What is a Computer?

Foundations of Computer Science

E.H. Turner and R.M. Turner

If you are reading this book, it is almost certain you are very familiar with computers. You probably own at least one (laptop, desktop, cell phone. . .), and you likely use them most days, probably for quite a few hours per day. It is possible, however, to be familiar with something without really knowing how it works, or even what is truly is. So what *is* a computer?

At one level, a computer is a tool. Like any tool, it allows us to do things better than we could without it, or even to do things we could not do at all prior to the tool's invention. Originally, and still to a large extent, computers augment our ability to do calculations: whereas we may be able to do one or two mathematical operations per second, with a computer, we can do *billions*; by ourselves, we may be able to laboriously do complex calculations, but with a computer, it becomes easy; by ourselves, and with pencil and paper, we may be able to remember quite a few data points, but with a computer, we can remember a virtually unlimited amount.

Computers do more than math, of course. In fact, they can do so many things, including controlling things in the real world, that computers have been called the "universal tool". We are all familiar with the myriad things that this tool enables us to do: write papers, carry hundreds of books in our pocket and read them, talk over long distances (cell phones, the Internet), listen to music, create music, watch movies, create movies, play video games, construct things (3D printers, robots), drive effortlessly (modern cruise control, autonomous cars), explore other planets and the deep sea, defend ourselves or attack others (torpedoes, drones), and on and on.

It's amazing to think that something so flexible, so powerful, at heart can do so very little: only few dozen basic operations, at most, and even many of these aren't truly necessary. The trick, of course, is that these operations can be done billions of times per second, and so they can be combined into extremely powerful operations—programs—that can still be done quickly. Coupled with a computer's ability to store vast amounts of information and to interact with the rest of the world, this makes computers very powerful tools indeed.

Saying that something is a tool still doesn't really answer the question of *what* it is: saying that a hammer is tool for nailing things doesn't tell us anything about what a hammer looks like, or even how it functions. So we need to look a bit harder at what a computer is.

Hardware

Computer systems are composed of both *hardware*, the physical part of the machine, and *software*, the set of programs that allow the computer to do what it does. Hardware is anything that you could touch; it's physically there. Software (and data) is generally encoded as states of hardware: magnetic domains on a disk, or voltages in memory cells, etc.

When we talk about computer hardware, we usually are talking about the components that store, transmit, or manipulate data. Other parts of a computer exist, of course: the case, the screen, the power supply, cooling fans, and so on, may all be a part of a complete computer system. But those aren't really what this book is about.

A computer, in general, consists of a *central processing unit* (CPU), *memory*, communication paths (called *buses*), input/output (I/O) interfaces and devices (sometimes called *peripherals*), and various electronic chips and circuits to support the operation of these things.

Consider a standard desktop computer. If we open the case, we will see a collection of computer *boards*, chips and other electronic components plugged in or soldered to the boards, and various wires and cables. There will also usually be several peripheral devices, or at least their controllers, inside the case, in particular one or more disk drives.

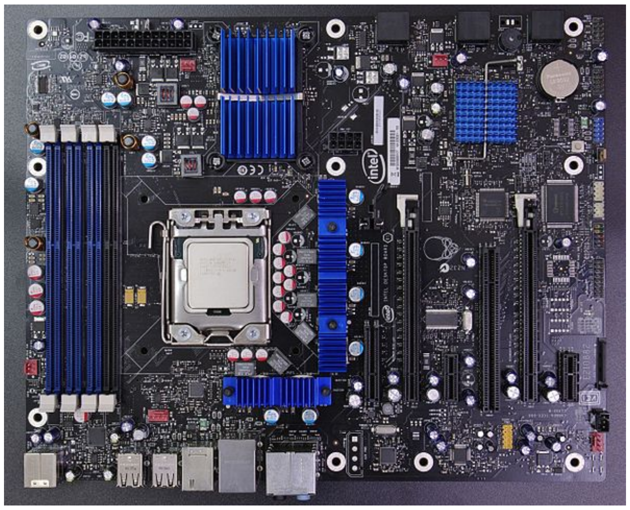
Figure 1: A motherboard. [Photo: Rainer Knäpper, Free Art License (artlibre.org/licence/lal/en/)]

