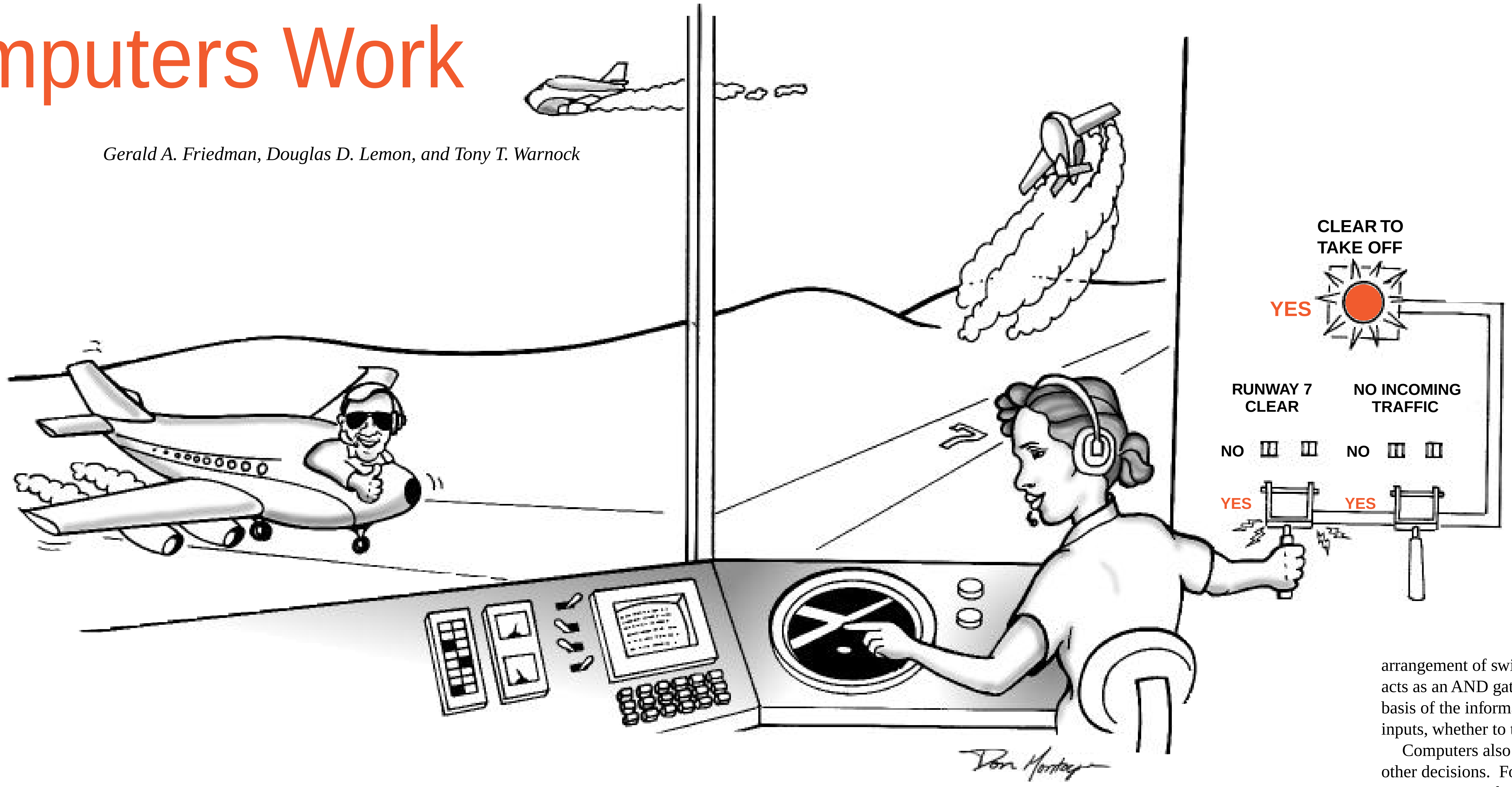
How Computers Work

an introduction to serial, vector, and parallel computers

Gerald A. Friedman, Douglas D. Lemon, and Tony T. Warnock



Parallel computers provide the most powerful information-processing capabilities available. Like all digital computers, they represent and process information in ways that are based on simple ideas of Boolean logic. In this article we introduce those ideas and sketch the ways in which they are implemented in the elementary circuits of electronic computers. We then discuss the principles behind the high performance—in particular, the high speed—of vector and parallel supercomputers.

Digital Information, Boolean Logic, and Basic Operations

Suppose that an airplane is about to enter a runway for takeoff. The air-traffic controller must determine, with the aid of a computer, whether the plane can safely proceed onto the runway. Has the airplane that just landed cleared the runway? Are all incoming planes sufficiently far away? The answers to those questions—yes or no, true or false—are information of a kind that can be managed by digital computers. The two possibilities are encoded within a computer by the two possible output states of electronic devices. For instance, "Yes" or "True" can be represented by an output signal on the order of 5 volts, and "No" or "False" by a near-zero voltage level. This simple code is the basis of the more complicated codes by which information is represented in computers.

Suppose that a light on the air-traffic controller's console is to be lit if and only if (1) the runway is clear of other planes and (2) all incoming planes are at a safe distance. What is the mechanism by which an electronic computer "decides" whether to turn on the light? Each of the two conditions is represented by the voltage in a wire; when the condition is true the wire is at roughly 5 volts. (We imagine that the voltages are set by the controller using manual switches.) The wires are the inputs to a circuit called an AND gate. The output wire of the AND gate is at high voltage (to turn on the light) only when both input wires are at 5 volts, that is, when the following statement is true: The runway is clear, *and* all incoming planes are at a safe distance. If the voltage that represents statement 1 is near zero (False), the output of the AND gate will also be at near zero voltage, so the light will not be lit. Likewise if the voltage representing statement 2 is low or if both voltages are low, the output will be at low voltage, and the light will not be lit. The simple arrangement of switches drawn above acts as an AND gate that decides, on the basis of the information given by its inputs, whether to turn on the light.

Computers also use gates that make other decisions. For instance, OR gates give an output of True when either or both of two inputs is True. The decisions made by gates—whether an output is to be True or False depending on which inputs are True and which are False—are equivalent to the operations of Boolean logic (developed by George Boole, a nineteenth-century English mathematician). As the arithmetic operation of multiplication is specified by a multiplication table, Boolean operations (or gates) are specified by their

Figure 1. Examples of Truth Tables

NOT

Input	Output
F	T
T	F

AND

		Input 2	
		F	T
Input 1	F	F	F
	T	F	T
		Output	

OR

		Input 2	
		F	T
Input 1	F	F	T
	T	T	T
		Output	

"truth tables," which show all possible input combinations and the consequent outputs. Figure 1 shows truth tables for NOT, AND, and OR. Figure 2 shows examples of electronic circuits for NOT and AND gates.

