# Chapter 1
# Introducing ... the Universal Computer

"I wish to God these calculations had been executed by steam."
Charles Babbage, 1822

What is this thing we call a universal computer or universal machine? We would need many books (indeed, a whole computer science curriculum) to explain all the implications of this idea. Alan Turing, a pioneer of computing, coined the term 'universal machine' when he devised a very simple abstract device—now called a Turing machine—which he argued could be programmed to do what *any* computational device could ever do. A **universal machine** is a *general purpose symbol-manipulating machine, capable of solving any problem whose solution can represented by a* **program**—*an organized set of logical operations.* At its heart, a Turing machine is capable of just a few primitive operations such as copying a digit off a piece of paper. Step by logical step, it can solve increasingly complex problems. Though the core of the universal machine needn't be complicated, it does help if it's very fast. A modern computer can perform numerical calculations "faster than a speeding bullet." Equipped with legs, it can also walk around tall buildings. (Note that a universal machine is not the legs but the controller of the legs.) It can also manipulate non-numerical data, such as text or music or speech or images. It can even learn to adapt its behavior.

## 1.1 Common misconceptions about computer science

The universal machine is a foundational idea of computer science. Many people, however, have many misconceptions about what computer science is about. Computer science explores the kinds of problems that computers can (or cannot) solve, and how to get them solve these problems in ways that are efficient and user-friendly. Yet the misconceptions remain, reminding us of a poem, "The Blind Men and the Elephant." Seven blind men encounter an elephant. Grabbing the tail, one concludes that "An elephant is very like a rope!" Another, walking up against the side, concludes "An elephant is very like a wall." Still another, holding a leg, decides an elephant is like a tree, etc. The poem concludes, "Though all were partly in the right, yet all of them were wrong." Each blind man was misled by limited information. Similarly, we find that many students have misconceptions that lead them to be intimidated or bored or turned off before they even get started. So let's address some of these misconceptions.

*Misconception: computer science is for nerds, mesmerized by computer screens*

Watch your stereotypes! Real computer scientists are generally bright people, but they are diverse, of both genders and of every race and nationality, with wide ranging talents and interests, both in and outside the field. You will meet some real computer scientists in video clips that come with the multimedia—they are quite diverse and interesting people. And, although it is true that computer scientists often like working with big monitors or the latest laptop, they also do computer science by writing on blackboards or whiteboards, doodling on paper, and especially brainstorming with other people. Computer scientists solve interesting problems. (Hence the second chapter of this book is about problem solving.) Computer scientists solve problems by designing software, or by designing theories or models, as in other sciences. The problems of computer science are wide ranging: from designing really fast computers to designing really appealing web sites, from playing championship chess to understanding human speech, from making it easier to visualize problems to making it easier for people to solve problems cooperatively.

*Misconception: computer science is for math whizzes*

It is true that computer science started as an offspring of mathematics—Turing's invention of the universal machine was originally research in mathematical logic. And it is true that in many universities, computer science started out in mathematics departments (as is still the case in many small colleges). And it is true that computers are good at computational tasks. Nowadays, though, only a small part of computing is mathematical. Would you believe that back in the 1940's, Thomas Watson, Sr., who ran IBM at the time, thought there would only be a market for at most half a dozen computers? That was because he had this misconception that only a few scientists or engineers would use computers for their mathematical computations. (Fortunately for IBM, Watson's son saw the huge potential of computing!) Today, numerical analysis is just one sub-field of computer science. Case in point: while the third author of this book got a Ph.D. in Mathematics (he *also* got a Ph.D. in Psychology), the second author studied Philosophy, and the first author was originally an English major (and makes no claims that he is a mathematical whiz). Computer science is no longer a sub-field of mathematics, nor is calculus required to study most of computer science. If you are someone who likes solving problems, and you are willing to learn a discipline of systematic and patient problem solving, computer science may be for you.

*Misconception: computer science is about computer hardware*

Although it is true that some computer engineers have the job of developing faster and more efficient machines, that's just one corner of the field. Most computer scientists see computers as tools, rather than their objects of study. It has been said that "computer science is no more about computers than astronomy is about telescopes, biology is about microscopes, or chemistry is about beakers and test tubes." A better generalization is that computer scientists study *computing*: what is it possible to do with computers? Turing glimpsed some of the possibilities when he first described the universal machine.

*Misconception: computer science is about writing programs*

As Turing discovered, writing programs is how we get computers to do new and interesting things. Nowadays we call this "software development." A whole branch of computer science called software engineering is concerned with methods for writing (and rewriting) "good" software. Still, many areas of computer science deal only incidentally with software or programming. For example, some computer scientists work on how to facilitate business processes (e-commerce), while still others seek to advance the theory and science of computing.

*Misconception: computer science is about the using computers and using computer programs*

Many people are first exposed to the field via a course in applications—word processors, spreadsheets, databases, etc. Indeed, computers and programs are put to use in a wide variety of contexts (that's one reason why we called our book *The Universal Computer*), and some computer scientists do develop applications and new tools. As with any science, however, the applications are a minor aspect of the science itself. The applications are tools for solving problems; the science makes it possible to create the tools or to imagine what tools are even possible.

All these misconceptions about computer science indicate that it is a large discipline with many sub-areas for exploration which appeal to people with a wide variety of interests and talents. We hope this text (and accompanying multimedia) will give you a bird's eye view of the whole and maybe spark your interest in further exploring some area or aspect of computing.

**Exercise 1.1:** Write down any misconceptions you may have about computer science before you began to read this book or multimedia. Discuss why you may have had them. Is the book or multimedia beginning to give you a different idea about what computer science is about? In your own words, what do you think it is about now?

### 1.2 The universal computer and virtual machines

The universal computer enables us to explore the universe. Indeed, the space program could not have reached the moon without computers. NASA developed the first microcomputer for the Apollo mission to control the complex dockings and landings of the lunar module. Can you imagine the *Star Trek* ship *Enterprise* ever "going where no one has gone before" without computers? It is not only the actual universe that we can explore with the help of these machines. In the next chapter we will investigate *conceptual* universes—"problem spaces" such as arise when one considers the range of alternative moves in a game of chess. As we shall see, an "inner" space can also be vast.

Computers also let us explore *virtual* reality. Computers can extend the senses, projecting images and sounds that change as we move, giving the strong impression that we are moving in a three-dimensional, yet imaginary world. In Japan, prospective buyers use virtual reality to preview variations on a home they might build. In laboratories, scientists use virtual reality to explore the structure of molecules. On the *Enterprise*, officers and crew enjoy virtual reality on the "holodeck"—and occasionally have difficulty distinguishing what is real from what is virtual.

Indeed, the very idea of the virtual is an important elaboration of the idea of the universal machine. While an **actual machine** *implements behaviors in hardware*, a **virtual machine** *simulates behaviors in software, in a program*. An actual machine is tangible; a virtual machine is abstract. Before you get bored with abstractions, let's see just how real they can appear to be. We've mentioned virtual reality—that can *seem* pretty real. Then there's the World Wide Web—believe it or not, it's really a very elaborate network of virtual machines. A web browser is not, after all, an actual machine, it's software—actually a collection of programs, layers and layers of them (we will learn more about these layers of programs in the chapters on computer architectures and operating systems). Are you getting the picture? Virtual machines are powerful—a powerful idea.

Turing's universal machine can, in principle, simulate any other machine—say, one that withdraws cash from a bank account on the Internet. The real beauty of this idea is that it is possible for a rather simple actual machine to simulate a *more complicated* virtual machine. It's rather like an actor (who knows no physics at all), given an appropriate script, being able to play the role of Albert Einstein. Only in computing, things get even more interesting, for it is also possible to create yet another virtual machine from a virtual machine. (Imagine an actor portraying an actor playing Einstein.) Each level of implementation creates new capabilities that the underlying machine does not have, at least not in any straightforward way. For example, an actual machine can send electrical signals across a telephone wire, yet it is unaware of what these signals mean. A virtual machine might interpret a stream of signals as a request to connect to a particular machine (perhaps via other machines) or as a request for a particular document in another machine's file system or even a request to run a program (another virtual machine) on another machine. Watch for virtual machines in this book.

The universal computer is now ubiquitous, showing up on millions of desks, laps and hands. The World Wide Web connects them to multimedia information available on other computers around the globe. Once upon a time, books were chained into libraries, because they were copied by hand. The printing press liberated books from their chains, and revolutionized society—the Reformation

spread like wildfire because Martin Luther was able to print his translation of the Bible. Similarly, faxes and computers helped undermine totalitarian regimes behind the old Iron Curtain. Nevertheless, computers are still chained to desks in university computer laboratories. What will happen when the price of computing, which has been dropping exponentially, drops to the price of paperback books? Yet there is a double-edged sword here. Though the universal computer increases communication, it can reduce privacy, enabling information about you to spread without your permission, or even your knowledge. It's the universal gossip. Whom will it tell your medical history, your overdue bills, even your taste in reading material? So the universal computer not only promises, it threatens. (We will explore social and ethical implications in a later chapter, as well as in a few exercises marked "*social/ethical*" throughout this book.)

In the next section, we briefly survey the history of the *idea* of the universal machine. It is not intended to be a full history of computing, which would probably overwhelm you. We just want you to appreciate how these great ideas—the universal machine and the virtual machines it can simulate—came about and came to revolutionize the world we live in. The last two sections will look at the practical machine itself—the hardware and the software.