There is one board that the CPU as well as other chips and boards plug into. This is called the *motherboard*. Figure 1 shows a picture of a motherboard, and Figure 2 provides a conceptual view. A motherboard contains a *socket* for the CPU, some buses for communication, some sockets for memory chips or boards containing memory chips, and sockets into which other boards can be plugged. Two important chips are also plugged into the motherboard to control access to high-speed devices (memory, graphics cards, etc.) and low-speed devices (disks, the keyboard, etc.). These are referred to as the *northbridge* and the *southbridge*, respectively, and together often referred to as the computer's *chipset*. The northbridge chip is directly connected to the CPU via a high-speed bus (the *front-side bus*), whereas the southbridge chip is connected to the northbridge via a slower bus (the *internal bus*).

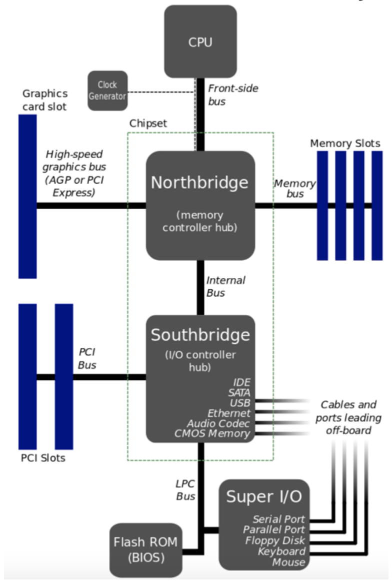

Figure 2: A diagram of the parts of a motherboard. [Creative Commons license CC-BY-SA, by Moxfyre at English Wikipedia.]

There are many different kinds of buses and I/O devices. Although we will not discuss these further here, they will be discussed somewhat in later chapters (e.g., Chapters [[chapter:computer-architecture]] and [[chapter:buses]]).

The CPU is the heart of the computer system. It is responsible for controlling all the rest of the computer and for carrying out programs. We will have much more to say about the CPU in Chapter **??**. Here, we will just mention a few highlights.

In the simplest case, a CPU can only work on a single instruction at any given time.[1] It *fetches* an instruction from memory, interprets (*executes*) the instruction to do what it instructs, then repeats—all a billion-odd times a second. This *fetch–execute cycle* is at the heart of what a computer does, and the speed with which it performs the cycle is key to its power.

[1] This is not precisely true, even for the simplest of modern processors, since they have a /pipeline/ with multiple instructions in progress at once, speculative execution of different branches, and other things we will ignore for now.

Although CPUs used to be rather large and contain many different discrete components, modern CPUs are single chips, also called *microprocessors*. The CPU contains several parts, as we will detail in Chapter **??**, including a *control unit*, an internal bus, and *registers*, which are a kind of very high-speed memory. Most CPUs these days also contain *cache*, high-speed memory that helps speed up access to the main memory. Some CPUs also contain network control circuitry, on-board graphics processing ability, and other things that are often relegated to their own chips.

One very important thing in computers is the main memory, often called *RAM* (Random Access Memory). Most of us who have bought computers know something about memory, if nothing else that more is better. The memory is where data and programs in active use are located. RAM is very fast, though usually it has the property that its contents go away when the computer is powered off. Thus it is dynamic, rather than static, memory.

RAM is generally measured in *gigabytes*. A *byte* is composed of eight *bits*, or binary digits: the fundamental unit of all data or programs in a computer. For now, you can think of a byte as being the amount of memory needed to hold one character, such as a letter or number. A gigabyte, abbreviated GB, is roughly a billion bytes (see Sidebar **??**.1). This means that a gigabyte is sufficient to hold about 2000 copies of *War and Peace*. At the time of this writing, it is common for laptops, for example, to have 8–16 GB of RAM.