The operations AND, OR, and NOT can be combined to make other Boolean operations by using the output of one operation as an input of another. For instance, computers frequently use the EQUAL operation, which gives a True output if and only if its two inputs are equal (both True or both False). That operation can be built from the gates already defined: Input 1 EQUAL Input 2 is equivalent to (Input 1 AND Input 2) OR (NOT Input 1 AND NOT Input 2). (The Boolean operations are combined in the same way as mathematical operations.) The EQUAL operation is used, for example, when a computer checks whether a user intends to delete a file. The computer might prompt the user, `"If you're sure you want to delete that file, type Y."` The character the user types is translated into a sequence of eight bits according to some code such as ASCII. If the sequence of bits representing the typed character is equal to the sequence of bits representing the letter Y, the output of the EQUAL circuitry activates the circuitry that deletes the file. In fact, circuits consisting only of the AND, OR, and NOT gates defined above can make any decision that depends only on which statements in a given set are true and which are false.*

In addition to processing characters and logical statements, computers must also process numbers. Two-state electronic devices represent numbers through the use of the base-2, or binary, system—for instance, higher and lower voltages can indicate 1 and 0, respectively. A many-digit number is represented by the voltages in an ordered group of many such electronic devices. In fact, any kind of quantitative information can be represented by a series of higher and lower voltages. Furthermore, all of the operations of arithmetic can be performed (on binary numbers) by circuits made up of combinations of logic gates and other simple components. For example, the truth table of AND is a multiplication table if T is interpreted as 1 and F as 0.

*In principle, AND, OR, and NOT gates can all be built out of NAND gates (see Figure 2); they can also be built out of NOR gates (whose output is the opposite of that of an OR gate). Thus a computer could be built out of only one kind of gate.

Computers must also be able to store and retrieve information. Storage is performed by "memory" devices that can be set to maintain either a higher voltage or a lower voltage in their outputs and that, once set, maintain that voltage until they are reset. In our example of runway clearance at an airport, these devices were manual switches, but an electronic computer needs a memory containing a large number of two-state devices that can be set to either output state electronically. Such electronic circuits are called flip-flops. A flip-flop can be set to either state by the application of a voltage pulse to the proper input terminal, and it remains in that state until the application of another input voltage pulse. Thus each flip-flop "remembers" a single binary digit (abbreviated as "bit"), which is the smallest unit of quantitative information. Flip-flops are usually organized into cells, ordered arrays that each store a "word" of binary data (often 16, 32, or 64 bits long), which may encode a number, an alphabetic or other character, or quantitative information of any other kind.

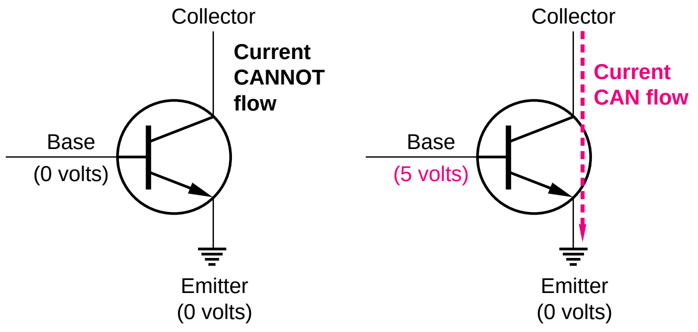
The elements of electronic computers—electronic representation of information, logic gates, and flip-flops—provide the abilities to execute any Boolean operation and to read and write data. The English mathematician Alan Turing and others showed that if a machine has those abilities (and if it has sufficient memory and enough time is allowed), it can perform any known information-processing task that can be specified by an algorithm—a sequence of well-defined steps for solving a problem. An algorithm may include alternative actions depending on contingencies. A great many tasks can be specified by algorithms, as shown by the abilities of computers to perform them—such as controlling a satellite in orbit, trading stocks, and managing the text and figures for this magazine. Indeed, the limits of computers are unknown—though computers cannot yet comment insightfully on a poem, no one has conclusively ruled out that possibility.

Stored-Program Computers

Computers accomplish calculations and other tasks by executing "programs," which are lists of instructions. Examples of instructions are fetching a number from memory, storing a number in memory, adding two numbers, and comparing two numbers or two characters. To "run" a program, the instruction list must first be loaded into memory in binary form. Then each instruction is read from memory and carried out by complicated circuits composed of the gates described above. Normally the instructions are performed in sequence. However, certain instructions allow different actions to be performed under different conditions. Depending on a specified input, these instructions cause execution either to

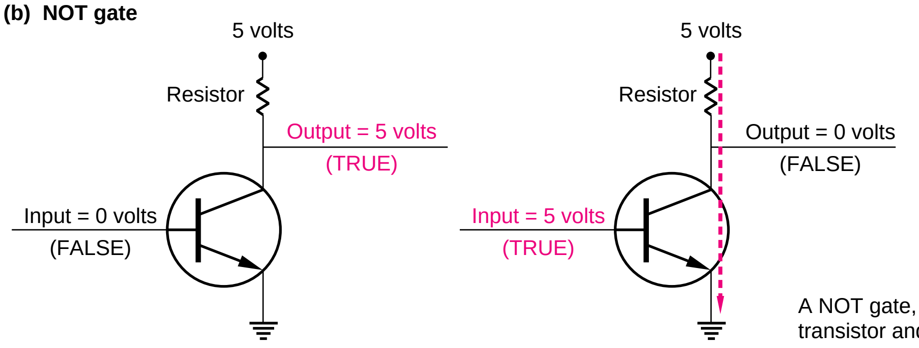
Figure 2. Computer Circuits

(a) Transistor



An NPN transistor can be put into either a conducting or a nonconducting state by the input voltage applied to it, so it can be used as the basic two-state switch in all the gates of an electronic computer. When an NPN transistor is used in a computer, the input is the voltage at the base and the output, which is controlled by the state of the transistor, is the voltage at the collector. (The emitter is grounded, or held at 0 volts). When the base is at a voltage near 5 volts, the transistor presents almost no resistance to the flow of electric current from the collector to the emitter, so the collector is effectively grounded. When the base is at a voltage near 0, the transistor blocks the flow of current between the emitter and the collector, so the voltage at the collector is determined by the other connections of the collector.

