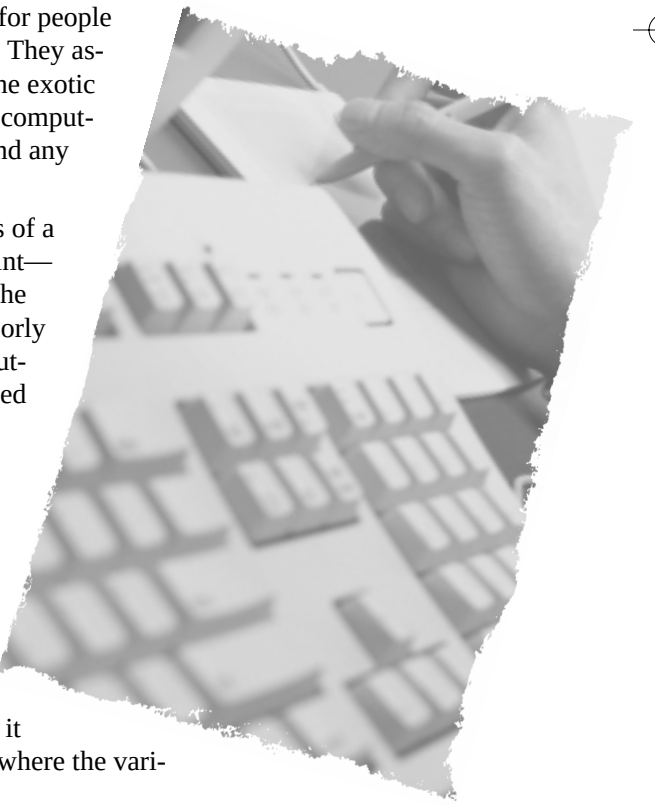
Part 1

Computer Hardware

(Chapters 1 through 4)

The four chapters in this first section will introduce you to computing's foundation—hardware. It's common for people who use computers to be intimidated by hardware. They assume that only engineers can hope to understand the exotic devices that lurk inside a computer's case. In fact, computing devices employ some fairly basic principles, and any interested person can easily understand them.



Once you're familiar with how the various parts of a computer work, you'll appreciate an additional point—there are a variety of ways to accomplish most of the tasks for which computers are used. Abetted by poorly informed sales staff, many people think that computers differ only in that some parts have a higher speed rating and others have more capacity. Not to say that speed and capacity aren't important—they most definitely are—but that's not the end of the story. For example, the simple term 'speed' is deceptive. A computer that might be very fast at one kind of task could be slow at another. Further, although it's always good to have more space on a hard disk or more memory, there are other qualifying factors. For example, what kind of memory is it? Where in the system is it located? There are different kinds of memory and where the vari-

ous pieces are placed in the system matters a lot. There are even different kinds of CPUs—the chips that do the actual processing. The speed of the chip, that famous "800 MHz!" or whatever it says on the ad, can even be deceptive. Some CPUs with slower speed ratings are much faster than their competitors at many tasks.

This last point brings us to architecture—a term that refers to both the way in which a computer's key parts are organized and also to the organization of the elements within the CPU. Both will be discussed in this section, and in the process you should appreciate that the kind of hardware you buy is increasingly dependent on the software you use. If games and graphics are your thing, then put your money in one place. If you are buying for work, you'll understand that it makes sense to structure the innards of your file server differently from that of your e-commerce Web server. On the in-chip side of architecture, we'll see that the choices available to designers, and of course therefore to buyers, are increasing rapidly. We're now entering the age of the "system on a chip," a concept that will be the driving force behind pervasive computing. But, these single chip computers will have an array of divergent designs. As more and more devices acquire computational power, it's likely that the number of distinct chip architectures will increase significantly.

To help you understand these issues, the first three chapters of this section are:

- Chapter 1: The Core of Computing: How the Key Elements of Hardware Work Together
- Chapter 2: Memory, Storage and Input/Output
- Chapter 3: Computer Monitors and Graphics Systems

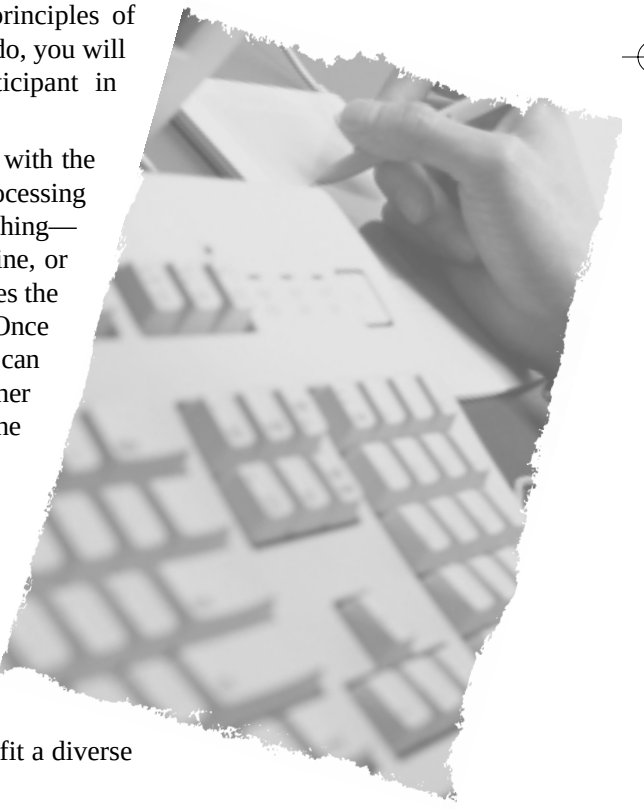
Before leaving the general topic of hardware, however, we'll move to discuss the overall issue of change. To introduce this, think of the question of flexibility. As designs in the marketplace proliferate, it's common for buyers to be advised to look for flexibility. In computers especially, much is made of the ease with which a particular model can be upgraded. While no one is going to say that you should ignore this issue in buying a computer, in fact the chances that you will significantly upgrade any computer you buy today are very small. The truth is that, if you find that you need new capabilities in a system three years after you bought it, the pace of change is such that it's almost certain that buying a new one is cheaper than upgrading the old. Why is this? The answers are in the final chapter of this section:

- Chapter 4: Silicon economics

After reading this, you'll appreciate that we could soon see the ultimate alternative to flexibility—the special-purpose throw away computer. So plunge in to the hardware pool. You'll find that it's easy to stay afloat, and once you've learned those basic principles, you'll be way ahead of your information-illiterate colleagues.