**Exercise 1.2:** A personal computer (PC) has a fixed amount of memory. Windows™ acts as if it has a much, much larger amount of *virtual* memory, so that you run program(s) larger than the available physical memory. How does the distinction between actual and virtual memory illustrate the distinction between actual and virtual machines? Why is this useful to you?

**Exercise 1.3** (*social/ethical*)**:** Do you think the FBI or other federal agencies should be able to intercept data transmitted on the Internet? (The Justice Department thinks so. After all, spies and thugs can use the Internet, too.) Why or why not? Should there by laws to forbid encryption of data, thus frustrating would be spies from hiding their activities. Why or why not?

## 1.3 The Very Idea of the Universal Machine

As we have noted, a computer, unlike any other machine, is a universal machine, a general purpose symbol-manipulating machine. A car or a microwave oven or an elevator is a *special* purpose machine, designed to do a particular thing. But there is a huge range of things that a *general* purpose computer can be programmed to do (or even be programmed to *learn* to do).

### 1.3.1 "Universal Machine" vs. "Number Cruncher"

A general purpose machine is one of the great ideas of computer science. Yet it is hard to shake the idea that computers are essentially "number crunchers." The *Random House Dictionary* (1980) defines 'computer' as "an electronic machine capable of . . . highly complex mathematical operations at high speeds." Even the *Grolier Multimedia Encyclopedia* (1993) defines 'computer' as "an apparatus built to perform routine calculations. . . ." This even though a multimedia encyclopedia depends on a computer's ability to manipulate, organize and search for text, pictures, sounds and movies. Why do you suppose this narrow conception of computers persists?

The word itself gives us a hint. The venerable *Oxford English Dictionary* (1926) defines 'computer' as "one who computes, a calculator, reckoner, specifically a person employed to make calculations in an observatory, in surveying, etc." Of course, the *OED* was compiled before there were any electronic computers, when numerical calculation was done by *human* computers, but the general idea still survives.

What's the difference between a computer and a calculator? A traditional calculator can only perform a relatively small set of numerical operations; a person rather than a program determines

which calculations. (Modern scientific calculators do have some computer-like capabilities, with memory, and a limited programmability, yet they are still special purpose devices.) Though many books include a picture of an abacus as a precursor of a computer, an abacus is really an ancient calculator. Similarly, the devices invented by the 17th century philosopher-mathematicians Pascal and Leibniz facilitated numerical operations—addition, subtraction, multiplication, division—calculations, not general purpose computations.

### 1.3.2 Babbage's Engines

The 19th century mathematician Charles Babbage got so frustrated by errors in human computation that he decided to build a mechanical "computer." He envisioned a steam-powered machine that would automatically compute mathematical tables by adding increments called differences to intermediate results. Just as machines were automating tedious manufacturing operations, Babbage's Difference Engine would automate the tedious mental operations of calculations and looking things up in tables. After completing a prototype, Babbage abandoned the Difference Engine, in order to embark on what he believed was "a better idea." The Difference Engine was, after all, still a special purpose calculator. Babbage's Analytical Engine would be a *general purpose* computer. Instead of a built-in sequence of steps, it would enable a programmer to modify the squence, using punched cards. Earlier in the 19th century, Joseph Jacquard had already introduced punched cards to control the patterns woven by his mechanical looms. Different patterns of holes in the cards led to different mechanical behaviors, thus producing different tapestries. Babbage wanted to mechanize the *logical control* required to solve any formal problem. Though Babbage never did complete the Analytical Engine—if he had, it would have been larger than a steam locomotive—his design anticipated many of the great ideas of computer design.

Born at the dawn of the Industrial Age, Babbage saw first-hand the advantage of automation: a machine performs repetitious tasks without boredom, and thus makes fewer errors than does a person. Babbage enhanced the design of his machine (unfortunately driving up the cost of its implementation) to avoid human errors. Instead of relying on a person to copy the results shown by machine, it would automatically print out its results. **Automation** is a theme of modern computing: *whenever there are tedious, error-prone steps, let the computer do them.*

**Exercise 1.4** (*social/ethical*)**:** What are the consequences of increasing automation and the changes it brings for workers in a society moving into the industrial age? What are the consequences for workers in a society moving into the information age? (If you need some background, see the chapter on social and ethical issues for a discussion of these "ages".)

**Exercise 1.5** (*social/ethical*)**:** How will increasing automation, especially with computers, affect *your* career? How can you prepare, now, for the changes that automation will bring?

As a general purpose computer, the Analytical Engine would be able to carry out *any mathematical* operation. Lady Ada Byron, Countess Lovelace, who wrote the first computer programs in anticipation of Babbage's machine, wrote enthusiastically about its possibilities:

> The bounds of *arithmetic* were, however, outstepped the moment the idea of applying the cards had occurred; and the Analytical Engine does not occupy common ground with mere 'calculating machines'. It holds a position wholly its own; and the considerations it suggests are most interesting in their nature. In enabling mechanism to combine together *general* symbols, in successions of unlimited variety and extent,

> a uniting link is established between the operations of matter and the abstract mental processes of the most *abstract branch* of mathematical science. A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truth so that these may become of more speedy and accurate practical application. . . .[1]

She also envisioned *non-mathematical* applications:

> Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degrees of complexity or extent.[2]

Like the Jacquard loom, the Analytical Engine achieved its generality from *stored programs.* One could modify its behavior by altering two boxes of punched cards--one a box of variables, or data, and the other a box of instruction codes. Figure 1.1 shows how Babbage encoded four arithmetic operations as different patterns on punched cards. Thus a box of punched cards might be a program—a sequence of instructions—or it might be data—a table that the engine had already computed and punched. Moreover, Babbage saw the importance of maintaining machine-readable libraries of programs and data, anticipating the idea of *reusable* software.

Babbage's engine separated a "store" from a "mill." These correspond to the two most

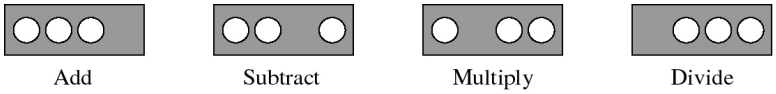



**Figure 1.1: Four cards for arithmetic operations**

important components of a modern computer: *main memory* and *central processing unit (CPU)*. Figure 1.2 depicts a schematic of a modern **stored program machine**, *which can execute any program of instructions stored in its memory*. (Babbage's schematic was somewhat more cluttered with gears and shafts.) A modern machine reads data from an external file into main memory. Similarly, Babbage's machine read punched cards representing numeric data and symbolic variables. His *store* would hold these data. He planned a capacity of a thousand 50-place decimal numbers. Babbage's *mill* read instructions from a separate box of cards. A modern machine reads both instructions and data into the same memory. The underlying principle is still the same: by reading programs which are created externally and are then stored within its memory, the actual machine becomes the virtual machine of the programmer's choosing.

Babbage saw that a few primitive operations, built into the machine, could be combined to create complex programs—different sequences of operations leading to different behavior. In this way, Babbage's mill could perform a variety of mechanical operations upon values brought in from the store. Like Babbage's engine, a modern machine performs only a small number of **primitive operations**—Add, Subtract, Load, etc.—except they are *"hard-wired" into its electronic circuits*.

---

[1]Reprinted in *Faster than Thought*, ed. B. V. Bowden (Pitman, 1953).

[2]*Scientific Memoirs, Selections from the Transactions of Foreign Academies and Learned Societies and from Foreign Journals*, ed. Richard Taylor, F.S.A., Vol. III, Article XXIX, London, 1843.

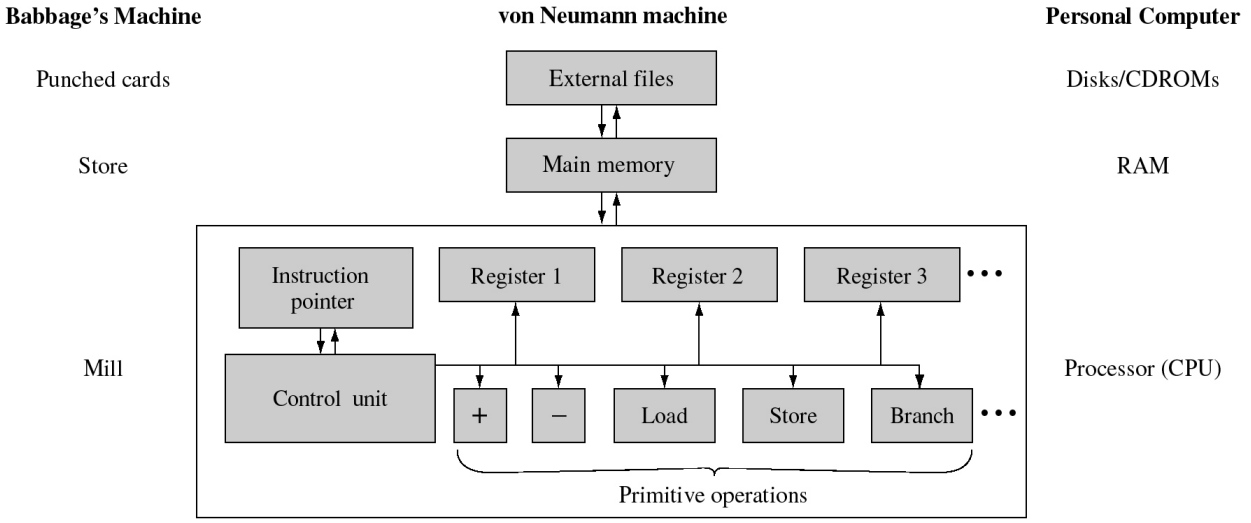



**Figure 1.2: Stored Program Machine**

(The chapter on computer architecture shows just how this is done.) The **instruction pointer** *tells where to find the next instruction in main memory*. The **control unit** *fetches an instruction* from memory, *decodes it* (for example, punched holes in the first three positions of a card meant '+') *and executes the corresponding hard-wired primitive operation the contents of special memory locations called* **registers**. For example, suppose register 1 holds the value 2 and register 2 holds the value 3. Then the '+' operation adds these values and puts '5' in a third register (called the 'accumulator'). In Babbage's machine, this operation was performed by a motion of gears and shafts. In a modern computer, it is performed by an electronic circuit. Though the latter is much smaller and faster, both implement the universal machine.

