locations before the E cycle begins.

During ring time 3 and 4, the first operand is fetched from main store and stored in *A*. During ring time 5, this operand must be transferred from *A* to an alternate location, depending on the nature of the instruction being executed. During ring time 5 and 6, the second operand is fetched and stored in *A*. Ring time 6 is also used to gate the op code to the *instruction decoder*, which is a combinatorial logic circuit accepting the *operation code*, or *instruction code*, from the *P* b in the op code. The decoder has 2^P input wires, one for each unique input combination. Thus for each bit combination entering the decoder, one output line is activated. These signals represent the start of an E cycle with each wire initiating the execution of one instruction by setting some latch, e.g., an add or multiply latch, or by initiating an appropriate microprogram.

Some op-code bit combinations may not be valid instruction codes. The outputs of the decoder are ORed together and fed into a circuit that ultimately interrupts the normal processing of the computer and indicates the invalid condition. At the beginning of the I cycle, the content of the instruction counter (IC) points to the instruction to be executed next. An *increment-counter* signal is generated during the I cycle in order to increment the counter, so that the address stored in IC points to the next sequential instruction in the program.

Instruction- and Execution-Cycle Control

In normal program operation, I and E cycles alternate. The I cycle brings forth an instruction from storage, sets up the ALU for execution, resets the instruction counter for the next I cycle, and initiates an E cycle. A number of conditions serve to interrupt this orderly flow, as follows:

1. When no more work is to be done, the end of a program is reached and the computer goes to a WAIT state. Such a state is reached, for example, by a specific instruction that terminates operations at the end of the I cycle in question, by a signal from the instruction counter when a predetermined limit is reached.
2. A STOP button may prevent the next setting of the *Start I cycle* hatch. A START button resets this latch.
3. When starting up after a shutdown, e.g., in the morning, activity is usually initiated by depressing an INITIAL PROGRAM LOAD button on the operator's console (the name varies from system to system, e.g., IPL, LOAD, START). The button usually performs three functions: a reset of all rings, latches, and registers to some predefined initial condition; a read-in operation from an I/O device into main store (usually the first few locations) of a short program; and an initiation of an I cycle. Program execution generally starts at a fixed location so that the IC initially is set to this value. In most computers, including PCs, this value is called IMPL (initial microprogrammed program load), which initializes the machine and performs self-testing on memory, I/O ports, and internal status conditions.
4. In multiprogramming, i.e., concurrent operations on more than one program, only one program at a time is in operation in the CPU, but transfers occur from one to another as required by I/O accesses, and so forth. The program to be transferred is handled by an *interrupt.* Under interrupt, an address is forced into the instruction counter so that on completion of the E cycle of the current program, a new instruction is referenced that starts the interruption. This instruction initiates program steps that store the data of the old program, e.g., in special registers or in special main-store locations. The contents of the IC, part of the IR, the contents of any registers, and the reason for the interruption are stored. The collection of these fields together is called a *program status word* (PSW). In most computers, this information, plus general registers must be saved when switching machine states. It can be referenced to reinitiate action at later time on the program interrupted.

Branch and Jump Instructions

Two kinds of instruction permit change in program sequence: *conditional* and *unconditional.* The purpose of such instructions is to permit the system to make some decision, so as to alter the flow of program, and to continue execution at some point not in the original sequence. In nonconditional branches the original program instruction provides for a branch or jump whenever the particular instruction occurs.

Conditional branches take the extra step of determining if some condition to be tested has been satisfied. Either the op code or the operand 1 field normally defines the test and/or the condition to be tested for. If the specified test has been satisfied, the branch is executed as described above. Otherwise no action is taken, and the next normally sequenced instruction is executed.

Advanced Architectural Features

The basic structure of the digital computer and its operation have been described in the previous paragraphs. This structure has proved to be flexible and adaptable to the solution of many different applications. There is, however, a continuing need to increase the performance of the computer and to make it easier to program so that even more applications can be handled. This has been achieved through a number of different approaches. One has been to develop sophisticated operating systems and to couple them closely to the hardware design of the computer. The net effect is that a programmer viewing the computer does not distinguish between the hardware and the operating system but perceives them as an integrated whole.