1 The Core of Computing: How the Key Elements of Hardware Work Together

Even experienced computer users have at best a sketchy idea of how a computer really works. They know how to launch and use applications. They can connect to the Internet, surf the Web, play games, or whatever. Some of the most advanced (and fearless) users are even able to install internal components, such as memory modules or network "boards." But few people actually stop to think about how a computer carries out the functions that it is asked to perform. If they did, they would be surprised at how easy it is to understand. Like Mr. Spock on Star Trek, everything in a computer is *logical.* You don't have to be an engineer or a scientist to learn how things work. And mastering these core principles of computing is worth your while. Once you do, you will become a far more knowledgeable participant in today's "information society."



This chapter deals, for the most part, with the heart of computing—the CPU (Central Processing Unit). If you tell the computer to do something—display a character on the screen, draw a line, or add two numbers—the CPU either completes the work itself or manages the parts that do. Once you understand the role of the CPU, you can quickly and easily branch out to learn how other parts of the computing system work. By the end of this chapter, you will be familiar with

- The basic components and operations of a computer
- The functions of the CPU
- How it is possible for CPUs to become faster and more effective every year
- How the CPU is being adapted to fit a diverse array of tasks

AN OVERVIEW OF HOW A COMPUTER WORKS

The best way to describe a mechanism as complex as a computer is to begin with a quick overview, sort of an orientation. This allows you to get a sense of the general flow of operations, making it easier to put the individual pieces in context as they appear. The difficulty with this approach, of course, is that you will encounter in this quick introduction some terms and concepts that may not be completely clear right away. Since the overview can't explain everything in depth—that would defeat the purpose—this section will include a number of the "Tech Talks" that are used throughout the book to explain key terms. If that's not enough, remember that more detailed explanations will follow quickly.

What Happens When a Computer Starts Up

The following section describes, in highly simplified form (and slightly rearranged for purposes of illustration), what happens when a typical microcomputer *boots up* (starts). The various parts of the computer are introduced as they appear.

Tech Talk

Booting: Computers are described as *booting up* after the observation of early designers that the system should "pull itself up by its bootstraps."

Power On

Once power is turned on, electricity flows through all of the chips and their circuits. Most components just sit and wait for instructions, but one chip, called a *ROM BIOS* (Read Only Memory, Basic Input/Output System, usually abbreviated just *BIOS*), is designed to begin giving commands as soon as it receives power. The BIOS contains an entire set of instructions, in effect a computer program written into the chip, that manages the boot-up process.

Tech Talk

BIOS: The Basic Input/Output System (BIOS) refers to a chip in some types of microcomputers, especially those that use Microsoft operating systems. The BIOS, which can be thought of as an extension of the operating system, holds information about attached devices such as disk drives, external buses, etc. The BIOS also helps systems boot up. See Chapter 7 for more detail.

The CPU Begins its Work

The first stage of the boot-up process verifies that all components are working properly. When that's complete, the BIOS hands over control to another chip, the CPU (central processing unit). Unlike the ROM, which simply holds instructions, the CPU is a microprocessor. As a processor (the micro just means that it is on one chip), the CPU has two distinct capabilities.

Tech Talk

ROM [vs. RAM]: Read Only Memory (ROM) is a kind of chip that has information permanently burned into it. Since ROM is nonvolatile, this is a handy way to provide the computer with information that has to be readily accessible but doesn't need to change. Early BIOSes used ROM, though Flash memory, which is also nonvolatile but is faster and can be rewritten, is now preferred.

Tech Talk

CPU: The Central Processing Unit (CPU) is the part of the computer that carries out the actual computation—arithmetic and logical operations. In the earliest computers, the CPU used many chips, but today's computers nearly all use microprocessors in which all functions are on one chip.

1. The CPU can carry out a variety of mathematical and logical operations. Its basic mathematical abilities include the traditional addition, subtraction, multiplication, and division. Building on these, it can also do any other kind of mathematics. The critical logical abilities of a computer focus on comparison—for example, it can determine if two numbers are equal. This seems a simple task, but it is one that is crucial to computer programs. Finally, while the CPU does all of its mathematical and logical operations on numbers, those numbers can be used to represent things that aren't numerical. For example, the letters of the alphabet, or objects such as lines in drawings, can each be given numerical codes that identify them and set them apart from others. As a result, the computer can work with text, graphic images, and so forth.
2. The second core ability of the CPU is to manage, in a very intelligent way, the flow of information (instructions and data) into and out of its circuits. We'll see why this is important when we turn to the next stage of the boot-up process.

Addresses Come into Play

The last instruction that the ROM sends to the CPU is to go to a specific location to find the next instruction. In telling the CPU where to look, the ROM provides it with what is called an *address.* Computer addresses, like those on letters, are simply directions to where something can be found. Like everything else inside the computer, addresses are numbers. We'll see later that the size of the number matters. Since CPUs use these addresses to create what amounts to a very sophisticated filing system, the bigger the number, the larger the *address space*—the maximum size of the file system.

Tech Talk

Address: Computers use addresses to keep track of information in much the same way as the post office uses them to find residences and businesses. The bigger the number in an address, the more locations it can refer to. Most current computers use a 32-bit *address space* for memory, which means that there can be over four billion separate locations to hold information. See Chapter 5 for more about addresses.

The address of the instruction the CPU is to carry out enters its chip on a set of wires that is called a *bus.* The bus that is used in this case is known as the *data bus*. This is a path that is able to carry information both into and out of the CPU's chip (see Figure 1.1).

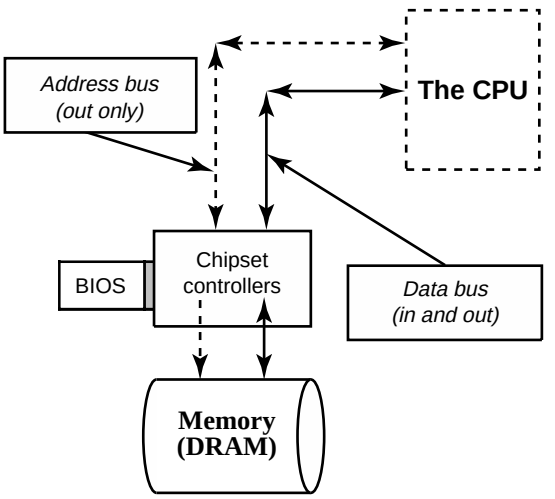


Figure 1.1 Connections through the chipset.
The address and data buses (together known as the system, or memory, bus) connect the CPU to the chipset (of which the BIOS is sometimes considered a part) and from there to memory (DRAM).