RAM is also called *primary storage*, since it is the memory, or storage, from which the CPU accesses data and programs. *Secondary storage* also exists, usually these days as magnetic disks or flash memory. Secondary storage is located "further" from the CPU than primary storage; in fact, it is usually accessed via an interface card or chip of some sort, via the southbridge chip and a bus other than the internal bus. Secondary storage is usually both cheaper and slower than RAM, and it almost always has the property of retaining data even when unpowered. A typical disk at the time of writing, for example, holds a terabyte (2^{40}, or about a trillion, bytes) of data and programs, costs about 1–2% as much per byte as RAM, and is up to 10^5 times slower (depending on what is being measured). Flash storage, when used in solid-state drives (SSDs), is intermediate between disks and RAM in terms of price (about 7–8 times as expensive as disk) and speed (up to 1000 times faster than a disk).

Figure 3: Front and back of an Intel Core i7 CPU. [By Atomicbre via Wikimedia Commons, CC-BY-3.0]

Secondary storage is one kind of *peripheral device*, or just peripheral, meaning that it is not part of the CPU or memory themselves. All other I/O fall into this category as well. Aside from disks and SSDs, I/O devices can include keyboards, displays (really, graphics cards), network cards, printers, and so on. We will have much more to say about all of this in later chapters.

Digital data representation

So far, we have talked about the components of a computer without specifying much about what kind of data is operated on and, hence, what kind of hardware we are really talking about. In this book, and almost always these days in almost all other contexts, "computer" means *digital electronic computer*. There is an earlier kind that is still in limited use: the *analog computer*. And there were digital computers that were not electronic, but rather mechanical or electromechanical; we will not worry about them here, though, since all modern digital computers are electronic. But what do we mean by "digital"?

Actually, there was an even earlier kind: humans, generally women, who computed things using pencil and paper or adding machines. These people were actually called "computers", leading the first computers (in the sense we usually mean) to be called "electronic computers", much as early automobiles were called "horseless carriages".

A digital computer is one in which data is represented as numbers, i.e., as strings of digits in some number system. This is in contrast to an analog computer, in which data[2] isrepresented as quantities of some physical thing, such as voltage. Digital representation, with a finite number of digits, means that values are *discrete*, as opposed to continuous, as in an analog computer. In a continuous representation system, there is always another value between any two data values; in a discrete representation system, this is not the case.

[2] We are aware that formally, "data" is plural and "datum" is the singular. However, we will usually treat "data" as a mass noun, like "water". Just as one would say "the water is in the cup", we will say "the data is in the memory"; when referring to a particular item of data (i.e., a datum), we will usually refer to "a piece of data" or "a unit of data".

Sidebar 1: Prefixes

Some of the powers of 2 have prefixes associated with them, analogous to the way powers of 10 do (kilo-, mega-, etc.) in SI (Système International, or metric) units. Common usage is to use the same prefixes for powers of two that have comparable magnitude to a decimal number. Thus 2^{10} bits is about 1000, so that unit is called a kilobit (abbreviated Kb). Similarly, 2^{20} is about a million, so that many bits is called a megabit (Mb), 2^{30} bits is a gigabit (Gb), 2^{40} bits is a terabit (Tb), and 2^{50} bits is a petabit (Pb).

This is not an ideal situation, as there can be confusion about what someone means when they say or write (e.g.) megabyte (MB): do they mean one million bytes (when talking about a disk, for example), or do they mean 2^{30} bytes (when talking about RAM, e.g.)?