(b) NOT gate

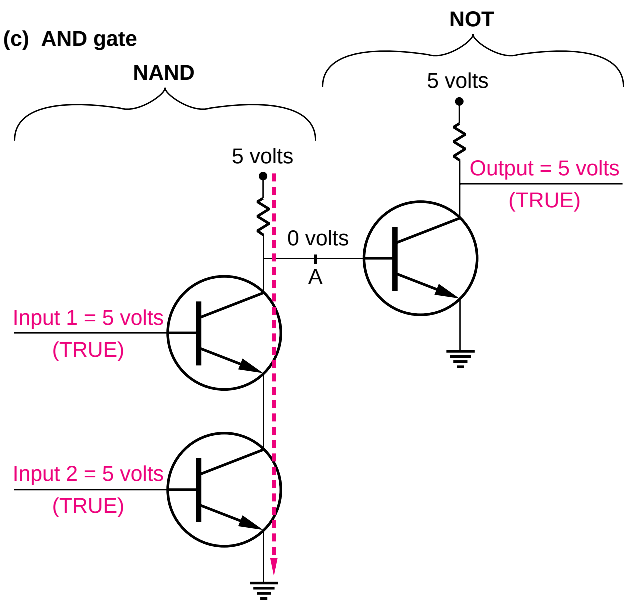


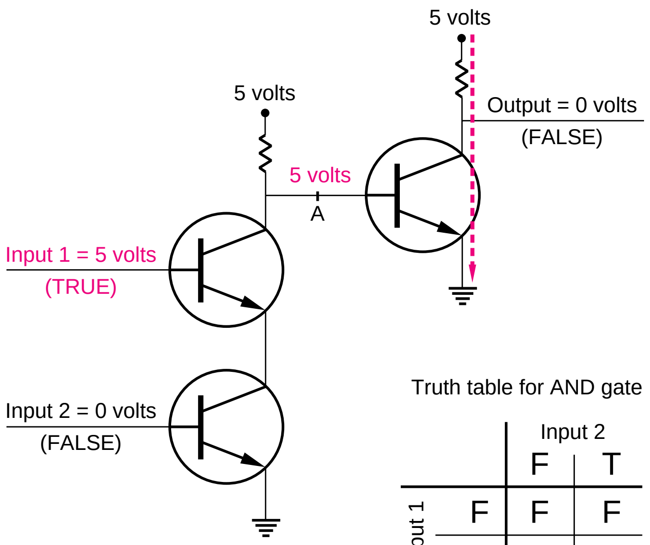
Truth table for NOT gate

Input	Output
F	T
T	F

A NOT gate, or inverter, can be made from an NPN transistor and a resistor. When the input is at 5 volts, the output (the collector voltage) is grounded and is thus fixed at 0 volts. When the input is at low voltage, the output is isolated from ground, no current flows through the resistor, and the output is consequently at 5 volts.

(c) AND gate





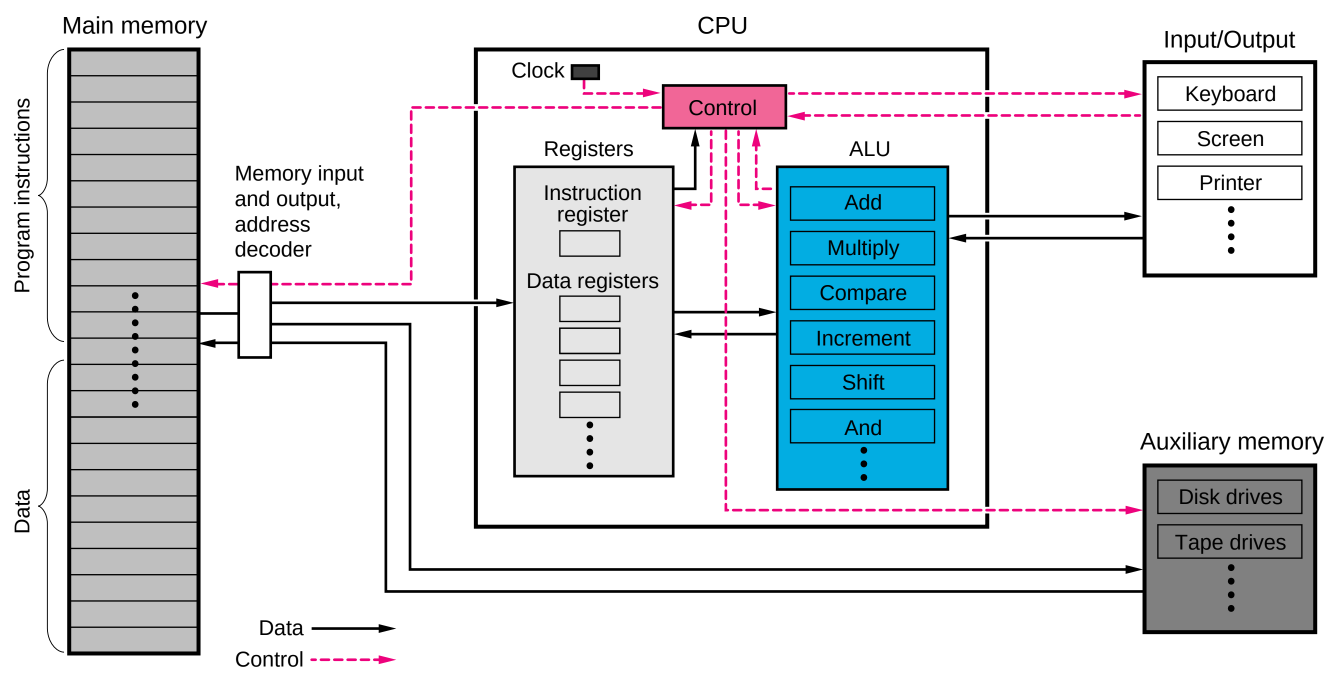
Truth table for AND gate

	Input 2: F	Input 2: T
Input 1: F	F	F
Input 1: T	F	T

Output

An AND gate can be made from NPN transistors and resistors. The part of the circuit to the left of point A is actually a NAND gate; that is, the voltage at point A is given by the operation NOT(input 1 AND input 2). The NAND gate operates by a principle similar to that of the NOT gate, and the voltage at point A is inverted by the NOT gate to the right. When both inputs are at 5 volts, point A is grounded (False), so the output of the AND gate is True. When either input is near 0 volts, point A is isolated from ground and thus at 5 volts (True), so the output of the AND gate is False.

Figure 3. Architecture of a Serial Computer



"jump" to a specified instruction elsewhere in the program or to continue with the next instruction in sequence.

Serial Computers

The simplest electronic computers are serial computers—those that perform one instruction at a time. Figure 3 shows a typical example of the architecture, or overall design, of a modern serial computer. Note that the architectures of computers vary enormously on all levels, from the choice and configuration of the large-scale components down to the choice of binary code used to represent numbers. Every illustration in this article shows an example chosen from among many possibilities.

