CHAPTER

1

Computing and Computers

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

1.1 THE NATURE OF COMPUTING

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

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

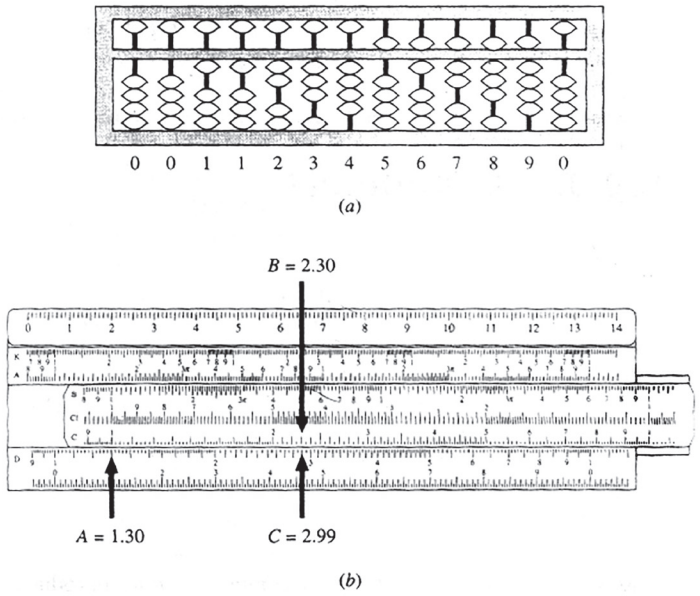

Figure 1.1

(*a*) Japanese abacus (*soroban*) displaying the number 0011234567890;

(*b*) Slide rule illustrating the multiplication $1.30 \times 2.30 = 2.99$.

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

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

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

The English computer pioneer Charles Babbage (1792-1871) often cited the following example to justify construction of his first automatic computing machine, the Difference Engine

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

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

1.1.1 The Elements of Computers

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

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

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

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

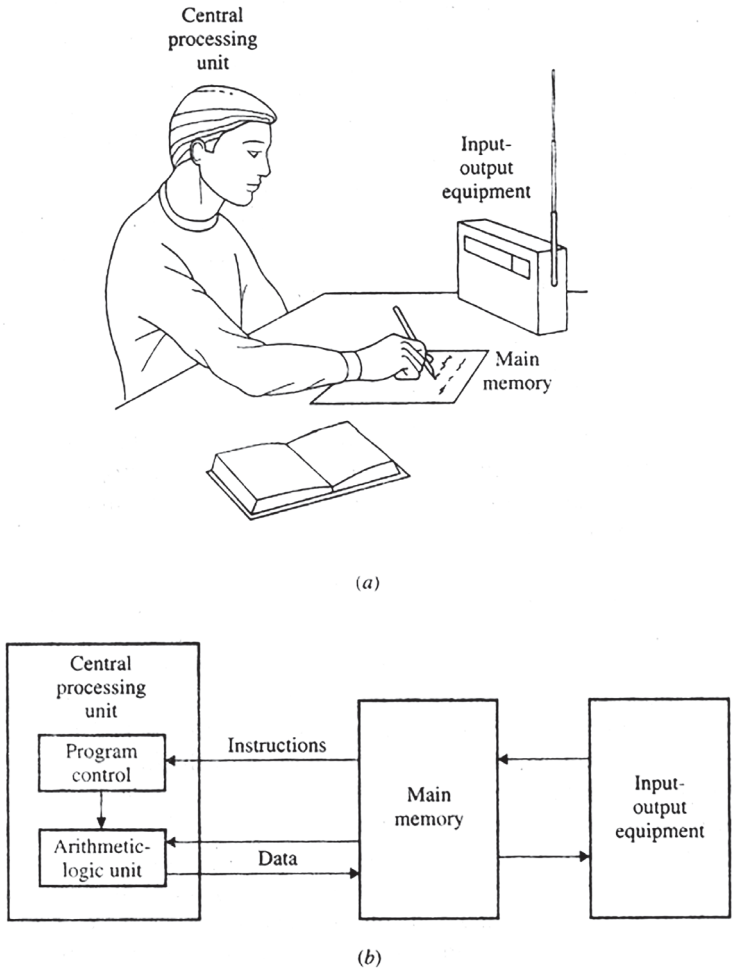

Figure 1.2: *Main components of (a) human computation and (b) machine computation.*

example, cannot be read to more than three decimal places. On the other hand, a digital device can easily handle a large number of digits. Even the simple abacus of Figure1.1*a* can display a number—admittedly just one—to 13 places of accuracy. This advantage of digital data representation over analog is also seen in the higher fidelity of the sound recorded on a compact disc (CD), a digital device, compared to an old-fashioned record (LP), which is an analog device.

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

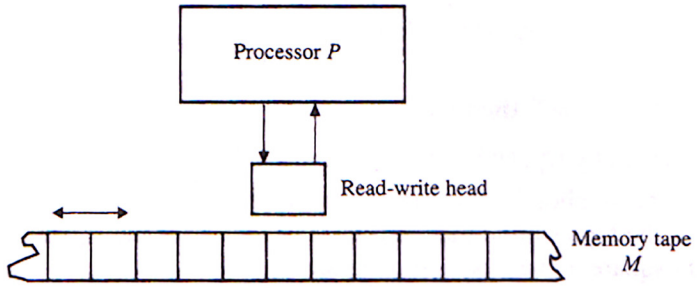

Figure 1.3: *A Turing machine.*

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

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

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

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

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

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

$$S_h\ T_t\ O_j\ S_k$$

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

to S_k. Another way of expressing this instruction, which is more in tune with the style of a modern computer programming language, is

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

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

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

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

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

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

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

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

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

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

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

Figure 1.4: *Turing machine program to add two unary numbers.*

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

1.1.2 Limitations of Computers

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1.5: *Influence of hardware technology on computing speed.*

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

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

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

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

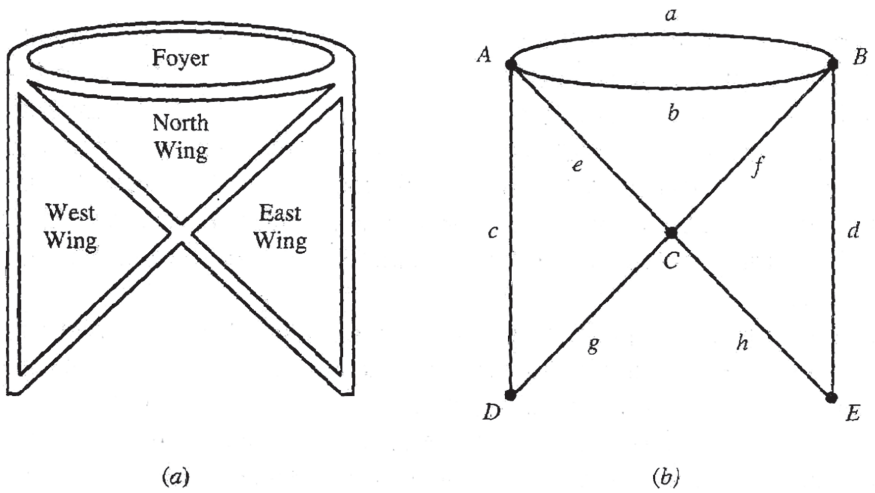

Figure 1.6: *(a) Plan of the aisles in an exhibition hall and (b) the corresponding graph model.*