One important primitive operation is the conditional branch operation. Usually, after an instruction is executed, the instruction pointer is incremented by one so that the control unit just fetches the next instruction from memory (or the next punched card from a box). However, some instructions, called **branch** instructions, *change the value of the Instruction Pointer*, so that instead of getting the next card from the box, the control unit fetches some *other* specified instruction, in effect jumping to another part of the program. Thus, as Babbage realized, a program need not be a rigid sequence of instructions, but can jump from one card to some other card, possibly going back to an earlier one. This type of instruction greatly enhances the flexibility and power of programmable machines. Branching sets up the possibility of making decisions—choosing among alternative behaviors as well as repeating an action. Lovelace defined a "cycle" as "any set of operations repeated more than once" and noted that "the power of repeating instructions ... reduces to an immense extent the number of cards required" (*Ibid*).

Alas, Babbage never completed either of his engines (though there are prototypes of the Difference Engine in the London Science Museum and the National Museum of American History). It was nearly a century before interest in automated general-purpose computation revived—made more feasible by advances in electronics.

**Exercise 1.6:** In your own words, describe at least three important ways in which Babbage's Analytical Engine anticipates the design of modern computers.
**Exercise 1.7:** Describe at least three things modern computers can do that Ada anticipated. What does her work imply about the relationship about the great ideas of computing and practical reality?

### 1.3.3 Electronic computers

By the end of the 19th century, the U. S. Census bureau was using punched cards to process population data. International Business Machines (IBM) and other corporations began manufacturing punched-card data processing machines. These were electro-mechanical devices. Instead of steam, electrical power drove mechanical motion--punching cards and paper tape, sorting cards, and turning the wheels of an adding machine. Yet these machines lacked the generality of Babbage's unfinished Analytical Engine. In the late 1930's Howard Aiken constructed the Mark I, a fully automatic electro-mechanical machine that featured special routines to handle logarithms and trigonometric functions. It was controlled by instructions pre-punched onto paper tape. However, there was no provision for branch instructions, so the *design* of the Mark I was still inferior to that of Babbage's Analytical Engine.

World War II accelerated technological development. Gunners needed tables to predict trajectories for their shells, and each new weapons system called for new tables. So the government sponsored the development of a fully electronic computer which could generate the various tables. The resulting ENIAC (Electronic Numeric Integrator and Calculator) could add 5000 sums or multiply 300 products per second, a thousand times faster than the previous generation of electro-mechanical machines. That was just the beginning: by the turn of the 21st century, a microprocessor could perform over a billion instructions per second, and produce many millions of products per second.

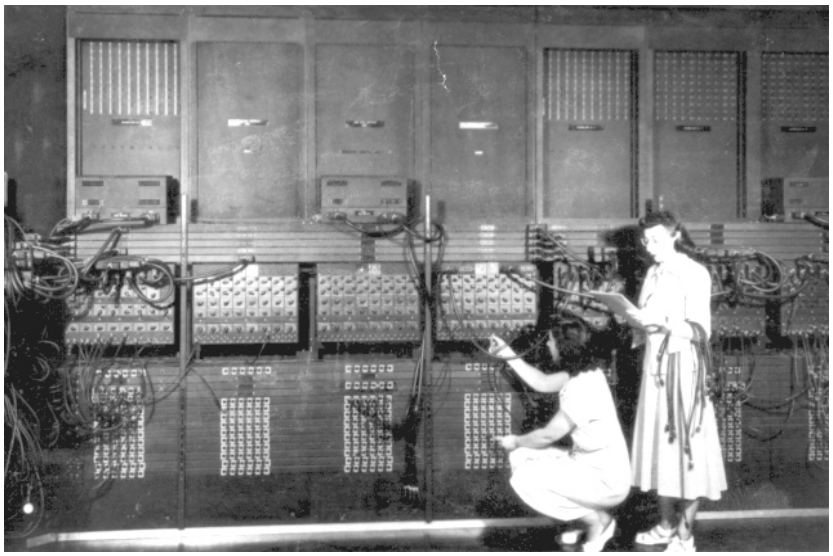
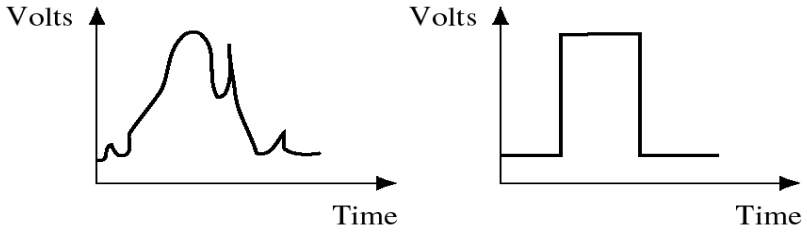


**Figure 1.3: Programming the ENIAC**

ENIAC used vacuum tubes to store data.[3] A vacuum tube is essentially a wholly electronic switch; getting rid of slow mechanical parts greatly improved the ENIAC's speed. Each vacuum tube could remember a single *binary value* or **bit**: on or off, yes or no, true or false, 1 or 0. ENIAC was a **digital** computer, *representing all information in terms of discrete on/off states*, rather than an **analog** device (such as a mercury thermometer or a volt meter), processing *continuous* quantities.

[3] A few years earlier John Atanasoff had already used the same ideas in a programmable vacuum tube device he had built at Iowa State College.




**Analog signal** **Digital signal**

**Figure 1.4:Analog (continuous) vs. digital (discrete) signals**

Figure 1.4 illustrates this distinction. ENIAC had over 18,000 vacuum tubes, connected by thousands of wires on boards, and consumed 180,000 watts of electrical power! It was said that the inhabitants of Philadelphia, the location of the ENIAC, knew when the ENIAC was turned on, because their lights would briefly dim.

ENIAC's most glaring flaw was that to program it to perform different computations someone had to unplug and replug hundreds of wires into boards. It was programmable, but tediously so! A mathematician, John von Neumann, looked over the shoulders of the engineers creating the ENIAC and suggested some improvements. Von Neumann demonstrated that a computer with a simple structure (similar to the one depicted in figure 1.2 above) could execute any kind of computation by means of *programs read and stored in memory*—hence, the **stored program machine**. This had two substantial advantages: (1) no longer would reprogramming require tedious modifications of wiring or hardware; (2) instead of accessing a card in a deck, the control unit accesses a location in memory—much faster! Von Neumann also specified a special type of machine instruction, the "conditional control transfer" (a kind of branch), and showed how to invoke subprograms: by interrupting a program, storing the machine's current state in memory, and restoring its state some time later. Frequently used subprograms did not have to be reprogrammed but could be kept intact in libraries of punched cards—just as Babbage had planned a century before—and later read into a computer's main memory. Babbage's dream had come true.

The designers of the first stored-program computer, the EDVAC, used these techniques to radically simplify its design. With just 2500 tubes, it could compute several times faster than and was far more flexible than the ENIAC. ENIAC showed the viability of electronic computing. Users were literally lining up to run scientific number-crunching jobs—only it took many hours to set up each job. Yet few people saw much of a demand for many of these machines. The first computer company—founded by J. Presper Eckert and John W. Mauchly, the two engineers who designed ENIAC—foundered. What market would there be for such expensive number-crunchers?

**Exercise 1.8:** How were von Neumann's design and the EDVAC an advance on the ENIAC?

### 1.3.4 Bits and Bytes

ENIAC actually computed with decimal values, but most subsequent computers have used binary values. Computer memory is like a vast grid of switches, each of which can only be in one of two states: on or off, 1 or 0. Each binary value is called a **bit** (short for **b**inary dig**it**). Strings of bits can represents all manner of things: from numbers and characters to colors and sounds.

Babbage's punch cards illustrate how bit string encoding works. As we saw in figure 1.1, the pattern representing an "add" instruction was three holes followed by a non-hole. We can also represent this pattern as "1110", where a '1' is a hole and a '0' is a non-hole. Similarly, we can represent each of the other instructions by a distinct binary code.