The major components of a serial computer appear in Figure 3. The central processing unit (CPU) is the heart of the computer. The CPU in a modern serial computer often consists of a single integrated-circuit chip. It contains the arithmetic-logic unit (ALU), the part of the computer that actually computes. The ALU consists of circuits that perform various elementary processes such as addition, multiplication, comparison of numbers, and Boolean operations. Such operations are performed on data that are in registers—very quickly accessible memory cells located within the CPU. The control unit contains circuits that direct the order and timing of the operations by "gating," or opening and closing electronic pathways among the ALU, the memory, and the input/output units. A typical calculation involves loading data from main memory into registers, processing of those data by the ALU, and storage of the results in main memory. Note that the CPU typically includes a clock, a device that synchronizes operations by emitting electric pulses at regular intervals.

Main memory stores information that can be read and written by other parts of the computer. It consists of memory cells that are packed together densely on a device consisting of one or more semiconductor chips; a single chip can store many millions of bits of information. Each cell is labeled by a numerical address, and other parts of the computer can read from or write to any cell by sending its address and a read or write signal to the memory device. Because the memory cells of the device can be accessed in any order, main memory is called random-access memory, or RAM. Main memory is "volatile;" that is, the stored information is lost when the power is switched off. Therefore main memory typically stores information that the CPU will use in the short term, such as the program being executed and the data that the program processes and generates.

The computer contains a very large number of connections between its main components and within the CPU. The connections are conducting paths that make the voltage at the input of a circuit equal to the voltage at the output of another. The lines in Figure 3 indicate schematically whether one part of the computer is connected to another part. Solid lines indicate data connections; dashed lines indicate control connections. The connections consist of multiple conducting paths or wires. For instance, the connection between main memory and the CPU includes separate pathways for data transfers, addresses, read and write signals, and other purposes. Furthermore, the data path must have a number of wires equal to the number of bits that must be transferred. Thus in a computer whose words consist of 16 bits, a data connection between main memory and the CPU consists of 16 wires.

Auxiliary memory refers to nonvolatile storage devices such as disk and tape drives. Auxiliary memory is generally much more capacious than main memory (and less expensive per bit of capacity). Present auxiliary memories have capacities of up to millions or billions of words. However, the access (read/write) times for auxiliary memory are greater than those for main memory. Therefore auxiliary memory is generally used to store programs and data while they are not in use.

Input and output devices communicate data between the main memory and users or other computers, translating the data between bit sequences used by the computer and forms understandable to people (or other computers). Probably the most common input device is the keyboard, and the most common output devices are the printer and the monitor. There are many other more- or less-specialized input-output devices, such as network connections, modems, mice, scanners, loudspeakers, and so forth. Some auxiliary-memory devices can be used for input/output—for instance when one computer writes onto a floppy disk that another computer will read.

To show how a serial computer performs a calculation, we will take as an example the calculation of the salary raises *r* of a company's 800 employees. Each raise is given by the product of an employee's present salary *p* and an adjustment factor *a* previously calculated on the basis of the employee's performance. In our example we use the common technique of storing the values of *p* in the computer as a one-dimensional array, or vector, `p(i)`; the value of *p* for the first employee is called `p(1)`, and so forth. When the program runs, the elements of `p(i)` are stored in consecutive memory cells. The values of *a* are stored similarly as elements of a vector `a(i)`. Thus the calculation can be written as `r(i) = p(i)*a(i)` for all `i`.

Figure 4 shows how an exemplary serial computer performs a multiplication. Since the auxiliary memory and input-output devices are not involved in this calculation, they are omitted from the figure. The program for the raise calculation has been loaded into main memory; each instruction is stored as a 16-bit number. The figure shows the location of one instruction, which is represented by the four digits 3123. The first digit in the instruction is interpreted by our example control unit as an operation, the next two digits are the numbers of the two registers containing the operands, and the final digit is the number of the register where the result is to be placed. Here 3 is the code for MULTIPLY; then the instruction 3123 means, "Multiply the number in register 1 by the number in register 2 and place the result in register 3."* The `p(i)` and `a(i)` arrays are also in main memory. In this example, each of the first two elements of `p(i)` has already been multiplied by the corresponding element of `a(i)`, and the resulting `r(1)` and `r(2)` have been stored in main memory. (The cell where `r(3)` will be stored is still blank.) The numbers in the registers—`p(3)` and `a(3)`—were loaded by the execution of the previous instructions.

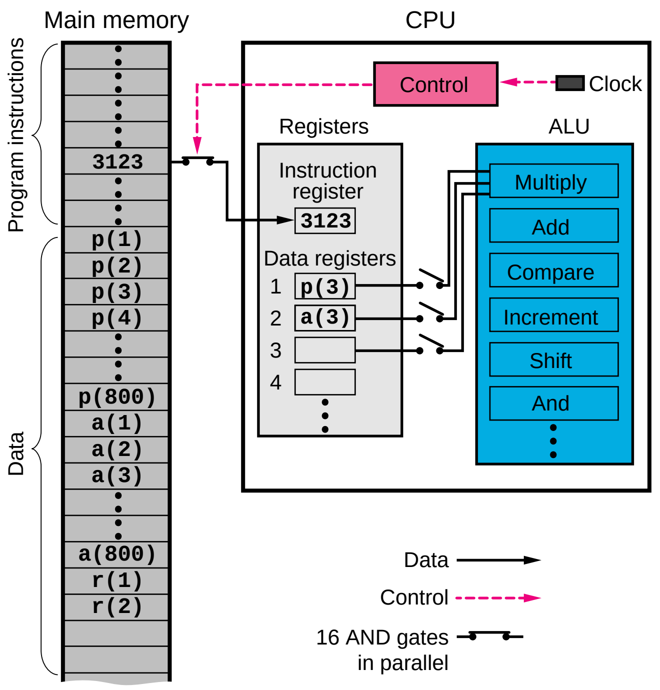
The computer in this example performs instructions in two stages: First it fetches an instruction from main memory, then it executes the instruction. Figure 4a shows the fetching process. We begin following the process just after the instruction to load `a(3)` has been executed. As always after an instruction has been executed, the control unit is in a state such that a voltage impulse from the clock causes it to fetch the next instruction. (The control unit was initially put in that state by the "run" command that began the execution of a program.) Also, the address of the multiply instruction is in the instruction-address register (not shown). At the clock signal, the instruction, stored as the number 3123, is fetched from main memory into the instruction register. The bottom part of Figure 4a illustrates how the outputs from the control unit transfer, or gate, the instruction from main memory to the instruction register. In general, gating is the control unit's primary means of controlling the operations.