In 1998, the International Electrotechnical Commission (IEC) approved a new set of prefixes and abbreviations for some powers of two. These are similar to the SI prefixes, but with the last two characters being "bi", to be mnemonic for **bi**nary numbers. Thus, 1024 bytes is known as a kibibit, and abbreviated (Kib), 2^{20} bytes is a Mebibyte (MiB), and so forth.

This system makes perfect sense, though to us old-timers, the names sound very odd. However, human nature being what it is, the original, arguably flawed system remains in use and will likely to continue to be in the future.

Digital computers, compared to analog, are more flexible, are easier to program for complex tasks, can be constructed from a few kinds of simple basic components (though using *many* of each), and are extremely fast. In addition, programs for them are specified in a form—text—that is easy to manipulate, transport, and share. In addition, analog computers are generally not considered to be Turing-equivalent (meaning they cannot necessarily compute all computable functions).

They do have downsides, however. With an analog computer, there are no inherent round-off errors or other artifacts of *digital noise* that show up in digital computers. For example, with any finite number of digits, there are simply going to be some numbers that cannot be represented. Numbers with very large absolute magnitude, of course, spring to mind, but there are also *small* numbers that cannot be represented accurately. A simple example in base ten is the number $\frac{1}{3}$, i.e., $0.\overline{3}$. Without an infinite number of digits, this number cannot be represented exactly. Irrational numbers, such as π, cannot be represented in a finite number of digits, either. Although we may get very close to the actual number, if we have a lot of digits to use, we still won't be able to exactly represent it. Not only that, but as we perform calculations on inexact numbers, errors will mount in our answers.

An immediate question that arises is, what *digits* should we use in a digital computer? This is equivalent to asking what the number and kinds of machine states are in a particular storage element (memory cell, register cell, etc.). There is no reason in principle why we could not encode our everyday symbols, such as Arabic numerals and alphabetic characters, directly as machine states. Arithmetic could then be done in familiar decimal (base-10) fashion, and non-numeric data would be easily converted into human-readable form. We could even, in fact, use *only* numbers to represent it all by assigning a number to each character, then interpreting the number as that character as needed. For instance, we could let 97 = "a", 98 = "b", etc. This kind of direct representation of base-10 numbers is what is done, for example, in mechanical calculators.

A drawback of this is that are few if any easily-implemented 10-state electronic devices: i.e., ones that can stably take on one of 10 different states to represent the numerals 0–9. Not only that, but using the decimal digits would make electronic circuits more complex than needed, for example, to do addition or other mathematical operations.

There are, however, simple electronic devices that can stably assume one of *two* possible states. For example, a transistor can be turned "on" and "off", similar to a light switch, by applying current to one of it inputs (the *base*). Not only that, but these devices can be switched from one state to the other extremely rapidly, currently on the order of up to 100 billion times per second.

If we want to use these very fast, very simple devices, the problem then is how to use two states to encode all the information we need.

For numbers, this is easy: instead of using base-10 numbers, use base-2, also known as *binary*, numbers. It is sufficient for now to realize that any number can be represented in any base, including, of course, base 2. Many readers will have already encountered binary numbers and binary arithmetic; not to worry if you haven't: we'll cover them sufficiently for our purposes as we go. For now, just recall that whereas our decimal (base 10) number system has 10 digits, 0–9, base 2 has only 2: 0 and 1. (In base 2, the digits are known as binary digits, or *bits*.) The "places" in decimal start on the right with 1s (10^0), then 10s (10^1), 100s (10^2), 1000s (10^3) and so on. Similarly, the places in binary start on the right with 1s (2^0), then 2s (2^1), 4s (2^2), 8s (2^3), etc. Any number can be represented in either system, given sufficient digits.

It turns out that all digital computers use binary representation and hardware. It is much easier to implement electronic hardware for this number system than any other, as we will see in Chapter **??**.

Representing floating point numbers, i.e., those with a "." in them (a decimal point in base 10, a *radix point* in any base), is also done with a string of binary digits, but it is somewhat complicated and will only be touched on later in this book (in Chapter **??**).