Next, let's consider numbers. You are familiar with decimal numbers, in base$_{10}$. Each

position represents a power of 10. So the decimal number $235_{10}$ = $2*10^2 + 3*10^1 + 5*10^0$ = 200+30+5. Computers use binary numbers, in $base_2$, where each position represents a power of 10. Consider the binary number $111_2$. Each place in a binary represents a power of 2: the rightmost or low order bit is $2^0$ (1), the middle bit is $2^1$ (2) and the leftmost or high order bit is $2^2$ (4). Add them up and you get is $4+2+1 = 7_{10}$. We can thus decode our binary strings by adding up the value of each place, for each bit that is '1'. Here are some more examples:

$001 = 0(2^2) + 0(2^1) + 1(2^0) = 0 + 0 + 1 = 1$
$011 = 0(2^2) + 1(2^1) + 1(2^0) = 0 + 2 + 1 = 3$
$101 = 1(2^2) + 0(2^1) + 1(2^0) = 4 + 0 + 1 = 5$

How many numbers can we represent with just three bits? The answer is $2^3$, or 8. In other words, with bit strings of length 3, we can represent up to 8 distinct values. That's not a lot of distinct values. With four bits, we can represent $2^4$, or 16 possible values. The binary number $1011_2$ is $1*2^3+0*2^2+1*2^1+1+2^0 = 8+0+2+1 = 11_{10}$. With bit strings of length 8, we could represent up to $2^8$ or 256 distinct values. Binary $01101111_2 = 0+2^6+2^5+0+2^3+2^2+2^1+2^0 = 0+64+32+0+8+4+2+1 = 111_{10}$. Because binary or $base_2$ numbers quickly get unwieldy, it is more compact to have computers print out corresponding values in ***hexa*decimal** or $base_{16}$. A hexadecimal system needs 16 digits, so by convention, the letters A through F represent the hexadecimal digits corresponding to decimal values 10 through 15, e.g., $C_{16} = 12_{10}$ and $1A_{16} = 16_{10} + 10_{10} = 26_{10}$. Figure 1.5 shows some decimal values and their binary and hexadecimal ($base_{16}$) equivalents.

| Decimal | Binary | Hexadecimal |
|---|---|---|
| 1 | 00000001 | 1 |
| 2 | 00000010 | 2 |
| 4 | 00000100 | 4 |
| 10 | 00001010 | A |
| 11 | 00001011 | B |
| 15 | 00001111 | F |
| 16 | 00010000 | 10 |
| 26 | 00011010 | 1A |

**Figure 1.5: Decimal, binary and hexadecimal number systems**

**Exercise 1.9**: What are the binary equivalents of the following decimal numbers?
a) $5_{10}$ b) $12_{10}$ c) $33_{10}$ d) $515_{10}$ e) $2050_{10}$

**Exercise 1.10**: What is the decimal equivalent for each of the following binary numbers?
a) $00000111_2$ b) $00001001_2$ c) $00100000_2$ d) $00100101_2$ e) $10011011_2$

**Exercise 1.11**: What is the hexadecimal equivalent for each of the binary numbers above?

**Exercise 1.12**: We said above that 8 bit strings can represent unsigned numeric values in the range 0 to 255. Why not 256? Alternatively it can represent signed numbers from -128 to 127. Explain how the *high order* (left-most) bit can distinguish negative from positive numbers.

**Exercise 1.13**: Suppose we wanted to represent just the four instructions of figure 1.1 in terms of

bit strings. How long would our bit strings have to be? Hint: how many instructions are we trying to represent, in terms of powers of 2?

**Exercise 1.14**: Invent a binary code for representing the 26 letters of the alphabet. How many bits do you need to distinguish 26 letters? (Hint: what power of 2 is just greater than 26?)Then assign a different pattern of holes and non-holes to each letter. Show how to encode your first name in your code. By the way, how many bits are needed to distinguish UPPER from lower case?

A **byte** is a string of bits of length 8. With a byte, we can represent numbers from 0 to 255, or from -128 to 127 (including zero). Alternatively, a one byte code could represent up to 256 distinct alphabetical characters. As exercise 1.14 suggests, the actual way we encode is arbitrary. Several codes have been proposed, but nowadays everyone accepts the ASCII (American Standard Code for Information Interchange) standard. In the ASCII character set, the character `'A'` is $65_{10}$ or $01000001_2$, `'B'` is $66_{10}$ or $01000010_2$, and so forth. There are eight bits in a **byte**, enough to distinguish $2^8$ or 256 possible values. Figure 1.6 gives a partial table of ASCII codes.

| Decimal | Binary | ASCII | Decimal | Binary | ASCII | Decimal | Binary | ASCII |
|---|---|---|---|---|---|---|---|---|
| 0 | 0000000 | Null | 7 | 0000111 | Bell | 9 | 0001001 | Tab |
| 32 | 0100000 | Space | 33 | 0100001 | ! | 34 | 0100011 | " |
| 35 | 0100011 | # | 36 | 0100100 | $ | 37 | 0100101 | % |
| 48 | 0110000 | 0 | 49 | 0110001 | 1 | 50 | 0110010 | 2 |
| 56 | 0111000 | 8 | 57 | 0111001 | 9 | 58 | 0111010 | : |
| 64 | 1000000 | @ | 65 | 1000001 | A | 66 | 1000010 | B |
| 88 | 1011000 | X | 89 | 1011001 | Y | 90 | 1011010 | Z |
| 91 | 1011011 | [ | 92 | 1011100 | \ | 93 | 1011101 | ] |
| 97 | 1100001 | a | 98 | 1100010 | b | 121 | 1111001 | y |
| 122 | 1111010 | z | 123 | 1111011 | { | 124 | 1111011 | \| |

**Figure 1.6: Partial table of ASCII codes**[4]

**Exercise 1.15**: Using figure 1.6, determine the decimal ASCII codes for the character '@', the digit '2', the letter 'Y' and the letter 'y'. Why does ASCII have different codes for 'Y' and 'y'?

**Exercise 1.16**: Go to a PC, start up an MS-DOS command window, and press the following key combinations: `Ctrl-A`, `Ctrl-H`, `Ctrl-I`, `Ctrl-M`. Note what happens for each one and explain.

**Exercise 1.17** (*explore*): Use a search engine (such as google.com) to find a complete ASCII code table on the web. Discuss two interesting characters encoded by ASCII and one by extended ASCII.

**Exercise 1.18** (*explore*): Lest you think a code for 128 or 256 characters is enough, the **Unicode** international character set uses 16 bits to distinguish $2^{16}$ possible characters. Why so many

[4]The first 32 characters of the ASCII are unprintable special codes, some of which are entered by special keys or combinations of keys: e.g., 7 rings a bell, 9 encodes tab (`Ctrl-I`), etc. You can infer the digits and alphabetical characters from relative order, e.g., the ASCII code for the digit '3' is 51, for the letters 'C' is 67 and 'D' is 68, and the letters 'c' is 99 and 'd' is 100.

characters? Hint: think globally. Use a search engine (such as www.google.com) to investigate Unicode and discuss a few interesting characters that Unicode can represent (which ASCII cannot).
**Exercise 1.19**: The ENIAC used a ring of 10 vacuum tubes to represent a decimal digit, e.g., to represent 3 the third vacuum tube would be "on" and the remaining vacuum tubes "off." How many vacuum tubes are needed to represent the numbers 0...999? As an alternative, numbers on the ENIAC could have been represented as binary numbers, where each bit would be represented by a single vacuum tube that is either "on" or "off." How many vacuum tubes would be needed to represent the numbers 0...999 in this case?

### 1.3.5 Turing's Symbol Manipulators

Alan Turing, a British mathematician, envisioned the universal machine as having much greater potential than simply being a "number cruncher." He had the key idea that a computer is essentially a **symbol manipulator**, a device which *represents and processes objects as symbols*. Words and hieroglyphics are symbols, and so are numbers—representable in terms of holes in punched cards or paper tape or switches in vacuum tubes or transistors. In fact, during World War II, Turing helped design a computer that cracked secret codes of the Nazi high command. (The COLOSSUS actually preceded the ENIAC, but was itself a secret for over twenty-five years.) Code-breaking was an early concrete demonstration of arbitrary symbol manipulation.

In the 1930's, Turing and other mathematicians were interested in studying the potential of logical computation as well as its limits. Given a set of mathematical axioms, is it possible to prove computationally whether or not a statement is a theorem? To study such questions, Turing designed a class of simple abstract machines, which have come to be known as *Turing machines*.

Figure 1.7 depicts a Turing machine. As you can see, it's quite simple: a read/write head

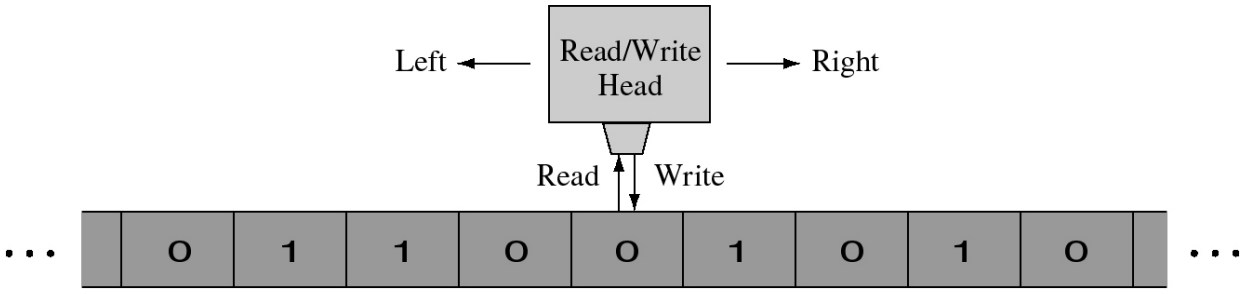



**Figure 1.7: A Turing machine**

pointing to one square on a tape of unlimited length. Each square on the tape either contains one symbol from a specified "alphabet" or else is blank. The head is in one of a number of internal states. The machine is capable of only a few simple operations: 1) *reading* what's written on the square it points to; 2) *writing* a symbol on the current square (or erasing the current symbol); 3) *changing state* from its current internal state to another (which enables it to "remember" what it has just read); 4) *moving* the head one square to the left or the right on the tape; and 5) *halting*. Each Turing machine has a set of instructions—a program—composed from these primitive operations.