At the next clock signal, the instruction is gated into circuits in the control unit. Figure 4b shows that the resulting outputs from the control unit gate the

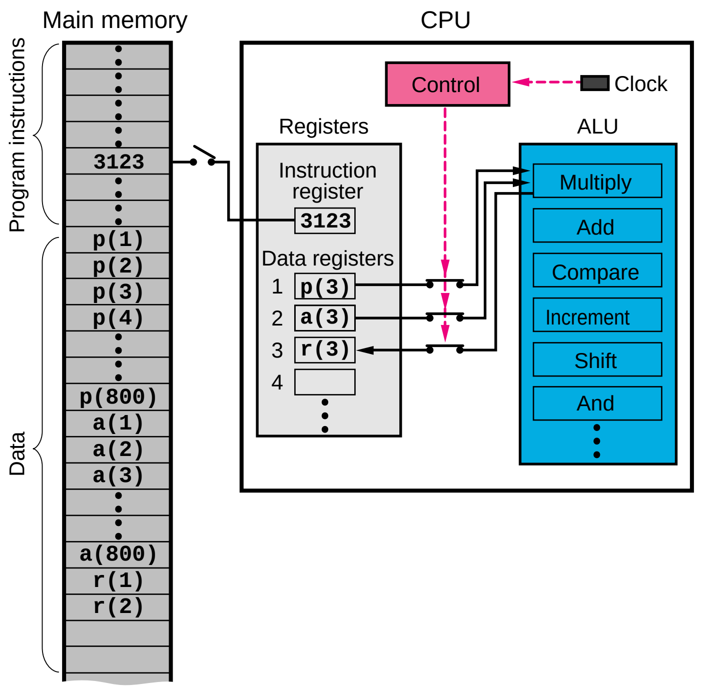
*In this example, each digit stands for one of the 16 four-bit numbers. and is therefore to be interpreted in base 16. The instruction represented by the base-16 number 3123 would be stored as the binary representation of that number: 0011 0001 0010 0011.

Figure 4 . Execution of One Instruction

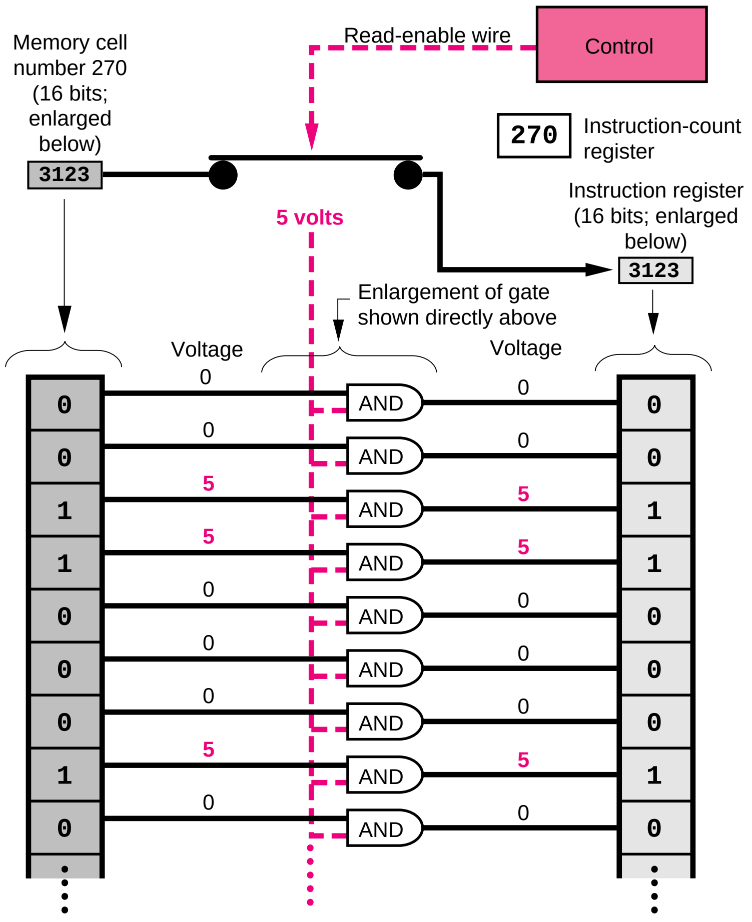
(a) Fetch instruction to multiply



(b) Execute multiplication



Detail of gating shown in (a) above


Memory cell number 270 (16 bits; enlarged below)
3123
Read-enable wire
Control
270
Instruction-count register
5 volts
Instruction register (16 bits; enlarged below)
3123
Enlargement of gate shown directly above
Voltage
Voltage
AND
0
0
1
1
0
0
0
1
0
5

Shown at left is the gating of an instruction, which consists of a single word, from a cell in main memory to the instruction register. A number of other connections controlled by AND gates are involved in fetching the instruction; for clarity they are not shown, but they are described in the paragraph below. The instruction-count register specifies that the word to be copied is in memory cell number 270. We assume the computer was designed to use words consisting of 16 bits; therefore the memory cell and the instruction register each consist of 16 flip-flops. The output of each flip-flop of the memory cell is connected to one input of an AND gate, and the read-enable wire from the control unit is connected to the other input of each AND gate. When the read-enable wire (red dashed line) is set by the control unit to 5 volts as shown, the output wire of each AND gate is at the same voltage as its input from memory. Therefore the output of each flip-flop of the instruction register is set to 5 volts or 0 volts depending on whether the bit stored in the corresponding flip-flop in main memory is 1 or 0. Thus the control unit "gates" the 16 bits of the word from the memory cell to the instruction register. However, when the read-enable wire is at 0 volts, all the inputs to the instruction register are at 0 volts. This set of AND gates might be compared to a starting gate at a racetrack, which either holds all the horses in place or lets them all run at once.

The appropriate memory cell is chosen by use of the circuit called the address decoder. The control unit gates the address of the desired instruction from the instruction-count register to the address decoder which, in combination with a signal from the control unit to the entire memory, turns on the read-enable wire of the memory cell at that address and no other. The output of the cell goes to the memory-output pathway (which along with the address decoder and the memory-input pathway was shown adjacent to the main memory in Figure 3). Yet another signal gates the contents of the memory-output pathway to the instruction register.

numbers in registers 1 and 2 to the multiplier and gate the output of the multiplier, which is the product of the two numbers, into register 3. The execution of the multiplication takes several clock cycles.

Other instructions, perhaps seven or eight, must be executed as part of the routine that calculates each employee's raise. Before the multiplication the memory addresses of `p(i)`, `a(i)`, and `r(i)` are calculated on the basis of the value of `i`. As we have mentioned, the factors `p(i)` and `a(i)` are gated into registers. This is the most time-consuming step; loading a word from memory may take 20 clock cycles. After the multiplication, the number in register 3 is stored in memory at the address for `r(i)`. Finally, the value of `i` is incremented and the result is compared with 800 to determine whether the calculation is finished. If not, the control unit jumps to the beginning of the routine (by writing the address of the first instruction of the multiplication routine into the instruction-address register) and repeats the routine for the next value of `i`. The total time to perform the routine for one value of `i` may be about 30 clock cycles.