A second solution has been the development of architectural features that permits overlapped processing operations within the computer. This is in contrast to earlier computers that were strictly sequential and resulted in reduced performance and low throughput because vulnerable computer resources could remain idle for relatively long periods of time. The newer architectural features include such concepts as data channels, storage-organization enhancements, and pipelining described briefly in the following paragraphs. Another totally different approach to improved computer performance has been the development of computers with non-von Neumann architecture, described in the next section.

An increase in computer performance can be achieved by overlapping input/output (I/O) operations and processor operations. *Channels* have been introduced in large systems to permit concurrent reading, writing, and computing. The channel is in effect a special processor that acts as a data and control buffer between the computer and its peripheral devices. Figure 18.1.37 shows the organization of the computer when channels are introduced. Each channel can accommodate one or more I/O device control units. The channel is designed with a standard interface that is designed to permit a standard set of control status signals and data signals and sequences to be used to control I/O devices. This permits a number of different I/O devices to be attached to each channel by using I/O device control units that also meet the standard interface requirements. Each device control unit is usually designed to function with only one I/O device type, but one control unit may control several devices.

The channel functions independently of the CPU. It has its own program that controls its operations. The CPU controls channel activity by executing I/O instructions. These cause it to send control information to the channel and to initiate its operation. The channel then functions independently by being given a channel program to execute. This program contains the commands to be executed by the channel as well as the addresses of storage locations to be used in the transfer of data between main storage and the I/O devices. The channel in turn issues orders to the device control unit, which in turn controls the selected I/O device. When the I/O operation is completed, the channel interrupts the CPU by sending it a signal indicating that the channel is

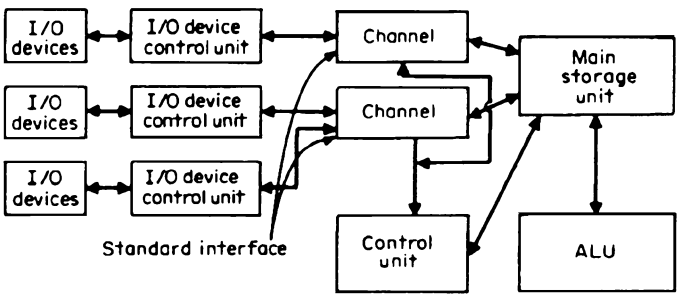

FIGURE 18.1.37 Computer organization with channels, separate logical processors that permit simultaneous input, output, and processing.

again free to perform further I/O operations. Several channels can be attached to the CPU and can operate concurrently. In PCs, I/O operation is controlled by adapter cards that fit into the motherboard.

Cache Storage

Cache storage was introduced to achieve a significant increase in the performance of the CPU at only a modest increase in cost. It is, in effect, a very high-speed storage unit that is added to the computer but is designed to operate in a unique way with the main storage unit. It is transparent to the program at the instruction level and can thus be added to a computer design without changing the instruction set or requiring modification to existing programs. Cache storage was first introduced commercially on the IBM System/360 model 85 in 1968.

Virtual Storage

Properly using and managing the memory resources available in the computer has been a continuing problem. The programmer never seems to have enough high-speed main storage and has been forced to use fairly elaborate procedures such as overlays to make programs fit into main storage and run efficiently. *Virtual-storage* systems were introduced to permit the programmer to think of memory as one uniform *single-level storage* unit but to provide a *dynamic address-translation unit* that automatically moves program blocks on pages between auxiliary storage and high-speed storage on demand.

Pipelining

A further improvement in computer performance was achieved through the use of *pipelining.* This technique consists of decomposing repetitive processes within the computer into subprocesses that can be executed concurrently and in an overlapped fashion.

Instruction execution in the control unit of the CPU lends itself to pipelining. As discussed earlier, instruction execution consists of several steps that can be executed relatively independently of each other. There is instruction fetch, decoding fetching the operands, and then execution of the instruction. Separate units can be designed to perform each one of these steps. As each unit finishes its activity on an instruction, it passes it on to the next succeeding unit and begins to work on the next instruction in succession. Even though each instruction takes as long to execute overall as it does in a conventional design, the net effect of pipelining is to increase the overall performance of the computer. For example, under optimal conditions, once the pipeline is full, when one instruction finishes, the next instruction is only one unit behind it in the pipeline. In this four-unit example, the net effect would be to increase the speed of instruction execution by a factor of 4.