Various notations for these instructions are possible; we will present them as "4-tuples": (current-state, reading, new-state, action). So an instruction (state5, X, state3,<-) represents the instruction "if in state 5 and reading an X, then change to state 3 and perform action <-, i.e., move left one square on the tape." When a Turing machine starts (in a specified starting state), it reads the square it is pointing to, then follows whatever instruction is appropriate to that state and that symbol. In most Turing machines (called *deterministic* Turing machines), there is only one

instruction for any state and any symbol, but *non-deterministic* Turing machines may have more than one instruction for any given state-symbol combination.

As a simple example, let's create a Turing machine that replaces each '1' in a binary numeral with a '0' and vice versa, using the following five rules.

(start,1,new,0); (start,0,new,1); (start,blank,start,halt);
(new,1,start,->); (new,0,start,->).

Suppose the binary numeral is 001. The machine begins in the "start" state with the head pointing to the leftmost symbol in the numeral. Here's a picture:

```
001
^
[start]
```

The second instruction is the only applicable one; it tells the machine to change its internal state to a state we have simply called "new" and replace the '0' with a '1'. The machine is now in state "new" and is looking at a 1:

```
101
^
[new]
```

Now the fourth instruction applies and tells the machine to return to state "start", and move one square to the right (i.e., to the next symbol).

```
101
 ^
[start]
```

The machine is again in the "start" state and is looking at the second symbol (also a '0'), so the same actions are repeated. As a result, the machine is once again in the "start" state and is looking at the third symbol (a '1').

```
111
  ^
[start]
```

The first instruction now applies, so the machine goes into state "new" and replaces the '1' with a '0', after which the fifth instruction applies. The machine state is changed to the "start" state again, and the head moves one square to the right.

```
110
   ^
[start]
```

It is now in the "start" state and is looking at a blank, so the third instruction tells the machine to halt. The original symbol string '001' has been changed to '110'.

Turing machines are not limited to binary code computations, since the squares may hold any symbol in the specified alphabet. Here's a set of rules that capitalizes the first letter of each word of a sentence, stopping at a period:

(start,a,new,A); (new,A,new,->); (new,a,new,->)
(start,b,new,B); (new,B,new,->); (new,b,new,->);
…
(start,blank,start,->); (new,blank,start,->)
(start,., start,halt); (new,.,start,->)

The ellipse, …, implies similar rules for the other twenty-four letters of the alphabet.

Suppose we create a tape that says **a bee sees**. Then we position a head in state "start" at

the first symbol on the tape, **a**. The first rule matches the situation, so changes the state to "new" and changes **a** to **A**; the second rule now applies, moving the head to the next position on the tape. The machine is now in state "new" and reading a blank, so the second rule from the last applies, changing the state to "start" and moving to the next position on the tape—i.e., the first letter of the second word. The fourth rule now applies, again changing the state to "new" and this time replacing the **b** with **B**. Now the fifth rule moves the head one position, so the machine is in state "new", looking at **e**. The fifteenth rule applies twice, moving the head to the blank between the last two words. Again, the second rule from the last applies, changing the state to "start" and moving to the next position on the tape—the first letter of the last word. The fifty-fifth rule now applies, changing the state to "new" and changing **s** to **S**. The fifty-sixth rule moves the head to the right, and then the fifteenth rule twice moves the head to the final letter, leaving the machine in state "new". The fifty-seventh rule moves the head to the next position (the period), leaving the machine in state "new". Now the machine reads the period instead of a blank, and the last rule changes the state to "start" and moves the head past the period to a blank. The next to last rule then halts the machine—"**a bee sees.**" has become "**A Bee Sees.**" Also note that this program will move the read head to the right forever if there is no period at the end of the sentence—fair punishment for poor punctuation! (Why does this happen? Hint: look at the third rule from the last.)

**Exercise 1.20:** Suppose that the first Turing machine is in the "start" state with an initial tape configuration of **1001**, with the head on the leftmost **1**. Trace the activity of the machine, step by step, showing how the state of the machine changes with each step, until it halts.

**Exercise 1.21:** The multimedia for this chapter includes a Turing machine simulation. After learning how to use it in both novice and advanced modes, implement and test the program described above, inverting binary numbers, with a starting configuration of **1001.**

**Exercise 1.22:** Discuss at least two things you learn about Turing machines and at least two things you learn about programming from the previous exercise.

**Exercise 1.23:** Here are the rules for another Turing machine, which inverts any string of **0**'s and **1**'s, as before, but only if the string is preceded by an asterisk. (i.e., if the asterisk is missing, it won't change the string.)

(start,*,good,->); (start,1,start,->); (start,0,start,->);
(good,1,new,0); (good,0,new,1);
(new,1,good,->); (new,0,good,->);
(start,blank,start,halt);(good,blank,good,halt).

Explain how this works. (Notice the changes from the earlier program. What is their function?)

**Exercise 1.24:** A Turing machine can use its internal states to "remember" a bit of information. Thus if you wanted to capitalize the initial letter of *every other* word, you could add another pair of states, going from "start1" to "new1", from "new1" to "start2", from "start2" to "new2", and from "new2" to "start1". Modify the program given earlier to do this. Explain how it works.

**Exercise 1.25:** Create a Turing machine, which given an alphabetic string between square brackets, encodes (or encrypts) it as a secret code. E.g., given an initial tape of **[secret]**, produce **[rdbqds]**. (Each letter is replaced by the previous one in the alphabet, where we assume **z** "precedes" **a**.) You may use the Turing machine simulation in the multimedia to implement and test your solution.

**Exercise 1.26:** The branch instruction is a crucial component of a general purpose machine, yet a Turing machine does not include branch among the capabilities of the head. How does a Turing machine get the effect of a conditional branch?

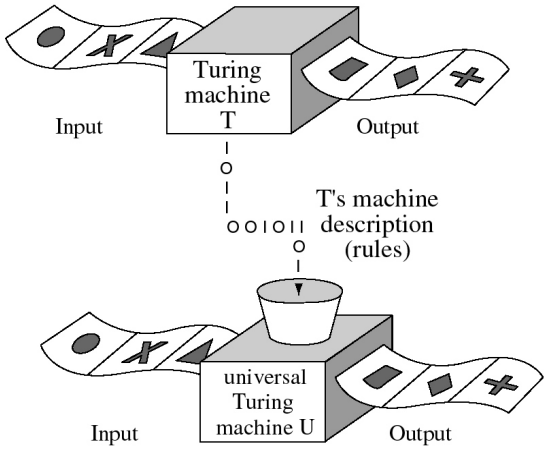



**Figure 1.8:**
**universal Turing machine**

Turing argued that these simple machines—which were not so much number crunchers as pure symbol manipulators—were capable of doing anything that is *computable*, just by steadfastly moving symbols on and off the tape. Moreover, he developed the idea of a single computational procedure (which computer scientists now call a *universal Turing machine*) which would be able to perform any computation whatsoever. Figure 1.8 may help you visualize what he had in mind.

Turing's "universal" machine, **U**, can imitate any other Turing machine, **T**. Just encode **T**'s rules onto **U**'s tape. In other words, **U** reads **T**'s rules as a program. **U** can thus simulate anything computable by any other machine **T** as a virtual machine. Turing's thesis (known as the Church-Turing thesis) is that anything computable can be computed by some Turing machine. It therefore follows that a universal Turing machine is indeed universal. It can compute anything computable.

In other words, a universal Turing machine can simulate any other computer—by running a program. It can simulate a von Neumann machine, another fairly simple machine with only a few primitive operations (but made more efficient by storing its program in high speed memory instead of a paper tape). It can simulate the processor of a Macintosh™ . The simulation of the von Neumann machine or the Mac is a virtual machine. Machine **U** isn't a Mac; it's simulating one. For that matter, there's a product that lets a Macintosh simulate a PC running Windows. For that matter, both the Windows and the Macintosh environments are actually virtual machines that can run "on top of" various actual machines.

Once you think of these machines as symbol manipulators, you see that they are capable of any procedure programmable in terms of logical operations upon arbitrary symbols. Though the word *computer* still carries the connotation of a number cruncher, one now ought to think of a computer as a general purpose machine, just as capable of manipulating text or graphical images or virtual reality as integers. Indeed, a child first meeting a computer is likely to think of it as a machine that beeps, displays colors, and plays games—not at all merely a device for performing calculations. Turing described many other examples of non-numeric problems that computers could solve, such as solving jig-saw puzzles, or playing chess. Turing speculated in 1947:

> Given a position in chess the machine could be made to list all the 'winning combinations' to a depth of about three moves on either side. This . . . raises the question, 'Can the machine play chess? It could fairly easily be made to play a rather bad game. It would be bad because chess requires intelligence. . . . It is possible to make the machine display intelligence at risk of its making occasional serious mistakes. By following up this aspect the machine could probably be made to play very good chess.[5]

You can buy reasonably good chess programs for your PC today. Just as Turing imagined, much

[5]Lecture to the London Mathematical Society, 20 February 1947, excerpted by Andrew Hodges, *Alan Turing: The Enigma* (Simon and Schuster, 1983).

of their skill derives from exploring several "moves on either side." The best chess programs compete with grand masters by examining dozens of moves on either side, evaluating hundreds of thousands of potential configurations. Are these machines intelligent, then? This was another question that fascinated Turing, one that we explore in the chapter on Artificial Intelligence.