Typical processors contain circuits to perform all these operations. The specifics of the elementary instructions, however, vary greatly among different computers, and the choice of "instruction set" is an important part of computer architecture.

A computer's instruction set is also called its assembly language. Users can write programs in assembly language, one encoded instruction at a time. However, the coding is quite tedious, and the result is difficult for a human being to read. Moreover, since instruction sets differ among computers, an assembly-language program written for one type of computer cannot be "understood" by any other. Therefore, many high-level languages have been developed. They consist of instructions that combine several assembly-language instructions in ways that are convenient for humans. For instance, when written in the popular language FORTRAN, the instructions (or code) for the raise calculation above might look like this:

```
do i = 1, 800
    r(i) = p(i)*a(i)
end do
```

Those lines of FORTRAN code instruct the computer to perform the statement between the `do` and the `end do` for all values of `i` from 1 to 800. Programs called compilers, assemblers, and loaders translate the high-level code into binary instructions and assign memory addresses to variables. Not only is the FORTRAN code easier to read than assembly code, but it is also portable; that is, it can be executed by any computer that has a FORTRAN compiler. (Complicated programs may not be perfectly portable—minor adjustments may be required to run them on computers other than the original.) All the advantages of high-level languages are counterbalanced by some cost in execution speed, so sometimes when speed is the highest priority, programmers write in assembly language.

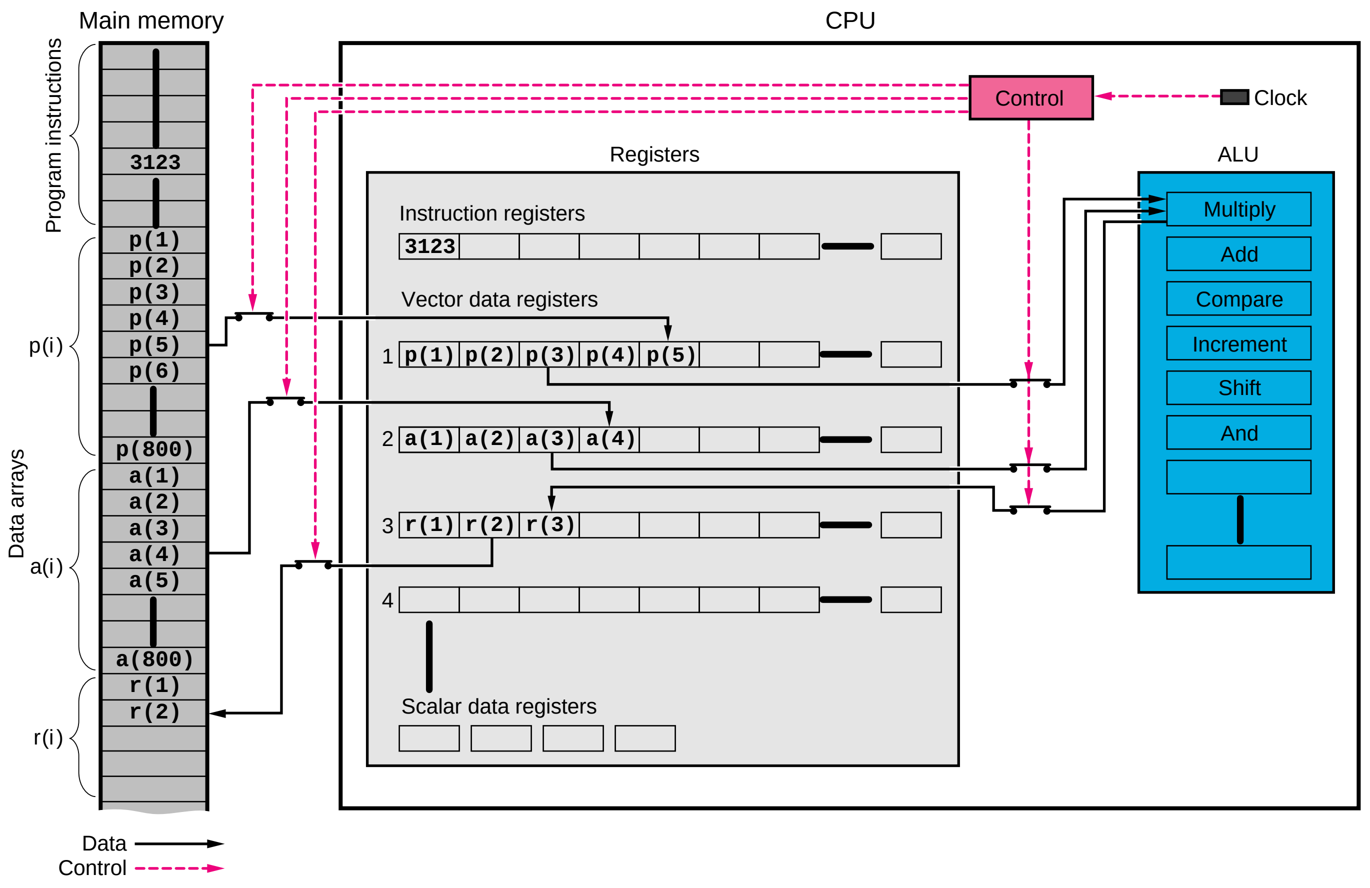
Speed is not always the highest priority, but it is always important. Speed is continually being increased by the development of faster and smaller gates and flip-flops. (Smaller components can be packed on a chip in larger numbers so that there are more of the fast connections within chips and fewer of the slower connections between chips.) Integrated-circuit technology has reached the point where mass-produced CPU and memory chips are fast enough to serve as components in the fastest computers used for large calculations in science and engineering.

To take advantage of high-speed components, manufacturers use architectures designed for high speed. For instance, the main memory must be large so that it can store all the data for large problems without requiring transfers of data to and from auxiliary memory during the calculation. (Such transfers are relatively slow.) Also, the bandwidth, or capacity, of the connections must be increased by adding wires, so that the connections can transfer more data in a given amount of time. These features provide higher performance, but serial computers have the essential limitation that instructions can be performed only one at a time. Therefore supercomputers—the fastest (and most expensive) computers—usually have architectures that allow the performance of several instructions at once. There are two such architectures: vector and parallel.

Vector Computers

Vector computers are based on processors that execute calculations on the elements of a vector in assembly-line fashion. As illustrated in Figure 5, a vector processor contains five to ten "functional units" that can act simultaneously, each carrying out an instruction. Functional units are thus analogous to the stations on an assembly line. The functional units include various circuits on the ALU. Three additional functional units are the separate connections between main memory and registers that allow the simultaneous loading of two numbers into registers and storing of one number from a register. (These connections are an example of high-bandwidth connections that speed up transfers of data.) Another distinctive feature of vector processors is that they contain several vector registers. The vector registers shown in Fig-

Figure 5. Simultaneous Execution of Instructions on a Vector Processor



ure 5 can contain 64 elements of a vector at once.