This approach can be carried to the point where 20 or 30 instructions are at various stages of execution at one time. This type of processing is called *pipeline processing.* No difficulties arise during uninterrupted processing. When an interrupt does occur, however, it is difficult to determine which instruction has caused the interrupt since the interrupt may arise in a subsystem sometime after the IC has initiated action. In the meantime, the IC may have started a number of subtasks elsewhere by stepping through subsequent cycles. Operands within subunits may not be saved in an arbitrary intermediate state, since information is in the process of being generated for return to the main program. Because of the requirement that no further I cycles be started, interrupt is signaled when the pipeline is empty. At the time the interrupt is signaled, the IC does not point at the instruction causing the interrupt but somewhat past it. This type of interrupt is called *imprecise.*

Advanced Organizations

In addition to the von Neumann computer and its enhancements, a number of other computer organizations have been developed to provide alternative approaches to satisfying the needs for improved computational performance, increased throughput, and improved system reliability and availability. In addition, certain unique architectures have been proposed to solve specific problems or classes of problems.

TABLE 18.1.3 Classification Scheme for Computer Architectures

Acronym	Meaning	Instruction streams	Data streams	Examples
SISD	Single instruction stream, single data stream	1	1	IBM 370, DEC VAX, Macintosh
SIMD	Single instruction stream, multiple data stream	1	>1	ILLIAC IV, Connection Machine, NASA's MPP
MISD	Multiple instruction stream, single data stream	>1	1	Not used
MIMD	Multiple instruction stream, multiple data stream	>1	>1	Cray X/MP, Cedar, Butterfly

Computer organizations can be categorized according to the number of procedure (instruction) streams and the number of data streams processed: (1) a single-instruction-stream-single-data-stream (SISD) organization, which is the conventional computer; (2) a multiple-instruction-stream-multiple-data-stream (MIMD) organization, which includes multiprocessor or multicomputer systems; (3) a single-instruction-stream-multiple-data-stream (SIMD) organization; this uses a single control unit that executes a single instruction at a time, but the operation is applied across a number of processing units each of which acts in a synchronous, concurrent fashion on its own data set (parallel and associative processors fall into this category); and (4) a multiple-instruction-stream-single-data-stream (MISD) organization. (Pipeline processors fall into this category.) Table 18.1.3 summarizes the categories.

One way to achieve an improvement in performance and to improve reliability at the same time is to use multiprocessors. The American National Standard Vocabulary for Information Processing defines a multiprocessor as "a computer employing two or more processing units under integrated control." Enslow[61a] amplifies this definition by pointing out that a multiprocessor contains two or more processors of approximately comparable capabilities. Furthermore, all processors share access to common storage, to I/O channels, control units, and devices. Finally, the entire system is controlled by one operating system that provides interaction between processors and their programs.

A number of different multiprocessor system organizations have been developed: (1) time-shared, or common-bus, systems that use a communication path to connect all functional units, (2) crossbar-switch systems that use a crossbar switching matrix to interconnect various system elements, (3) multiport storage systems in which the switching and control logic is concentrated at the interface to the memory units, and (4) networking of computers interconnected via high-speed buses.

A number of large problems require high throughput rates on structured data, e.g., weather forecasting, nuclear-reactor calculations, pattern recognition, and ballistic-missile defense. Problems like these require high computation rates and may not be solved cost-effectively using a general-purpose (SISD) computer. Parallel processors, which are SIMD organization types, were designed to address problems of this nature. A parallel processor consists of a series of process elements (cells) each having data memories and operand registers. The cells are interconnected. A central control unit accesses a program, interprets each program step, and broadcasts the same instructions to all the processing elements simultaneously.