### 1.3.6 A brief history of practical universal machines

The history of the *idea* of the universal machine really ends in the 1940's, once the first actual machines based on the theoretical machines envisioned by Turing and von Neumann had been constructed. (Note that Turing's machines were mathematical abstractions, never intended as actual machines. But his theoretical work laid the foundations for the practical machines that Eckert and Mauchley and others built a decade later.) After that, the history of *computing* as a discipline and an enterprise accelerates, becoming a history of increasingly powerful and practical machines.

Turing recognized that for symbol manipulation to be useful, it has to be cheap and accessible. Programs stored on punched cards or paper tape would be too cumbersome. For the stored program machine to be practical, it needs a lot of cheap, fast, directly accessible storage:

> One needs some form of memory with which any required entry can be reached at short notice. This difficulty presumably used to worry the Egyptians when their books were written on papyrus scrolls. It must have been slow work looking up references in them, and the present arrangement of written matter in books which can be opened at any point is greatly to be preferred. We may say that storage on tape and papyrus scrolls is somewhat *inaccessible.* It takes a considerable time to find a given entry. (*Ibid.*)

In other words, practical computers need lots of **RAM** (*randomly accessible memory*)—a vast, directly accessible store. Instead of having to wind a tape forwards and backwards, a computer can read or write to any cell in RAM at the same very high speed. The first electronic computers started the computer revolution. With each cycle of computer hardware development, the price of RAM drops an order of magnitude (by a factor of 10) and the speed of processors increases an order of magnitude. (See Figure 1.9 for a comparison of orders of magnitude.)

| Unit name | Power of two | Number of bits |
|---|---|---|
| **bit** | $2^1$ | 1 |
| **byte** | $2^8$ | 8 |
| **kilobyte (K)** | $2^{18}$ | 8,192 |
| **megabyte (M)** | $2^{28}$ | 8,388,608 |
| **gigabyte (G)** | $2^{38}$ | 8,589,934,592 |
| **terabyte (T)** | $2^{48}$ | 8,796,093,022,208 |

**Figure 1.9: Comparison of bits, bytes, Ks, Megs and Gigs**

The first electronic computers of the 1940's had 1,000 words of memory (each computer "word" can hold a number). By the 1950's, computers had 8,000 words, and in the 60's, IBM came out with a very successful line of **mainframe** computers with a capacity of 64,000 words. This

trend accelerated further with the development and increasing availability of **microcomputers**. In the mid-1970's, Intel Corporation fueled the microprocessor revolution by making a complete processor available on a single microchip, the 4004. Until then, computers were very large, intimidating machines kept in air-conditioned inner sanctums and ministered to by specially trained operators. Putting the CPU on a chip enabled hobbyists to configure their own computers. Apple Computer and Radio Shack began introducing microcomputers to a mass market. Early Apples available in the late 1970's had 8 kilobytes of RAM memory. What are kilobytes? A **byte** is a string of eight bits, enough to distinguish 256 numerical or character values. A **kilo**byte or **K** is $2^{10}$ = 1024—roughly a thousand—bytes. A **mega**byte is roughly a million bytes and a **giga**byte is roughly a billion When IBM came out with its first PC in 1981, its designers felt 64K was ample. Within a few years most PCs had 640K. There's an urban legend that Bill Gates said, "640K ought to be enough for anybody," though Microsoft's famous founder vehemently denies that now. Engineers figured out how to put thousands and then millions of bytes of memory on single components made out of silicon chips: 64K in 1985, 256K in 1987, 1M ("meg" or million bytes) in 1988, 4M in 1990. By 2000, 64M chips were commonplace, and by 2002, 256MB chips were plentiful.

Meanwhile, processor speed also began to double, and processor prices to halve about every two years. The speeds of input/output devices, such as CDROM drives and graphics screens, have similarly increased. As a result, many things that were at one time only theoretically possible because of the lack of speed or of adequate memory are now becoming practical: digitizing movies, simulating weather patterns, or recognizing human speech. Much of what Turing dreamed about has become reality.

**Exercise 1.27:** How much memory does your personal computer have, in bytes and in bits? (If it's a Windows PC, you can usually find out by pressing the keyboard combination Windows-Break.)
**Exercise 1.28** (*explore*)**:** How much is a terabyte? Why would anyone need that much data? Do a search with a web search engine to find a couple of applications for terabyte storage.
**Exercise 1.29:** Take a look at Knobby's World—it's accessible from the multimedia for *The Universal Computer* via the Tools button, or the Knobby icon in *The Universal Computer* folder. When Knobby appears on the scene, press the Run button to watch him carry a flag to the top of the mountain. Then press Reset to restore his initial state and press the Step button repeatedly to observe how he accomplishes his task. What sequence of primitive instructions does he use to pick up the flag? How is Knobby's specific program somewhat like the program of a Turing machine? What are his primitive instructions? How are they similar to those of a Turing machine?

## 1.4 Anatomy of a Computer

We now look at the components and structure of a personal computer. (The multimedia corresponding to this section will help you visualize this material.) The **hardware** or *physical components* of the machine include the central processing unit (CPU), the memory, and the peripheral input and output devices. The **CPU** or **processor** is the heart of the machine: it fetches instructions from memory and executes them, and thus *performs logical and arithmetic computations and controls the flow of data* between main memory and peripheral devices. Looking back at figure 1.2, the inner box—showing the instruction pointer, control unit, data registers and primitive operations—is a simple sketch of a CPU. The **registers** are *high-speed* but relatively expensive *storage*, fast enough to keep up with the processor. Much of the activity of the CPU is transferring data from slower **main memory** (RAM or Random Access Memory) to registers and back. The Analytical Engine had mechanical registers, the ENIAC had vacuum tube registers, and

modern computers have very small transistors etched into silicon wafers. Unlike the ENIAC, a von Neuman or stored program machine loads its programs from external files (at first on punch cards and later on disk drives) into memory. Once a program has been loaded into memory, the **instruction pointer** (or program address register) *holds the address of the next machine instruction in memory*. The CPU runs the following cycle:

fetch the next instruction from memory
decode the instruction (some binary code corresponds to some instruction)
execute the corresponding hardwired primitive operations

The primitive operations manipulate data in the registers, much as the head of a Turing machine manipulate data in the squares of a tape—the obvious difference is that a modern CPU and its registers are much faster. Faster CPUs run the above cycle more times per second. In the 90's processor speeds were typically in the millions of cycles per seconds (mega-hertz or MHz), but with the turn of the century processor speeds were measured in GHz or billions of cycles per second.

The CPU is connected by a bundle of electrical lines called a **bus** that *allows the transfer of data among the CPU, main memory, and peripheral devices*. **Peripheral devices** are so called because they stand outside the heart of the machine, yet they are the primary means of *getting information into and out of the computer*. Some devices just provide **input** to the machine, including the **keyboard**, which transmits characters to the CPU, the **mouse** and other **pointing devices** (such as trackballs, pens and touchpads), which track coordinates corresponding to locations on a screen, **scanners** and optical character readers (OCRs), which convert images into digital signals, and **microphones** which convert sounds into digital signals. Other devices only produce output, such as **monitors** (also called displays or screens) which display characters or graphical images, **printers**, which provide hard copy of data, and **speakers**, which produce audible output. To help a CPU process increasing amounts and complexity of data from these devices, many computers have special purpose processors (actually small, dedicated computers) such as video and sound cards. Special purpose programs, called *drivers*, help the CPU communicate with these devices. Finally, some devices can go both ways, providing either input or output. A **floppy disk drive** can read from or write to a diskette. These disks let you carry your data around with you. Early versions were called "floppies" because they once were flexible. Nowadays, diskettes are firm and hold 1.44M (megabytes) of data. Before you can use a disk, you must **format** it, preparing it for use. Figure 1.10 depicts how diskettes (as well as hard drives) are formatted.