To continue with our example, the address given to the CPU by the BIOS is for something the CPU doesn't have on board. As a result, the CPU promptly puts the address on another bus, in this case the appropriately named *address bus.* When the CPU does this, it is called a *fetch.* The address bus in effect carries a request for help—the CPU is asking other parts of the computer to provide it with needed information. Unlike the data bus, this connection goes only from the CPU to the outside—nothing comes in on the address bus.

Tech Talk

Bus: A bus is a pathway that carries information between two or more parts of a computer. There are internal buses, such as the system (memory) and I/O buses, and external buses, such as USB or FireWire.

The Role of Memory

The address bus connects first to a part of the computer called *memory.* The term memory refers to a class of silicon chips that are able to hold instructions or data. The CPU can either *read* information from memory or *write* to it. The most common type of memory chip is called DRAM (Dynamic Random Access Memory). Current DRAM chips can hold 64 million bits of data (see below for an explanation of *bit*) which, at ~5,500 bits per page, translates to some 12,000 pages of simple text per chip. In most computer systems, DRAM chips are arrayed in groups of eight, with the result that they can hold quite a bit of information. Unfortunately, DRAM chips are *volatile,* which means that they need electrical power to retain their contents. Since the computer has just been turned on in our boot-up example, this means that the instruction the CPU seeks isn't to be found in DRAM.

Tech Talk

DRAM: Dynamic Random Access Memory (DRAM) is a type of chip-based storage that has the advantage of being reasonably fast and inexpensive, but the disadvantage of being *volatile*—losing its contents when electrical power is cut off.

The Chipset

In order to get to memory, the address bus has had to travel through a small group of chips known as the *chipset.* These chips (the number varies according to the type and brand of computer) perform what is known as *glue logic,* which is to say that they bind together, both physically and functionally, the various pieces at the core of the

computer. Their importance is illustrated by the fact that their individual names usually include the word *controller*, as in *memory controller*.

Tech Talk

Chipset: The chipset refers to a group of chips that provides an intelligent interface for the core components of a computer—CPU, memory, graphics, I/O system. Described as *core logic* or *glue logic.*

Back to our boot up example. When the instruction can't be found in memory, the chipset redirects it to another bus, known as the I/O (Input/Output) bus, that connects the chipset to other places where information is stored.

The I/O Bus and the Hard Drive

In a typical computer of today, the chipset will send the address over the I/O bus to what is called the *hard disk* (see Figure 1.2). This device, which consists of a series of magnetic platters, operates in a physical sense very differently from memory, but has the same functional role of allowing the CPU to read and write information. Why does the computer have two places—memory and disk—that have the same function? The answer is that the two complement each other. Memory operates very fast, but has a high cost per unit of data stored and is volatile. The hard disk, on the other hand, is much slower, but has a low cost per unit of data maintained and is not volatile. As a rough measure, for the same cost, the hard disk provides fifty times as much storage as memory. To get the best of both worlds, most desktop and similar computers combine chip and disk storage and operate them in a hierarchical fashion,

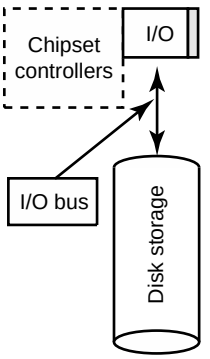


Figure 1.2 Disk I/O.
Disks connect to the CPU via the I/O bus, which is in turn connected to the chipset. See also Figure 1.1.

moving information from the slowest level to the fastest in anticipation of need. We'll see further examples of this as we follow the boot up process. Figure 1.3 provides a description of the storage hierarchy.

The disk receives the address, finds the information, and sends it to the CPU. After going through the chipset, it is put on the data bus to get into the CPU. One of the functions of the chipset is to "bridge" the two buses. Figure 1.2, which is an extension of Figure 1.1, shows the I/O bus connecting to the chipset.

The Storage Hierarchy

The next action taken by the disk reveals why hierarchical storage works so well. Once the originally requested information is sent out, the disk proceeds on its own to retrieve and send to memory information at addresses that immediately follow the requested address (usually these are grouped together in a single *file.* It does this because computer programs are organized in a sequential way. This means that after the CPU has done what the requested instruction tells it to, *executed* it, it will most likely ask for the instruction at the address following the previous one. If this information has been copied to memory, it can be fetched and executed much faster than if the CPU has to go to the disk each time. When we get into the details of CPU and memory system operation, we'll see that this process of anticipating which information

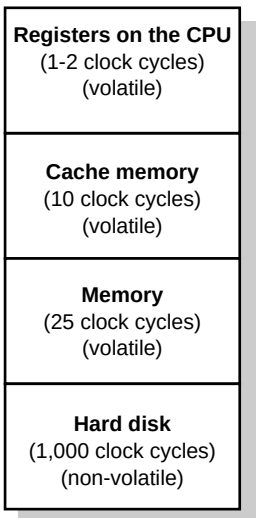


Figure 1.3 The storage hierarchy.
The storage hierarchy, showing how many clock cycles the CPU has to wait to get information from each device (in very rough equivalents).

will be needed next, and then passing it up the hierarchy to a more rapidly accessible area, extends to the fastest kind of storage, a special form of memory known as *cache.*

The Computer Begins to Work

Once the BIOS has finished telling the CPU what to do, the computer is almost ready to go to work. Almost, because it can't use what are known as *applications* (software that the user controls, such as spreadsheets and word processors) until it has loaded its own software, the operating system.

The Operating System

The first instruction that the CPU executes is one that tells it how to begin the process of loading a key component of software—the operating system (OS). Just as the instructions contained in the BIOS manage the boot-up process, the OS is software that controls the operation of the computer once it is running. Key functions of the OS include apportioning memory and CPU time among the application programs that the user has launched, maintaining connections with attached devices like monitors and printers, providing a user interface (things like windows and menus), and managing network links. We won't discuss the OS further here—software is covered in Part 2. Instead, let's move on to see what the CPU does.