We don't escape digital noise with binary, however; we just change which numbers we can represent exactly. While $\frac{1}{3}$ is still a repeating "binary fraction" ($0.\overline{01}$), so are other numbers that are not repeating decimals, for example, 0.2, which is the binary fraction $0.\overline{0011}$. Irrational numbers are still not representable exactly, either; π, in binary, looks like:

$$11.0010010000111111011010101010001000\ldots$$

So that's numbers. But computers can handle so many more kinds of data: text, audio files, images, movies... How are these represented in a computer?

Any data can be represented as strings of binary digits, as it happens. For example, text is handled by assigning unique binary numbers to each character, then stringing them together in some way (see Chapter **??**). Thus, "a" might be represented as 01100001_2, "b" as 01100010_2, etc. (A subscript is used to denote the base, where there may be confusion: $97_{10} = 01100001_2$, for example.) Images can be encoded in many different ways, the simplest of which might be simply assigning three numbers to each *pixel* (picture element) in the picture, one each for the intensities of red, green, and blue components of that pixel.

Thus, with one simple representation scheme—binary—a computer can encode and manipulate virtually anything.

Software

Computer hardware would do nothing without software. By software, we mean the programs that tell the CPU what to do. Software and data are both stored in hardware: they are usually represented as the states of various components (e.g., memory cells, spots on a disk, etc.)

A CPU can itself, without additional software, understand only a limited set of instructions, called *machine instructions*.[3] Each instruction is a particular sequence of binary digits, generally with different *fields*, or subsequences, that tell the CPU what to do (the operator or *op code* field) and what to do it to (the *operand fields*). The collection of all instructions that a particular CPU can understand is called its *instruction set*. Instruction sets may differ from CPU to CPU. All programs, no matter how complex, ultimately are carried out by the CPU executing sequences of instructions.

[3] We often refer to a computer as a machine, even though the quiet box sitting on our desk bears little resemblance at first glance to something like a bulldozer, or even something simple like a wheel.

It is important to realize that since instructions are just sequences of bits, a computer's programs can be stored in its memory just like data. In fact, a computer can treat its program as data or vice versa. Not only does this mean that we don't need special storage technology for software, but it also means that programs can themselves be treated as data. This becomes important when we discuss programming languages.

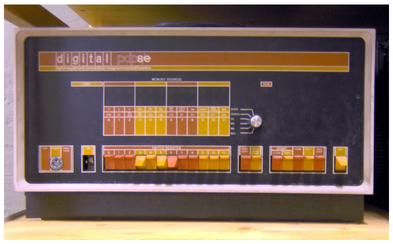

Figure 4: Front panel of a Digital Equipment Corporation PDP–8 computer, showing the toggle switches. [Photo by Arj, via Wikimedia Commons, CC BY-SA 3.0]

As an aside, a computer that stores its programs as data is called a *stored-program computer*, which all modern computers are. A computer that is organized and that operates as we have so far described, and is a stored-program computer, is also known as a *von Neumann machine*, after John von Neumann, who (with colleagues) first described such a machine.

If we consider just the hardware, that is, the "bare machine" without any other software, we can program it by storing a program composed of machine instructions in its memory, then telling the CPU to begin executing at some *address* in memory corresponding to the start of the program. From then on, the fetch–execute cycle will move the CPU's attention through the program appropriately. Another way of saying that is that the CPU's *program counter*, which contains the address of the next instruction, will change to point to the correct place in the program as it is executed.

Programming a bare machine is extremely tedious, however. In some early computers, it involved using toggle switches on the computer's front panel to input each instruction, one at a time. Figure 4, for example, shows the front panel of a PDP–8 computer; programs were entered via the toggle switches, one instruction at time, with each bit being set by a switch. Usually, there would be some other peripheral from which programs could be read, for example, a paper tape reader. The computer operator would toggle in a small program using the front panel that would then be able to read other programs from the secondary storage device.