This simplified drawing shows an organization of four concentric **tracks**, each with eight **sectors**. This organization enables the device to find data as requested. A typical diskette has 40 to 1024 tracks and hundreds of sectors. Once a diskette is in a drive bay, a motor spins it very rapidly, and a magnetic **read/write "head"** *detects the presence or absence of magnetic marks* on the surface of the disk. A **hard disk drive** stores information in a manner similar to diskettes. They are usually fixed inside the machine (though it is now possible to obtain portable hard drives) and contain far more data than floppies: anywhere from 10M to 100G (gigabyte or billion bytes) or more. Instead of just one disk, a hard drive consists of a stack of disks (or platters) and multiple read/write heads. An alternative to the electromagnetic technology of diskettes and hard disks is optics (light). Instead of detecting

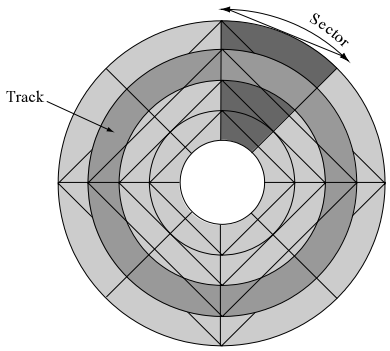



**Figure 1.10: Format of a diskette**

magnetic charges, optical storage devices detect the reflection of lasers by pits on the surfaces of platters. A **CDROM** (Compact Disc Read Only Memory) is a write-once, read-many disk; that is, a relatively expensive device writes data onto the platter just once, then less expensive CDROM drives can read it as many times as you like. CDROM disks can hold far more data than floppies or magnetic disks—over 600M in the early 1990's; DVDs introduced in the late 90's can hold nearly an order of magnitude more. As with all computer technology, the devices for writing on CDROM disks have dropped in price. At the same time the technology for creating compact disks improved so that they could be rewritten. Now, **CD-RW** (Compact Disk-ReWritable) devices are commonly included with a PC. CDROMs and DVDs are the medium of choice for delivering large amounts of data, such as multimedia.

Other input/output devices enable computers to communicate with each other. A **modem** (modulator-demodulator) transmits a computer's digital information across ordinary analog telephone lines. Figure 1.11 illustrates how modems let a pair of computers communicate. On the sending side, one modem converts digital information into analog form and puts it on the line. The other modem converts analog information back into digital form for the receiving computer. When two modems connect (usually with a loud hiss), the two sides must agree on the speed at which they will exchange data. Newer modems are capable of ever faster baud rates (a **baud rate** measures bits per second), jumping from 300 baud of the mid-80's to 56K baud in the mid-90's.

Even faster communication between machines is possible if high speed lines transmit digital

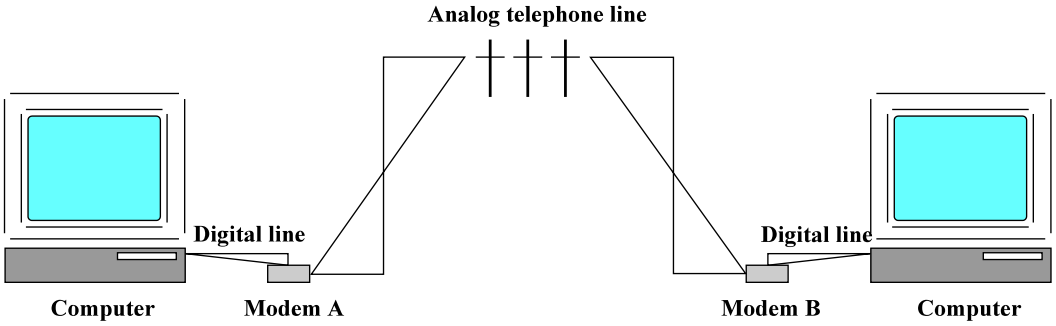



**Figure 1.12: Communication between two modems**

information. Computers connected by high speed wires form **local area networks (LANs)**, allowing them to share relatively expensive resources such as large disk drives, printers or connections to computers elsewhere. Networks of computers talking to other networks of computers give rise to **wide area networks** (WANs). The **Internet** is a very large WAN that spread rapidly around the globe. Another chapter covers computer networks in more detail.

**Exercise 1.30:** What was a key advance of the stored program machine architecture, realized in the EDVAC? What does the instruction pointer do?

**Exercise 1.31:** What is the processor speed of your personal computer, in MHz or GHz, and in CPU cycles per second? (If it's a Windows PC, you can usually find out by pressing the keyboard combination Windows-Break.)

**Exercise 1.32:** After studying the **Anatomy of a Computer** section of the multimedia for this chapter, introduce yourself to a nearby actual machine. Make a list of as many of its components as you can find and describe in a sentence or two what each component can do for you.

**Exercise 1.33:** Investigate the connection speed of a PC in your house, dorm or campus. You can

get an idea of optimal performance, in Windows™, by clicking **Start**, pointing to **Settings**, clicking on **Control Panel**, then **Network and Internet Connections**. To get a better idea of actual throughput, try using a file transfer program such as ws_ftp™, and note its report of how many bytes per second it transfers.

## 1.5 Faces of the Computer

The "face" of a computer is more than just the screen and keyboard—some of the hardware of the previous section. It's really the **software**—the programs that a computer reads into memory and executes--that enables you to interact with the machine. The earliest computers, despite flashing light bulbs put on some machines for show, were not designed for easy use. Early computers only allowed **batch processing**, processing just one program at a time. Though this was fine for many applications, a need arose for **interactive processing**, in which a machine responds to users while executing a program. With the emergence of personal computers, putting machines on the desks of schoolchildren and salespeople, a need also emerged for "friendlier" interfaces: enter Macintosh, Windows, Linux, etc.

Behind each of these interfaces is an **operating system (OS)**, *a program that manages the resources of a computer system*. An OS is a fairly powerful and complex program, doing everything from managing access to memory and controlling access to input/output devices to running system commands and other applications programs for users and other programs. Even in batch processing environments, there is a need for an OS to efficiently schedule a queue (sequence) of jobs. With interactive programs there developed an even greater need for automatic system management. After all, why should a computer wait around idly while a user is typing with one finger or even just scratching his or her head? It could just as well switch its attention to other tasks—this OS capability is called **multitasking**. Of course, if a machine has too many tasks or too many users, its services will bog down, rather like a waiter in a very busy restaurant, spending more time running between patrons and spending little time actually serving the patrons. You'll learn more about how operating systems work in another chapter.

Larger machines, such as mainframe computers handling airline transactions for hundreds of travel agents around the country, are **multiuser** systems. Workstations running the **Unix™** operating system also allow *many users*—such as students on a college campus—*to "log in" and share their resources* across a network. A personal computer, on the other hand, is usually assumed to be dedicated to a single user. **MS-DOS™** (Microsoft Disk Operating System), an operating system introduced along with the IBM PC (making a fortune for Microsoft and its founder, Bill Gates), assumes a single user will just do one thing at a time. We'll call it "DOS" for short.

Operating systems interact with users via an **interface**, *part of a program that performs tasks at the request of a user*. The multimedia associated with this section will give you a hands-on demonstration of two general kinds of "faces" that you will encounter on computers: command-line interfaces, such as older DOS or Unix systems and graphical user interfaces, such as Macintosh™ or Windows™. Here we will focus on DOS, because it is easier to explain in text than a graphical interface and because it is available from all Windows platforms. A **command-line interface** is *line-oriented, accepting one command per line* (ending with pressing the `Return` or `Enter` key).[6]

---

[6]In Windows 2000 or XP, we recommend that you launch DOS by pointing your mouse at the ***start*** button, then selecting **Run...**, then enter **cmd**. A feature of **cmd** processor is that it lets you cycle through a history of previous commands with the up and down arrows, and to edit commands using the right and left arrows.

The system displays a **prompt**—*a signal that the system is waiting for user input*—at the beginning of a line, something like this:

```
C:\ >_
```

To get the machine to do something, a user types in a **command** (shown in bold-face type):

```
C:\>format a: _
```

The DOS command interpreter will then attempt to format a floppy diskette in the drive labeled `a:` (thus preparing the diskette for further use via DOS).

A few years after MS-DOS came out, Apple Computer™ introduced the Macintosh™, with an OS that supported multitasking as well as a **graphical user interface (GUI)**, *in which a user interacts with graphical objects displayed on a screen using a mouse* or similar pointing device. Microsoft™ imitated this "look-and-feel" in the Windows™ system, and X windows (distributed by MIT) provided a foundation for portable GUIs for Unix-based systems. A GUI system displays on its screen (which one may think of as a metaphorical "desktop") a set of **icons**, *small graphical images intended to suggest their association with particular computer functions or applications*. For example, a Macintosh "trash can" icon suggests a place where one can dispose of unwanted data. (Windows 95™ remodeled this icon as a recycling bin.) The mouse activates behaviors associated with icons. For example, clicking on a Macintosh trash can reveals its contents—just in case you want to drag something back out.

The OS controls access to a computer's input/output devices, responds to commands entered via the keyboard, displays responses on the monitor, maintains data on disk drives, etc. Many OS commands enable users to organize information on disk drives into **files**, each of *which names a contiguous block of data*. For example, the file named `CMD.EXE` holds the DOS command interpreter itself. Since this (and most files ending with the extensions `.COM` or `.EXE`) is a **binary**, *computer-executable program*, humans will find its contents quite unreadable. A file named `README.TXT` (supposing such a file happened to exist on your `C:` drive) would probably be a **text file**, *containing humanly readable information*, which you could examine by entering the command:

```
C:\>type readme.txt
```

or

```
C:\>type readme.txt|more
```

The DOS command `type` displays the contents of a file on the monitor. If your file is too big to be displayed in one screen, the second line above shows how to get `type` to display one screenful at a time. The vertical bar, `|`, is a **pipe**, so that the output of one program is input to another; in this case the output of `type` as the input of `more`.