Four Stages within the CPU

Inside the CPU, things happen in a four-step sequence: fetch, decode, execute, and store. You'll recall that the first action that the CPU performed back when it asked for information at an address was called a *fetch.* Since the CPU relies on the outside world for instructions and data, a fetch operation always has to be the first. The CPU's fetch unit plays a role in the computer's hierarchical memory system because, like the disk we saw earlier, it anticipates what the CPU will need and goes ahead and gets the instruction from the next address in a sequence. CPUs actually have a *prefetch* area that performs this function.

The second step taken by the CPU is to *decode* the instruction—decide what to do with it. Various things can happen here. For instance, if the instruction itself requires data (for example an instruction that says to add the value at address xx to the value at address yy), then a request to fetch the data will have to go out on the address bus. Another key task of the decode stage involves preparation for the next stage—picking the appropriate circuits needed to execute the instruction.

Execution, the third step, is where the action occurs. This is the stage where numbers are added, multiplied, compared, and so on. Execution is carried out by a part of the CPU called the *ALU* (Arithmetic Logical Unit). When we get into the details of CPUs, we'll see that if you're an engineer with a big enough "transistor budget," you can design an ALU that will really make things move.

Once a calculation is complete, the fourth step, *store*, occurs. If the instruction at hand is one of a series—for example, add x to y, and then add the result to z—its result will be stored on the chip in areas called *registers*. Registers are a kind of scratch pad that can hold the results of one calculation while waiting for the next one to be completed. Registers are at the very top of our memory hierarchy. They can be accessed much faster than other kinds of memory—a thousand or so times faster than disk storage. However, registers are used only for temporary storage. In the event that an instruction is completed, its result will be sent back to memory or to disk.

Keeping Things under Control

The descriptions of computer operations provided above have focused on the basic flow of operations. While they aren't as visible, the ways in which the system keeps its work organized and stable are important to mention. The most important control mechanism is the use of timing. The CPU uses a clock, a quartz crystal like that used in watches, to coordinate operations on its chip. Rigid adherence to the dictates of the clock allows everything to stay synchronized. Other chips, as well as the various buses, all use timing mechanisms of one kind or another.

Additional examples of system management include the control section of the CPU, which manages the paths that instructions and data take, and the control bus, which, among other things, shifts the direction of the data bus between input and output as needed. There are also control lines in the I/O bus. In addition, note that computers often distinguish between the end of one instruction and the beginning of another by adding special codes that always indicate the start or the end of a piece of information. This approach is widely employed in telecommunications, but is used as little as possible inside the computer because the redundant information slows things down. Instead, the focus inside a computer, especially inside the CPU, is on keeping everything on the same time schedule.

Connecting the Parts

Thanks to the continuing shrinkage of the microprocessor, there are many different kinds of computers. The differences between the principal types are discussed in Chapter 4. Since this overview has focused on desktop computers, also called micro-

computers, there is one other major part that needs to be introduced. The components that have been mentioned previously—the BIOS, CPU, memory, chipset, buses, hard disk—are all attached to something called the *motherboard.* The motherboard serves two purposes. On the one hand, it is a circuit board that assists in connecting devices—most of the buses are actually built into the motherboard. For this reason, the motherboard is a fairly high-tech product, one that requires careful manufacturing. On the other hand, though, the motherboard is just a board—the surface that provides a resting place for the various parts, including ones we haven't discussed, like the power supply, the fan, and the case. In today's microcomputer world, there are a few standard *form factors* (external sizes) that most companies use for motherboards. This simplifies design and allows cheaper mass production.

Binary and Digital

Our preview wouldn't be complete without some explanation of the way in which computers use numbers. All modern systems use a form of mathematics that is binary, which means with only two options. Instead of 10 digits (0–9) as in the decimal mathematics of modern society, computing systems use only two digits: zero and one. The same numbers can be represented in both binary and decimal systems, so why choose binary for computing? The answer is that using just two values corresponds nicely to the way electronic systems work. It's quite easy to design an electronic machine that will reliably differentiate between a zero (the absence of an electrical charge) and a one (the presence of a charge). By contrast, it would be much harder to create a system that could make a reliable distinction between ten levels of electrical charge in order to correspond to the ten digits of the decimal world. Binary numbers are also easily manipulated in electronic circuits—they can be added, multiplied, compared, etc. with fairly simple designs. Since even in the beginning electronics were very fast and inexpensive (by comparison to calculating machines with gears and levers), scientists and engineers decided that it was worth the modest effort required to convert between decimal and binary in order to have the benefits of electronics.

Tech Talk

Binary: Binary mathematics is a form that uses only two digits vs. the ten in our normal decimal system. Binary numbers are used in computing because they conform very well to the *on* and *off* states of electronic devices.

Tech Talk

Analog: An analog system uses a representation of information rather than a numerical version in its processing. For example, traditional radio sends and receives sound with an electronic wave that is an analog of

the voice, music, or noise waves that enter a microphone. Analog systems are well suited to carrying information, but not to modifying it.

Tech Talk

Digital: A digital system is one that translates all of the information it works with to numbers—*binary* numbers in the case of computers. Compare to *analog*.

Binary and digital systems gave birth to the term *bit*, which is a contraction of *binary* and *digit*. A bit is a single digit that can be either one or zero—in other words, it can hold just two values. But, by putting bits together in a row, you can get to bigger numbers. How big is easy to calculate—you just use the number of digits as a power of two. This is illustrated in Table 1.1 below.

When computers were first commercialized, engineers needed a standard binary value to use for the exchange of information both inside and outside of the machine. They chose eight bits since this binary number can represent 256 values and therefore can describe all of the things needed for normal commerce—every letter of the alphabet, all normal punctuation, all of the decimal numbers, and a bunch of special symbols. Thus, the term *byte*, shorthand for any eight-bit number, became the standard unit of measure in computing. IBM really made the byte a standard. Its early business computers were designed to manipulate one byte at a time in their internal circuits.