Distributed Processing

One of the newest concepts in computer organizations is that of *distributed processing.* The term distributed processing has been loosely applied to any computer system that has any degree of decentralization. The consensus seems to be that a distributed processing system consists of a number of processing elements (not necessarily identical), which are interconnected but which operate with a distributed, i.e., decentralized, control of all resources. With the advent of the less expensive micro- and miniprocessors, distributed processing is receiving much attention since it offers the potential for organizing these processors so that they can handle problems that would otherwise require more expensive supercomputers. Through resource sharing and decentralized

control, distributed processing also provides for reliability and extensibility since processors can be removed or added to the system without disrupting system operations. The types of distributed processing can be described by configurations in terms of the processing elements, paths, and switching elements: (1) loop, (2) complete interconnection, (3) central memory, (4) global bus, (5) star, (6) loop with central switch, (7) bus with central switch, (8) regular network, (9) irregular network, and (10) bus window. Distributed systems have been designed that fall into each of these categories. Hybrid forms use combinations of two or more of these architectural types.

Distributed and parallel processing systems can be thought of as being loosely coupled (each has its own CPU and memory) or tightly coupled (each has its own CPU and shares the same memory). Master–slave and client–server systems are part of the distributed and cooperative processing. In master–slave, one computer controls another, whereas in client–server, the requesting computer is the client and the requested resource is the server.

Stack Computers

In the CPUs discussed thus far, the instructions store all results, so that the next time an operand is used it has to be fetched. For example, a program to add A, B, and C and put result into E appears as

MOVE A to E	1 fetch, 1 store
ADD E to B store in E	2 fetch, 1 store
ADD E to C store in E	2 fetch, 1 store

In languages such as PL/I or FORTRAN this program might be written as the single statement

$$E = A + B + C$$

This equation describes a sequence of actions, as in the case of the program, but the specific sequence is not described. Since addition is commutative, a correct result is achieved by $E = [(A + B) + C]$, $E = [A + (B + C)]$, or $E = [(A + C) + B]$, each step occurring in any order. The computer, however, uses a specific program in achieving a result so that the method of writing the equation must generate a specific sequence of actions.

A method of writing an equation that specifies the order of operation is called *Polish notation.* For the above example of addition, one possible Polish string would be

$$AB + C + E =$$

In this string, the system would find A and B and, as determined by the plus sign *following* the two operands, add them. The result is then combined with C under addition called for by the second plus sign. The $E =$ symbols indicates that the result is to be stored in E. The plus sign appears *after* the A and B, and the specific string shown is called *postfixed.* An equivalent convention could place the operator first and would be called *prefixed.*

Any complex expression can be translated into a Polish string. For example in PL/I language, the statement

$$M = (A + B)*(C + D*E) - F;$$

means evaluate the right-hand side of the equation and store the result in the main-store location corresponding to variable M (asterisks indicate multiplication). The Polish string translation for this statement is

$$AB + DE*C + *F - M =$$

In translation from the types of expression permitted by higher-level languages, a machine can be programmed to analyze successively an arithmetic expression of the types shown above. In so doing, first, the outermost expressed or implied parentheses are aggregated and successively broken down until not more quantities remain. The first such quantities analyzed are generally the last computed, so that in the development of a Polish string from an algebraic expression, a first-in, last-out situation prevails.

Stacks

Evaluation of a Polish string in a machine is best performed using a *stack* (push-down list). A stack has the property that it holds numbers in order. A PUSH command places a value on the stack; i.e., it stores a number and an operation at the top of the stack and, in the process, lowers all previous items by one position. Numbers are retrieved from the stack by issuing a POP command. The number returned by the stack on a POP command is the most recently PUSHED one. The following example illustrates the behavior of a stack. The value in parentheses is the value *placed* on the stack for PUSH and returned by the stack for POP (assume the stack is initially empty):

PUSH (*A*)	stack contains	*A*
PUSH (*B*)	stack contains	*B A*
POP (*B*)	stack contains	*A*
PUSH (*C*)	stack contains	*C A*
POP (*C*)	stack contains	*A*
POP (*A*)	stack contains	nothing

Such a stack lends itself very well to the evaluation of Polish strings. The rules for evaluation are:

1. Scan the string from left to right.
2. If a variable (or constant) is encountered, fetch it from main store and place its value on the stack.
3. If an operator is encountered, POP the operands and PUSH the result.
4. Stop at the end of the string. If executed correctly, the stack is in the same state at the end of execution as it was at the start.