We use the calculation of raises as our example again. As shown in Figure 5, the calculation is set up in "assembly-line" fashion so that at any time each functional unit is working on a different element of the result `r`. At the stage shown, the first element of `r` has been completely processed and stored. The second element is being stored, the multiplication for the third element is being performed, and data for the fourth and fifth elements are being loaded from RAM into elements of vector registers. When 64 elements of `r` (enough to fill a vector register) have been calculated, the process repeats to compute the 65th through 128th elements, and so forth until all 800 elements have been computed.

After the first element of `r` is stored, another element comes off the line in every clock cycle—just as cars may come off an assembly line every hour even though assembling a single car takes days of work. Since a serial calculation of a single element requires perhaps 30 clock cycles, a vector processor that performs those instructions simultaneously reduces the calculation time by a factor of roughly 30. Calculations that use more of the processor's functional units are sped up still more. Furthermore, each instruction is fetched only once and executed many times, so the speedup is greater still. (In a typical serial computer, fetching each instruction requires a clock cycle or more.) Some "overhead" time is needed to set up the assembly line, but that time can often be reduced or eliminated because functional units not used in one assembly line can set up the next one. Despite a slight additional overhead at the beginning and end of the calculation when some functional units on the assembly line are idle, speedups of factors of hundreds are possible in favorable cases.

However, many calculations cannot be sped up by vector processing because they cannot be "vectorized," or put in the form of identical operations on elements of vectors so that each functional unit has a predictable task on the "assembly line." The number of elements of the vectors must be at least comparable to the number of elements of the vector register so that the time saved by performing the calculation on an assembly line more than compensates for the overhead. Also, the calculations on different components should be independent of each other so that they can proceed simultaneously. An example that does not satisfy that condition is the calculation of the sum of many terms, $\sum_i \texttt{x(i)}$. If the calculation is performed by the straightforward method of adding each element in turn to the subtotal, it cannot be vectorized because the the subtotal of the first `i` elements must be completely calculated before the addition of `x(i + 1)` can begin. (Ingenious methods for vectorizing sums of many terms have been invented, but nonetheless vector computers perform sums more slowly than such calculations as the multiplication shown in Figure 5.) Calculations that vector computers can perform particularly fast include linear algebra, Fourier and other integral transforms, and other calculations used in such scientific applications as finite-difference and finite-element simulations of fluid flow. Even when a calculation cannot be vectorized, vector computers often provide some speedup if the calculation involves large amounts of data, because one may be able to vectorize the loading of the data from main memory and the storing of the results.

Vector computers have been made relatively easy to program. In fact, programs written for a serial computer can usually run on vector computers, although to take full advantage of vector processing, the programs often must be rewritten. However, techniques for doing so are well known. Furthermore, much of the necessary rewriting is now automated; that is, the programmer can write code as if for a serial machine, and the compiler will translate the program to assembly code optimized for the vector processor.