The capability of a computer in bits is a way of describing the width of its circuits. If you look at Table 1.1, you can see that moving an entire byte at once, in parallel, requires more wires than moving just one bit. So, if a computer is described as 8-bit, this means that there are eight wires that move information from place to place

Table 1.1 *Binary Values*

Binary Values		
Bits used/values in powers of 2	***Values expressed in binary form***	***Total values***
1	0, 1	2 (2^1)
2	00, 01, 10, 11	4 (2^2)
3	000, 001, 010, 011, 100, 101, 111, 110	8 (2^3)
4	0000, 0001, 0010, 0011, 0100, 0101, 0111, etc.	16 (2^4)
8	00000001, 00000010, 00000011, etc.	256 (2^8)
16	0000000000000001, 0000000000000011, etc.	65,536 (2^{16})
32	00000000000000000000000000000001, etc.	4,294,967,296 (2^{32})

Table 1.2 *Bits and Bytes*

Bits and Bytes				
	Thousand	**Million**	**Billion**	**Trillion**
Bits	Kbit	Mbit	Gigabit	Terabit
Bytes	KB	MB	GB	TB

in the CPU, that the circuits that do mathematics and logic are eight transistors wide, and that registers can hold eight bits at once. The various buses can also be described in terms of width. In today's desktop computers, the standard circuit size is now four bytes, or 32-bits, and all internal CPU structures are that wide, as are the various buses (some are even wider).

The common way of referring to large numbers of bits and bytes is to use a standard prefixing: *kilo* (or just K) for thousand, *mega* (M) for million, and *giga* (G) for billion, and *tera* (T) for trillion. If the prefix is in front of *bits*, it will normally be written out (e.g., 1 Mbit refers to a million bits), although a lowercase *b* is also used (Mb=Mbit). If it is in front of bytes, it will normally just get a capital *B* (1 MB is a million bytes).

With this overview behind you, you're ready to get into the core of the computer, the CPU. Until Intel's "Intel Inside" advertising campaign started in the mid-1990s, many computer users didn't even know they had a CPU, let alone how it did its work. But since CPU design is going through some major changes, ones that will present users with a lot of choices in the near future, it makes sense to get acquainted with this most essential part of computing.

THE INTERNAL OPERATIONS OF THE CPU . . . .

As noted earlier, the CPU is also called a *microprocessor* because all of its components—at least all those needed to carry out calculations—are on a single silicon chip (see Chapter 4 for a discussion of how chips are made). We'll begin our overview of the microprocessor with a simple description of its functions. The classical CPU includes a four-sequence operation (see Figure 1.4).

1. Fetch
2. Decode
3. Execute
4. Store

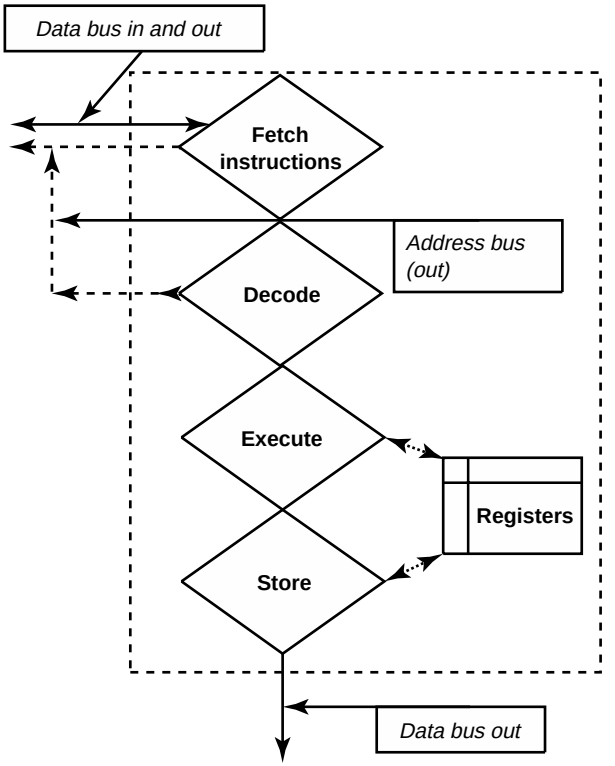


Figure 1.4 Four stages of CPU operation.
The four steps completed by a CPU include fetching information (instructions and data); decoding instructions to determine what to do with them; executing instructions (for example adding numbers); and storing the result of the instruction either on-chip in a register to await another instruction or back in memory or storage if immediate access isn't needed.

Fetch

First, instructions and data are fetched from outside the chip (usually this means from DRAM). An example of an instruction is one that contains a simple mathematical operation such as "add." In some cases, the data used by an instruction are included with it; in other instances, the instruction references the locations where the data are held. These locations are called addresses. So, an "add" instruction might carry the numbers to be added with it, or it might state that the value in *address x* is to be added to the value in *address y*. An instruction actually contains two parts: an *opcode*—the action to be performed—and the *operand*—the data it works on. So, "add" is an opcode, and *address x* and *address y* are operands.

Decode (Analyze)

Once the CPU has received the instruction, it is turned over to an area of the chip that decodes (analyzes) it in order to determine which of the chip's circuits should be used for processing. This analysis stage can also include other functions. For example, some chips will look into the stream of incoming instructions to reorder them so they can be completed in the most efficient way possible. Also, in the event that the instruction does not include the actual data that will be used, but just their addresses, this is the point at which the CPU will retrieve the data.

Execute

If the CPU is the brains of the computer, then the ALU (arithmetic/logical unit) is the part where the actual thinking (*execution* of the instruction) takes place. The ALU includes groups of transistors, known as *logic gates*, that are organized to carry out basic mathematical and logical operations. An appropriately arranged collection of logic gates can then execute a complete mathematical instruction (such as "add" or "divide" two numbers) or a logical instruction (such as "compare" two values). The instructions that a specific ALU can execute are known as its *instruction set.* CPUs from different vendors (e.g., Intel, Sun) have different instruction sets. Problems of systems compatibility begin at this level. Software written to the Intel instruction set won't work (at all) on other types of processors such as Sun's SPARC family or the IBM/Motorola Power PC series. Like different languages, instruction sets vary in both their vocabulary and their grammar (for example, they use different ways of organizing instructions).

Tech Talk

Logic gate: A logic gate is a series of transistors connected in a way that allows it to carry out a mathematical or logical operation such as addition. Logic gates are grouped into electrical circuits that execute the CPU's instructions such as to "add" two numbers or "compare" two values.

REGISTER HERE

Registers provide a good example of the connection between hardware and software. Intel's original 16-bit CPU, the 8086/8088, had only eight general purpose registers that programmers could use to hold data and instructions. As more room became available on the

chip, it would have been logical to increase this number, since in most cases more registers means better performance. But Intel couldn't do this because it would have made newer chips incompatible with older "legacy" software. The number of general purpose registers finally increased—to 48—when Intel completely rearranged the architecture of its chips with the sixth generation (Pentium Pro) in 1995. Significantly, these CPUs included a mechanism for translating old register instructions to new ones.