Because there may be hundreds of files on a computer's hard drive, modern operating systems provide a way to organize them into **directories** (also called **folders)**. This is analogous to how businesses organize printed information: manila folders are arranged into drawers, which are in turn filed in cabinets. Computer files are like the manila folders, subdirectories are like drawers, and directories are like cabinets. In fact, *directory systems allow users to organize data in files hierarchically*, as deeply as one likes, just as one could conceivably organize cabinets into closets, and closets into rooms, and rooms into storehouses.... In DOS, the `C:\` prompt tells the user that DOS is currently looking at the root or top-level directory. Here are a few DOS commands showing some things you can do with directories (if you are a DOS novice, you might want to get on a computer and try this):

| | | |
|---|---|---|
| 1 | `C:\>dir` | (lists content of root **dir**ectory) |
| 2 | `C:\>md mydir` | (**m**akes a new sub**d**irectory called `mydir`) |
| 3 | `C:\>cd mydir` | (**c**hanges **d**irectory to `mydir`) |

```
4    C:\mydir>dir                              (lists content of mydir—now, empty)
5    C:\mydir>md mysubdir                      (creates another subdirectory under mydir)
6    C:\mydir>cd mysubdir                      (change focus to mysubdir)
7    C:\mydir\mysubdir>copy \windows\win.ini   (copy a file from another directory)
8    C:\mydir\mysubdir>dir                     (now mysubdir has one file in it)
9    C:\mydir\mysubdir>del *.*                 (delete all files in mysubdir)
10   C:\mydir\mysubdir>dir                     (now mysubdir again has no files in it)
11   C:\mydir\mysubdir>cd ..                   (shifts up one directory level to mydir)
12   C:\mydir\mysubdir>rd mysubdir             (remove directory mysubdir—must be empty)
13   C:\mydir>cd \                             (shifts from anywhere to root directory)
14   C:\rd mydir                               (remove directory mydir—cleaned up)
```

The `dir` command on the first line will display a list of all the files in the "root" directory. Each line contains additional information about each file: its name, size (in bytes), and time of its creation or most recent modification. A line that says <DIR> indicates that this is the name of a subdirectory. The `md` (or `mkdir`, for "make a directory") command on the second line creates a new subdirectory, whose name shall be `mydir`. The `cd` (for "change directory") `mydir` command changes the active directory down to `mydir`—notice that the DOS prompt changes to show the current directory. The fourth line creates yet another subdirectory called mysubdir under `mydir`, and the fifth line shifts down to that directory. Now the prompt, `C:\mydir\mysubdir>`, shows a "path" down from the root directory, `\`, through `mydir` to `mysubdir`. The `dir` command shows that this new subdirectory has no files yet. The `copy \windows\win.ini` command makes a copy of file `win.ini` from the `windows` directory and puts it in the subdirectory, as the following `dir` command shows. (You could use `type win.ini` or `notepad win.ini` to see what it contains.) Saying `cd ..` goes back "up" a level in a hierarchy of directories; saying `cd \` always returns to the root directory. The `del` command deletes a file or files; the `*.*` uses a "wild card" notation, where `*` matches any pattern. To delete all files in the current directory, type `del *.*`. Careful, this could be catastrophic! The `rd` (or `rmdir`, for "remove a directory") command tries to delete a directory (the `del` command deletes a *file*), but in this case, it will not do so because the directory `mydir` is not empty. You must delete its contents first, and before you can do that you must delete the contents of `mysubdir`.

Needless to say, these commands only scratch the surface of the capabilities of DOS, let alone the more powerful capabilities of other operating systems.

**Exercise 1.34:** Work through and describe, step by step, two different ways to copy a text file to a new directory or folder on a floppy drive, to view the directory, and finally to print the file, one by using commands in DOS and another by interacting with icons in Windows. What are some pros and cons of these two methods?

**Exercise 1.35:** After starting DOS from Windows with **`cmd`**, enter **`dir /?`** to display a description of the command and its options (or switches). For many DOS commands running in the **`cmd`** processor, **`/?`** is an option that displays a helpful description. Describe a few interesting options of **`dir`** or other commands and how you might use them.

**Exercise 1.36:** Holding down one key then pressing another key trigger special functions in a GUI. In Windows, what do the following key combinations do: Alt-Tab, Ctrl-Esc? Many Windows PCs come with a special Windows key—what do these key combinations do: Windows-D, Windows-E,

Windows-F, Windows-R, Windows-F1? When or why might these short-cuts be useful?

**Exercise 1.37:** The authors freely admit their bias towards Windows—our bias is not necessarily indicate a preference, just what's most widely available on our campus. If another GUI is available to you, such as Macintosh or a GUI running on top of Linux, how does it accomplish some of the functions described in this subsection? How does it let you create, copy and delete files and folders (or directories)? Is a command-line interface available? Is there an on-line help system and if so what are some things you learn from it? Are there useful short-cuts for common functions? Which GUI do you prefer, and why?

**Exercise 1.38:** What happens when you put a file in a trash can or recycle bin of a desktop GUI? What's the difference when you use the **`del`** command in DOS? Why is it important to be aware of this difference? Is there a way to get the effect of **`del`** in Windows Explorer? (If you don't know, in press Start then Help, or Windows-F1, to investigate.)

**Exercise 1.39:** Suppose you want to buy a new PC and you are wondering whether to invest in a faster processor or more memory. People who know would usually advise you to get more memory. Why? Hint: think about what a modern OS typically does, such as multi-tasking and virtual memory.

**Chapter review**. Are the following statements true or false? Explain why or why not.

a) A universal machine can perform any task without anyone specifying how it is done.
b) A universal machine basically just crunches numbers and spits out fractions.
c) Automation is great for tedious, error-prone activities.
d) A stored program machine stores programs in its central processing unit (CPU).
e) A computer is hard-wired to perform thousands of different primitive instructions.
f) A branch instruction stores a value in a machine's instruction pointer.
g) A digital computer stores information in bits, which are on/off states.
h) A Turing machine has a powerful "head" which reads a finite tape faster than a speeding bullet.
i) A universal Turing machine can simulate any other Turing machine as a virtual machine.
j) RAM is a major practical improvement of modern computers over Turing machines.
k) A byte is a string of $2^8$ or 256 bits.
l) The CPU executes instructions that are not hard-wired into a machine.
j) A keyboard, mouse, monitor, printer, microphones, modems, etc., are all peripheral devices.
k) A floppy disk is ready for use as soon as you take it out of the box, just like pizza.
l) A CDROM is a write-once, read-many times medium.
m) A modem converts bits into analog form for transfer over a telephone line.
n) The Internet is a LAN.
o) An operating system is hardware that controls the physical devices of a machine.
p) An operating system is just the program that executes commands at the request of a user.
q) A multitasking operating system will let many users interact with a machine at once.
r) A command-line interface typically displays a prompt and waits for a user to enter input.
s) Macintosh™ and Windows™ are convenient command-line interfaces.
t) A file is a unit of data stored on a peripheral device such as a floppy disk.
u) Computers execute programs stored in text files.
v) Directory systems organize data in files hierarchically.

**Summary**

The **universal machine** is a general purpose symbol-manipulating machine, capable of anything one can express as a program—an organized sequence of logical steps. Babbage envisioned a machine that would perform any logical computation automatically. His machine

anticipated the design of a modern computer, distinguishing the "mill" (now called a **central processing unit** or **CPU**) from the "store" (**memory**). He also saw that one built-in instruction should be a **branch**, enabling the machine to make simple decisions that could lead it to different points in its program, possibly looping back to an earlier point. Alan Turing later provided a formal description of a simple kind of program and argued that such programs could perform any computational task. He also showed that a single program of this kind could do the work of any other program, thus introducing the concept of a universal machine and showing both its extraordinary capabilities and limitations. He also emphasized that such a machine could be thought of as more than a mere number cruncher, since it can manipulate anything that can be represented by arbitrary **symbols**. A number is a symbol, but so is a string of letters in a secret code or a mailing address or a position on a chess board. John von Neumann, observing the design of the ENIAC, proposed the design of a **stored program machine**, which would store the program(s) as well as data in memory.

Modern computers are **digital** machines, representing information as discrete on/off states or **bits**. A **byte** is eight bits, enough to distinguish 256 numerical or character values, as it does in the extended **ASCII character set**. The computer revolution has been fueled both by the rapid expansion of memory capacity and by the rapid increase of processing speed. Today's computers have random access memory (**RAM**) capacities on the order of many millions of bytes ("megabytes") and hard drives and CDROMs that can store hundreds of megabytes to gigabytes. Modems of the mid-90's could transmit data across telephone lines nearly 100 times faster than they could ten years earlier. Direct digital connections transmitting data thousands of times faster have become the main trunks of the "information highway," through which networks of machines can share complex data such as formatted chapters of books or continuous video.

Running inside all this physical **hardware** is much more malleable **software**, the programs that a stored program machine reads into memory and executes. The **operating system (OS)** is a program that operates a computer system: managing access to memory, controlling access to input/output devices, and executing system commands and other application programs for users and other programs. Though early OS's emphasized **batch processing**, efficiently scheduling a queue of jobs, **interactive processing** between machine and users has become much more important. These days the first thing you may associate with an OS is its user interface. DOS, like Unix and other earlier operating systems, features a **command line interface (CLI)**. Macintosh and Microsoft Windows have popularized icon- and mouse-based **graphic user interfaces (GUI)**. Behind the friendlier face, much remains the same, such as a hierarchical organization of data into directories of files. Nevertheless, when it comes to the universal computer, expect change! Out of the bit stream emerge symbols and programs, graphical representations of fractal computations and pictures from interplanetary robots, icons and voices and music, and more sound and fury. Who knows what will emerge next out of the bit stream?