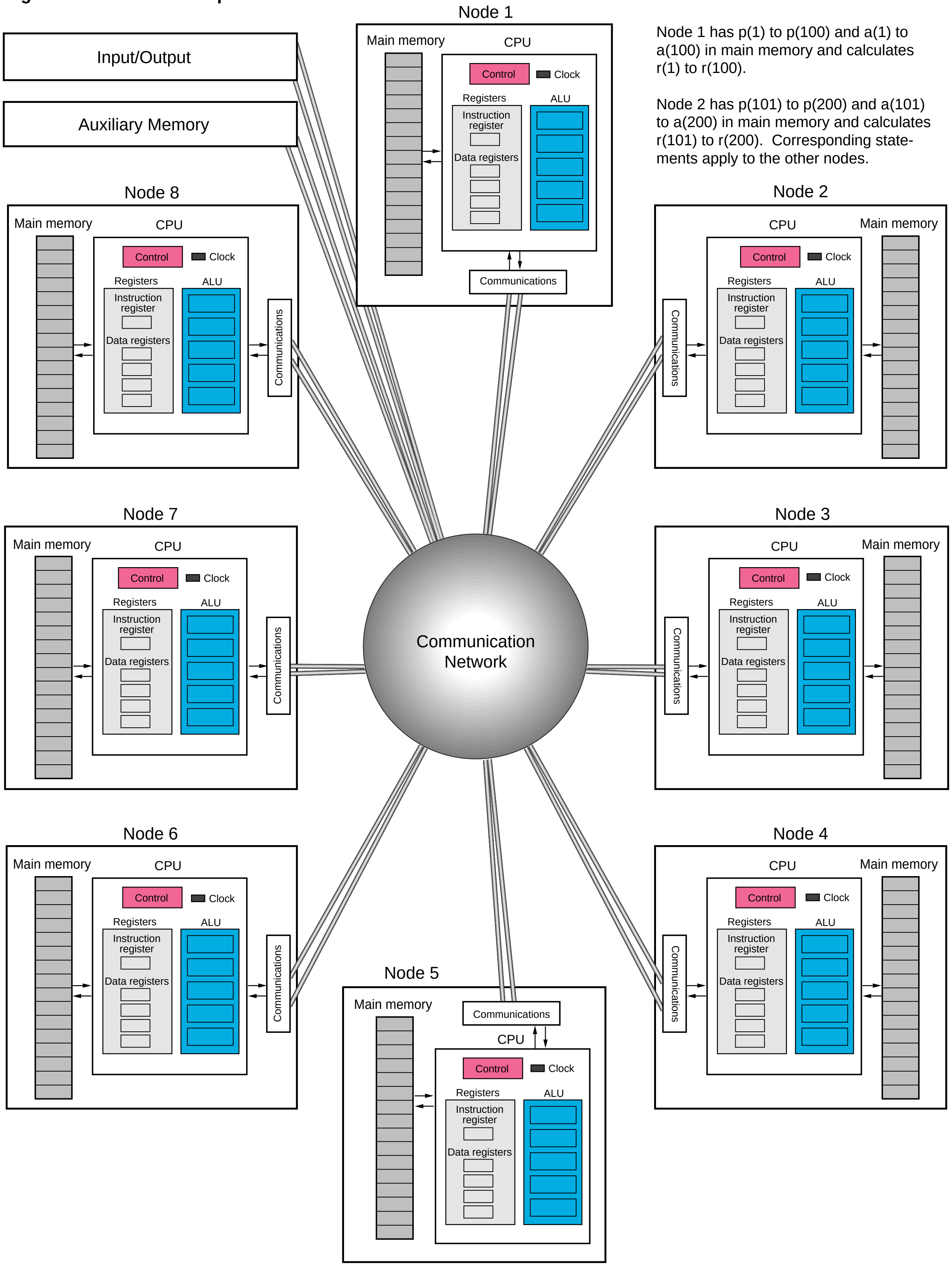
Parallel Computers

Parallel computers consist of a number of identical units that contain CPUs and essentially function as serial computers. Those units, called nodes, are connected to one another and simultaneously perform more or less the same calculation on different parts of the data. For example, Figure 6 shows an eight-node parallel computer calculating the salary raises of our earlier example. Each processor is connected to its own main memory. (This arrangement is called distributed memory.) The nodes are connected to each other by a high-bandwidth communication network. Each of the eight nodes stores and processes only one-eighth of the data—node 1 stores `p(1)` through `p(100)` and `a(1)` through `a(100)`, and it is calculating `r(1)` through `r(100)`. The maximum possible increase in speed would be a factor equal to the number of nodes. If that speedup can be attained for a given calculation, the calculation is said to be scalable. In practice, the speedup is somewhat less, because time must be spent in communicating the data between nodes and in coordinating the functions of the nodes. When a calculation can be made nearly scalable, it is well-suited for massively parallel computers—those that have hundreds of processors.

Parallel computers have the great advantage over vector computers that they can be built from off-the-shelf CPUs and other components made for serial computers. Since vector computers require special-purpose vector processors, parallel computers can offer higher maximum speed (in terms of operations per second) than vector computers of the same price. However, running parallel computers at their maximum speed is more difficult than running vector computers at their maximum speed. The two basic problems of parallel computers are the coordination of the processors and their access to data.

A simple hardware method of making data available to all processors would be to change the architecture shown in Figure 6 by giving the computer a shared memory—one extremely large memory to which all the processors have equal access. However, for technical reasons the number of data and control connections to a shared memory increases faster than the number of processors, so making massively parallel shared-memory computers is impractical at present. Therefore most massively parallel computers, like the computers sketched in Figure 6, have distributed memories, and each processor can retrieve data directly from its own memory only. Because reasons of cost require connections between the nodes of a distributed-memory computer to have less bandwidth than those between a processor and its own main memory, data transfers between nodes are relatively slow. An important problem in programming such computers is minimizing the time spent in transferring data from the memory of the processor that stores them to the memory of the processor that needs them. Several solutions to this problem have been tried. The choice of solutions presents computer buyers with a trade-off: A computer that handles the communication and control automatically and thus hides communication and control problems from the

Figure 6. A Parallel Computer



Node 1 has p(1) to p(100) and a(1) to a(100) in main memory and calculates r(1) to r(100).

Node 2 has p(101) to p(200) and a(101) to a(200) in main memory and calculates r(101) to r(200). Corresponding statements apply to the other nodes.

user is easier to program than one that requires the user to solve those problems. However, automation is relatively inflexible, so such a computer can speed up fewer kinds of programs than one that requires the user to do more programming work.

The earliest approach to solving the communication problem is particularly automatic and rigid. It is known as the single-instruction/multiple-data, or SIMD, architecture. In SIMD computers, all processors execute the same instructions in synchrony on different parts of the data set. On a SIMD machine, the control of the processors and communication between them is straightforward. Programmers can write code for SIMD computers in a style identical to that for serial computers, and each processor executes a copy of the code. Since the problems are divided in a predictable way, data can be distributed to the processors that need them without any need for complex calculations. However, only a limited set of problems can be approached so simply. In fact, most calculations suitable for SIMD computers are equally suitable for vector computers.

The other common parallel architecture is MIMD (multiple-instruction/multiple-data), which is being used for most currently available parallel computers. Each processor in a MIMD computer functions independently of the other processors. Because MIMD systems have greater flexibility than vector or SIMD systems, they work well on more kinds of programs: for instance, databases, simulations of many interacting bodies (see "State-of-the-Art Parallel Computing" and "A Fast Tree Code for Many-Body Problems), and Monte Carlo simulations (see "A Monte Carlo Code for Particle Transport"). However, interprocessor communication in MIMD systems is more complicated than in SIMD systems.

The different approaches to communication in MIMD machines include different programming models. (A computer's programming model is the structure of the programs it is designed to run.) Two programming models that offer relative ease of programming but limited versatility are data parallelism and distributed computing. Data parallelism is similar to SIMD in that each processor performs the same routine, but differs in that each processor can spend a different amount of time on the routine. High-level languages have been devised to automate most of the communication in data-parallel computation. At the other extreme of processor synchrony is distributed computing, in which the processors are nearly independent. Distributed computing is particularly useful in Monte Carlo calculations; then each processor can perform simulations independently of all the others.

The most versatile programming model, between data parallelism and distributed computing in the degree of processor independence, is message-passing. This model effects communication by having the processors send messages to each other as needed to request or transmit data. Thus communication can be performed more flexibly than in data-parallel computers, where communication must fit into a fixed model. However, programs that call for message passing must include specific instructions for the messages. Therefore programmers often need to understand the details of how the particular computers they use handle communication. Programmers are making progress in applying messages-passing methods, as described in several articles in this issue. We should note that some massively parallel computers allow the user to choose or combine programming models to suit each problem. The CM-5 Connection Machine supports both the data-parallel and the message-passing models. The article "State-of-the-Art Parallel Computing" describes the Laboratory's CM-5, including the control of and communication among the processors.

Finally, we should note that there is no reason not to combine vector and parallel architectures. Vector computers have long incorporated up to sixteen processors with a shared memory. Furthermore, one modern supercomputer, the CM-5, is a massively parallel computer in which each node contains one or more vector processors in addition to its main CPU. Computers of these "hybrid" types dominate the supercomputer market. Making predictions about supercomputers is very risky because of the rapid development of hardware, software, and architectures, but we can expect that for some time the fastest supercomputers will probably be massively parallel computers and vector-parallel hybrids. ■

Acknowledgement

We are grateful to Randy E. Michelsen for highly valuable discussions.

Further Reading

Caxton C. Foster and Thea Iberall. 1985. *Computer Architecture.* Van Nostrand Reinhold Company.

Alan Freedman. 1991. *The Computer Glossary: The Complete Illustrated Desk Reference,* fifth edition. The Computer Language Company Inc.

Dharma P. Agrawal, editor. 1986. *Tutorial: Advanced Computer Architecture.* IEEE Computer Society Press.

Anthony Ralston, editor. 1976. *Encyclopedia of Computer Science.* Petrocelli/Charter.

Glenn Zorpette, editor. 1992. *Teraflops Galore.* September 1992 issue of *IEEE Spectrum.*