Store

Instructions have to tell the CPU not only what operation to perform, but where to put the result. There are a number of options. If the instruction is *iterative*, for example adding two numbers then adding another to the result, the instruction will tell the CPU to place the product of the first addition in a special short-term on-chip storage area, known as a *register*, until it is needed. Because the registers are interwoven with the ALU's circuits, they allow very fast retrieval. Alternatively, if the result is not expected to be used again right away, it will be sent off-chip to memory (fast retrieval) or to disk storage (a lot slower—see Figure 1.5 for an overview of these parts).

Tech Talk

Register: A register is an on-CPU storage space where instructions and data can be transferred and held temporarily for fast retrieval.

Recent changes in computer architecture have simplified the way in which the CPU manages instructions. In the old style, it was common for a programmer to tell the computer to "add the value at memory location xx to the value in memory location yy and place it in memory location zz." The problem with this approach, often called *fetch/execute*, is that operations that need to access memory are slow. The CPU has to wait for many clock cycles while the information is looked up and retrieved. Current system strategy limits memory accesses to *load* and *store* operations. In this approach, the programmer will first give the computer instructions to load the appropriate data from memory to a register or registers. Then, an *add* instruction would be something like this: "add the value in register a to the value in register b and place the result in register c." Subsequently, another instruction could tell the CPU to "store the value in register c in memory location zz."

The value of the *load/store* approach is that every operation doesn't have to begin with a *load* since some or all of the data to be manipulated are already in registers. In considering this strategy, you will appreciate that it is important to have a lot

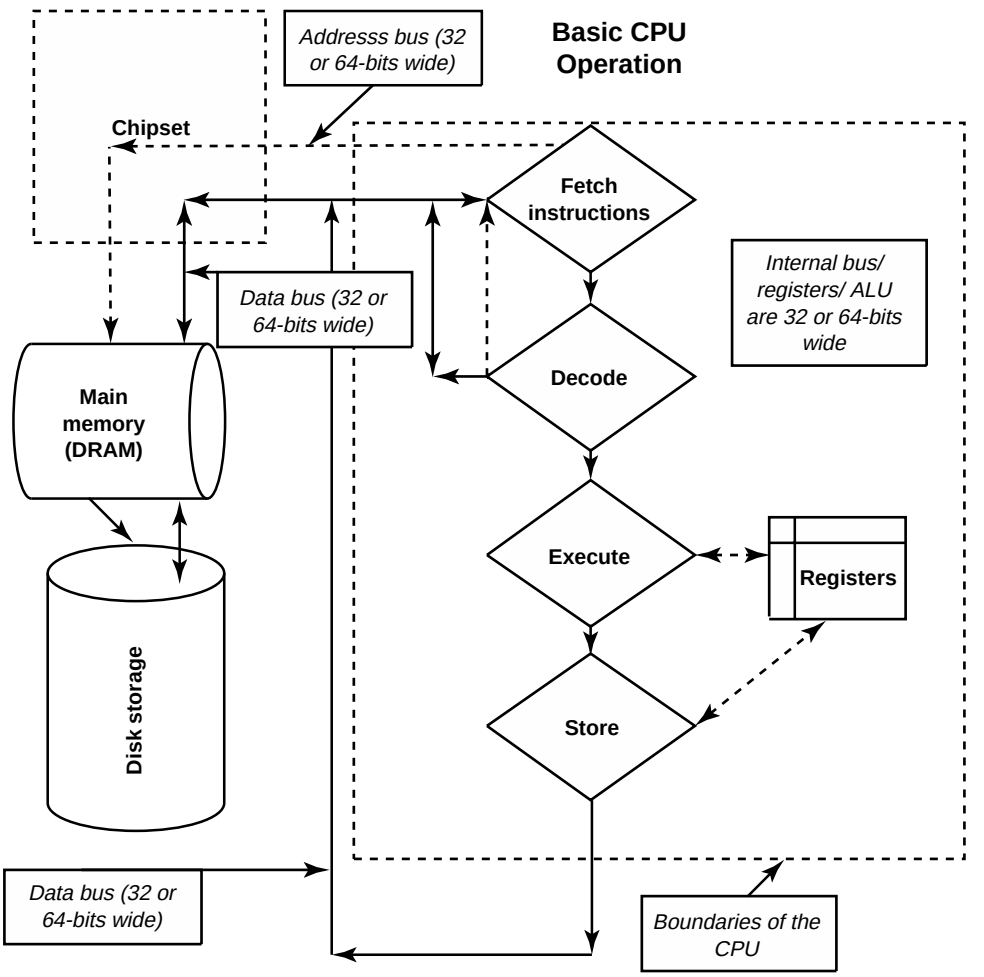


Figure 1.5 The flow of instructions in a CPU.
Note that this illustration shows bus widths.

of registers. With a large *register file*, you can use one load operation to bring an entire data series onto the CPU and do all your additions or whatever without having to access memory again. It's easy to see that adding registers is an important way of making a CPU more productive.

Tech Talk

Instruction set: The instruction set refers to both the instructions that a CPU can execute and the way in which they are organized. CPUs from different vendors have different instruction sets unless one is a *clone* of the other.

The Clock

Correctly executing instructions in a CPU depends on perfect synchronization. If instructions somehow got out of logical sequence, more of the results would be erroneous than accurate, and even worse, it would be impossible to tell which was right and which was wrong. Since even one error is intolerable, something in the system has to make sure that actions within the CPU are coherent. This management function is performed by the clock. As described earlier, the clock's circuitry is based on a quartz crystal system like that used in watches. At precisely timed intervals, the clock sends out pulses of electricity that cause bits to move from place to place within or between logic gates or between logic gates and registers. Simple instructions, such as add, can often be executed in just one clock cycle, while complex ones, such as divide, will require a number of smaller steps, each using one cycle. Chip speed is measured in cycles per second, which was once referred to by the acronym *cps*, but which is now known by the word Hertz (abbreviated Hz and named for the famous German scientist Gustav Hertz). So, a 700 million cycle per second CPU is described as functioning at 700 MHz.

Interrupts

If the CPU couldn't be interrupted until it had completed a task, its usefulness would be greatly reduced. In effect, the CPU would be like the primitive batch processing computers of the earliest days of computing; they could work on only one program at a time and were unable to do anything else until that program was completed.