The advantage of using a stack is that intermediate results never need storing and therefore no intermediate variables are needed. In sequences of instructions where there are no branches, the operations can be stored in a stack. A program becomes a series of such stacks put together between branches.

This approach is called *stack processing.* In stack processing, a program consists of many Polish strings that are executed one after another. In some cases the entire program may be considered to be one long string.

Stacks are implemented by using a series of parallel shift registers, one per bit of the character code. The input is placed into the leftmost set of register positions. PUSH moves an entry to the right, and POP moves it to the left. The length of the shift registers is finite and fixed. The stack, however, usually must appear to the user as though it were infinitely deep. The stack is thus implemented so that the most active locations are in the shift register, and if the shift register overflows, the number shifted out at the right on a PUSH is placed in main storage. There the order is maintained by hardware, microprogramming, or a normal system program.

Trends in Computer Organization

Several trends in computer architecture and organization may have a significant impact on future computer systems. The first of these are the *data-flow computers*, which are data-driven rather than control-driven. In the data-flow computer, an instruction is ready for execution when its operands have arrived; there is no concept of control flow, and there is no program counter. A data-flow program can feature concurrent processing since many instructions can be ready for execution at the same time. In another area *capability systems* are receiving increased attention because their inherent protection facilities make them ideal for implementing secure operating systems. A capability is a protected token (key) authorizing the use of the object named in the token. One approach to implementing capabilities is through a *tagged* architecture, where tag bits are added to each word in storage and to each register. This tag specifies whether the contents represent a capability or not.

Special-Purpose Processors

Certain classes of problems require unique processing capabilities not found in general-purpose computers. Special-purpose processors have been designed to solve these problems. In some cases they have been designed

as stand-alone processors, but often they are designed to be attached to a general-purpose computer that acts as the host. One such class of special-purpose processors is the *associative processor.* It uses an associative store. Unlike the storage units described earlier, which require explicit addresses, an associative store retrieves data from memory locations based upon their content. The associative store does its searching in parallel over its entire storage in approximately the same time as required to access a single word in a conventional storage unit.

In digital signal processing (DSP), many repetitive mathematical operations must be performed, e.g., fast Fourier transforms, and require a large number of multiplications and summations. DSP algorithms are being used extensively in graphics processing and modem transmission. The *array processor* has been designed for these types of operations. It has a high-speed arithmetic processor and its own control unit and can operate on a number of operands in parallel. It attaches to a host CPU, from which it receives its initiation commands and data and to which it returns the finished results of its computation.

The *hybrid processor* uses a host digital CPU to which is attached an analog computer. These systems operate in a digital-analog mode and provide the advantages of both methods of computation. Hybrid processors are being replaced by very high-speed digital processors.

HARDWARE DESIGN

Design and Packaging of Computer Systems

Even a small computer may have as many as 3 million integrated circuits on a single chip, whereas a large system may have up to 100 million by the year 2000.

Nanosecond circuit speeds are common in high-performance computer systems. Since light travels approximately $^1/_3$ m in 1 ns, system configurations must be kept small to take advantage of the speed potential of available circuits. Thus the emphasis is on the use of microcircuit fabrication and packaging techniques. The layout of the system must minimize the length and complexity of these interconnections and must be realized, without error, from detailed manufacturing instructions. To permit these requirements to be met a *basic* circuit package must be available. The upper limit to the size of such a basic unit, e.g., an integrated circuit, is set by the number of crystal defects per unit area of silicon. If these defects are distributed at random, the selection of too large a chip size results in some inoperative circuits on a majority of chips. There is thus an economic balance between the number of circuits that can be fabricated in an integrated circuit and the yield of the manufacturing process.

Another limit on the size of the basic package is set by the number of interconnections between the integrated circuits. Reduced VLSI-chip power requirements and shrinking device dimensions have increased the number of circuits on a chip.