This is an example of a *bootstrap program*, whose purpose whose purpose is to load the user's real program. (This is where the term "booting the computer" comes from, by the way.) Often, a bootstrap program was pre-loaded in *read-only memory* (ROM), thus avoiding the tedious toggling process.

Even without toggling in a program, it was cumbersome in the extreme to write programs using binary, i.e., machine language. Early on, programs called *assemblers* were written that allowed programmers to use mnemonics and familiar number representations (e.g., `ADD 3,4`) instead of binary. The assembler would then read in such an *assembly language* program and translate it to a program in machine language.

Still later, *high-level languages* (HLLs) were invented, which unlike assembly languages, have more than one (usually *many* more than one) machine instruction for each statement in the language. HLLs, such as Python, Java, C, etc., are designed to make the task of programming a computer easier for humans. Other programs then take programs written in these languages and either translate them into machine instructions or directly interpret them to carry out the operations required. This has many advantages, which will discuss later when we discuss programming languages in more detail starting in Chapter **??**.

Even with HLLs, programming a bare machine is unhandy and, usually, not very efficient. The programmer would have to include code (a generic term for a program or pieces of a program) for common tasks, such as input/output, math functions, and so forth. He or she would also have to manually sequence tasks such as translating a HLL program to a machine language program, loading the program, running the program, etc. This contributes to the inefficiency, as does the fact that while his or her program is, for example, waiting for input, the computer is doing nothing.

Luckily, few of us will have to deal with a bare machine these days unless we are working the area of embedded systems. Instead, we almost always interact with another program, the *operating system*, which controls everything about the computer. The operating system (OS) helps the programmer with the task of translating and loading his or her program, provides common libraries, and manages the computer's resources (such as memory, disk space, etc.).

As important, an OS improves the efficiency (in terms of CPU utilization). When a program pauses to wait for I/O, for example, the OS can let another program run. In fact, by switching rapidly between different running programs (also known as *processes*), the computer gives the illusion that more than one program is running at a time, even when there may be only a single CPU core.[4] For example, the laptop with which I am writing this currently has over 350 processes running at once.

[4] In this book, unless we say otherwise, we will usually consider a CPU to be able to execute only one instruction at a time, even though most modern CPUs have more than one "core", each of which functions almost like a separate CPU.

Finally, as we will discuss in Chapter **??**, an operating system determines the "look and feel" of the computer as far as its users are concerned. For example, a Macintosh seems quite different from a computer running Windows. However, as far as the hardware is concerned, they are extremely similar; in fact, most machines could run any of the major modern operating systems (macOS, Windows, or Linux). What makes the computers seem different is their operating systems. That is, the operating system plus the user-visible hardware (mouse, display, keyboard, etc.) comprises a *virtual machine*.

An OS also defines what the computer looks like to programs as well as users. This is very important for program portability and efficient programming.

Summary

In this chapter, we have given a *very* quick overview of what a computer is. In the rest of the book, we expand upon this. We order the topics roughly in order of the level of *abstraction*. While at heart, a computer is nothing more than millions of transistors and supporting electronic components, it is more useful to think of it more abstractly, for example, in terms of components called *gates* that can operate on binary data. We talk about this in the next chapter, then continue with higher levels of abstraction in future chapters: components instead of gates, programming languages instead of machine language, and so forth.

Exercises

1. What are some differences between primary and secondary storage?
2. Suppose you are able to toggle in an instruction every 5 seconds using the PDP–8's front panel.
   (a) How long would it take you, in seconds, to enter in a 4 kilobyte program? How many minutes? How many hours?
   (b) Microsoft Word on our computer takes up about 34 megabytes. How many hours would this take to toggle in? How many days? How many years?
3. (Advanced) Find an example of a computer that is *not* a von Neumann computer and describe it.
4. (Advanced) Describe how an analog computer works using information you find on the Web.