To make interrupts possible, CPUs have lines (wires) that connect them to an external interrupt controller chip (part of the chipset), which contains a small database of what are known as *interrupt vectors*. When an interrupt signal comes onto the chip, the CPU saves what it is doing and goes to the interrupt vector (which is just a fancy name for a numerical table) to find the address of the instruction that the interrupt is telling it to execute instead. When finished, it goes back to the previous task. The CPU keeps track of where it was by writing the address of an interrupted task to a special register known as a *stack*. Interrupts have various priority levels that are interpreted differently according to the task the CPU is engaged in. For example, a busy CPU might temporarily ignore low priority interrupts, such as those coming from the keyboard, but would respond immediately to a high priority one that carried something like a disk error message. Similarly, a software program can be designed to *mask* lower priority interrupts if it isn't capable of being stopped.

Designing a Faster CPU

As computers become more powerful, defined as being able to do a greater amount of work in a given unit of time, they become more useful. How do you get more power? This section describes the most important strategies.

Faster Clock Speeds

It's obvious that a faster clock will allow more to be done in a given unit of time. The original IBM PC's clock ticked just under 5 million times a second (actually, 4.77 MHz). As this is written (1999), standard PCs are beginning to exceed 700 million clock cycles per second. This hundredfold-plus improvement has taken just over 15 years. And the beat goes on. Expect speeds of desktop machines to double—to a billion and a half cycles per second (1.5 GHz)—by 2001.

A chip's maximum clock speed, the fastest it can go without producing errors, is a function of how it is made. Specifically, higher speeds require that distances within the CPU have to be shorter. Thus, for the hundredfold improvement we have seen so far, the connecting lines between transistors in logic gates have to be shorter (thinner) and closer together, and these in turn have to be closer to other logic gates, registers, and other circuits. Faster chips necessarily have greater density; that is to say they have more transistors in a given area. Figure 1.6 illustrates one type of transistor.

Tech Talk

Clock speed: CPUs (and other devices) are controlled by quartz crystal clocks. The consistent timing provided by the clocks helps to keep operations synchronized. Clock speed is usually measured in millions of cycles per second; abbreviated MHz.

There are several reasons why shorter distances are essential to faster chips. One is that the speed of electricity in a wire is constant. Our speedier clock is not actually making bits move faster, it is just making them move more frequently. Since electrons won't go faster, a higher speed circuit must assume shorter distances for each clock cycle. Distance is an issue even within transistors. If the parts of a transistor are closer together, it can cycle faster. A bonus is that chips with shorter internal paths also require less power.

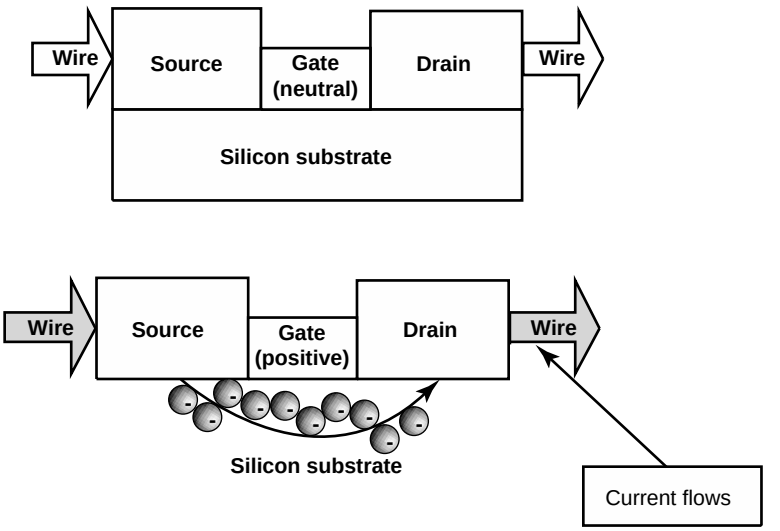


Figure 1.6 One type of transistor.
In the top drawing, the gate is neutral and there is no movement of electrical current. If a positive charge is applied to the gate, as in the lower drawing, electrons are attracted and allow current to flow from the source to the drain. If wires are attached to the source and drain, the transistor is a switch. If current flows, it is a 1; if not, it is a 0. The narrower the gate, the faster the chip can cycle, or turn a circuit on and off. The transistor is naturally a binary device. There isn't much you can do with one transistor, but put them together in groups, logic gates, and you have something really powerful.

TIMELESS CHIPS

Clock-driven CPUs work well, but this synchronous approach is energy inefficient. Every time the clock ticks, electricity pulses through the chip, whether any computation is going on or not. Laptops and other devices that need to save power do so in part by slowing or even shutting down the CPU when not in use. The penalty in this method, of course, is that the CPU is slow to reawaken when needed. Asynchronous (non clock-driven) chips exist, and have been shown to have much lower power consumption, but the required control circuitry uses lots of transistors that could be employed for computation. However, as shrinking parts allow more transistors on a chip, expect asynchronous designs to be used for special purposes where both low power and fast reaction are essential.

Making distances shorter in a CPU (or any other chip) means that the fundamental features of the circuits need to be smaller. Among the most important of these elements are the *traces,* microscopic equivalents of wires that make the paths between logic gates. Feature size, in the form of trace widths and transistor design rules, has gone from about 0.5 microns (a micron is one millionth of a meter or about one twenty-five thousandth of an inch) in the original PC to about 0.18 microns in the 700 million-plus cycles per second screamer of today; 0.15 design rules should be standard and 0.13 in testing by the time you read this. These successive stages of miniaturization are the result of some incredible improvements in process engineering. You would expect to be told next that these advances don't come cheap. Well they don't; the cost of chip factories, or *fabs,* goes up by huge amounts with each new generation. New ones today are in the several billion dollar range. Amazingly though, each of these fabs is radically more productive. This means that once processes are perfected, smaller, faster CPUs are also cheaper than their bigger, slower predecessors. This is the reason for "Moore's Law," which states that the number of transistors on a chip doubles every eighteen months, while the cost falls by about fifty percent. By the way, since there was no Bureau of Computer Laws where Moore could register his thought, there are a lot of variations of his law floating around.

Tech Talk

Trace: The wires that connect devices within a chip are so tiny that they are called *traces.*

Tech Talk

Moore's Law: Intel pioneer Gordon Moore stated that the number of transistors on a chip would double every eighteen months, and that their cost would fall by fifty percent during the same time.

Table 1.3 provides an example of the impact of smaller design rules on a chip's size and speed. A CPU made by IBM and Motorola, the Power PC604e, was unusual because it kept the same design through three process generations. Fortunately, this anomaly provides an excellent illustration. You'll observe that as its internal elements got smaller, the chip shrank in size while its speed increased. Most important, the smaller and faster version was also less power-hungry and a lot cheaper to manufacture than its bigger, slower, energy-gulping predecessors. The manufacturing dimension is discussed in Chapter 4, but note for the moment that it's likely that the 47mm^2 size yielded four times as many good chips from one wafer as the 148mm^2 version.

Wider Paths

In addition to a faster clock, a way of speeding up the CPU is to have it process more bits on every cycle. The first generation of microcomputers moved 8 bits at a time, the first IBM PC was 16 bits wide, the current generation of microprocessors is 32

Table 1.3 *The Power PC 604e, which stayed with the same 5.1 million transistor design over three process generations, provides an excellent example of the impact of feature size on CPU area and speed.*

Impact of Feature Size on Chip Dimensions and Clock Speed			
Year	**Feature size in microns**	**Size in mm^2**	**Speed in MHz**
1996	0.5	148	225
1997	0.35	96	233
1997	0.25	47	375

bits, and the next (standard by about 2002) will be 64 bits. The number of bits is described as a function of *width* because down there inside the chip, that's what it means. Compared to its 16-bit predecessor, a 32-bit computer has twice as many traces connecting logic gates, the logic gates themselves are about twice as wide, and so on. You will have observed right away that wider is the enemy of denser. That's true. A 32-bit chip will be bigger than a 16-bit one with the same number and size of logic gates. A 32-bit register will be bigger than a 16-bit register, etc. Thus, part of the benefit of smaller feature sizes has been employed not just to make chips smaller, but also to give them wider circuits. The tradeoff is well worth it, though. For a variety of reasons, including that the wider chips can use the same control structures to manage twice as much data, they are *much* more efficient. Instructions can be more complex; more data can be processed at once, and so on.

Tech Talk

Word: A computer's *word* length is the amount of information that it can process at one time. Current desktop computers use 32-bit words; 64-bit systems will be commonplace soon.

Doing More than One Thing at a Time

Higher clock speeds aren't the only way to increase productivity within a CPU. An obvious additional strategy is to do more than one task during each clock cycle. A popular technique of this type is called *pipelining.* In a pipelined CPU, while one part of the chip is fetching a new instruction, another is analyzing, yet another is executing, and another is storing results. Complex CPUs can have lots of pipeline stages. Intel's sixth generation CPUs (Pentium Pro, Pentium III), have 14 stages in their pipelines vs. only five for the Pentium. Modern CPUs also fetch several instructions at once, placing them in the first stage of the long pipeline, the *prefetch queue.* This

helps to ensure that the pipeline is always full. Today's processors, with their longer pipelines are, of course, described as *super*pipelined. Figure 1.7 illustrates piplining.

Tech Talk

Pipeline: Engineers describe the path that instructions follow through a CPU—*fetch, decode, execute, store,* and variations—as its *pipeline.* In a *pipelined* processor, every stage of the pipeline is doing work on every clock cycle.

A second technique, following logically from the first, is to have more than one pipeline. In this approach, instruction 1 in pipeline A is executed at about the same time as instruction 2 in pipeline B even though it followed 1 into the CPU. This is a lot more efficient as long as 2 doesn't depend on the results of A. Fortunately, programmers can often organize the flow of instructions in a way that avoids dependencies. CPUs with multiple pipelines are called *superscalar* (see Figure 1.8). Most contemporary microcomputer CPUs have this characteristic. Most also have different kinds of pipelines. The standard pipeline, one present in all CPUs, performs integer

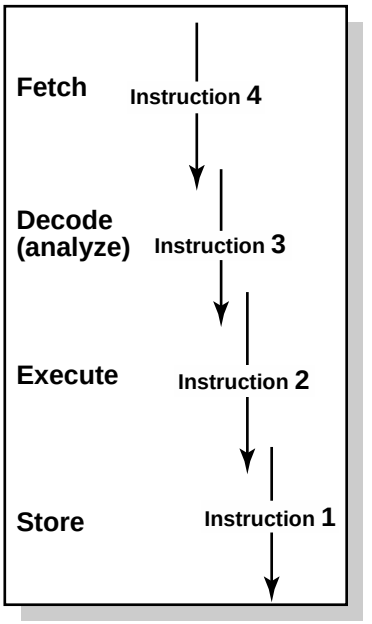


Figure 1.7 Pipelining.
A pipelined design can complete one instruction for each clock cycle as compared to one every four cycles for a pipe that handles just one instruction at a time.

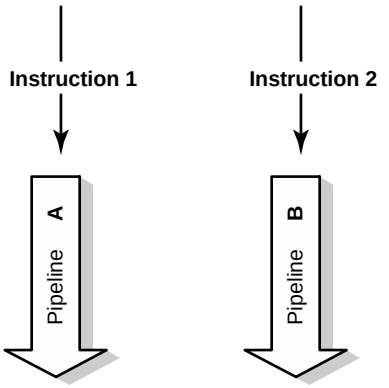


Figure 1.8 Superscalar.
In a superscalar design, Instruction 2 enters its pipeline one clock cycle after Instruction 1 and each flows down its corresponding pipeline in parallel. This is the simplest approach to parallel execution.

mathematics (whole numbers). Most CPUs also have pipelines that are optimized to do floating point math (using decimal points). Integer pipelines can be made to do floating point calculations, but for certain kinds of applications, such as scientific analysis, CPUs with one or more special floating point pipelines can be dramatically faster.

Tech Talk

Superscalar: A CPU that is *superscalar* has more than one pipeline.

More exotic ways of doing more than one thing at a time include such techniques as *branch prediction* and *speculative execution.* Branch prediction circuitry is designed to deal with the fact that typical computer programs have lots of decision points (branches).

To illustrate, a branch occurs when an instruction tells the CPU to perform a certain operation and, if the result meets the prescribed test, then to go on to the next instruction. As an example, the computer system in a portable pocket organizer might want to determine if the next appointment time has passed. To do this, it would subtract that appointment time from the current time and then compare the result to zero (i.e., the appointment time equals the current time). If the result is zero, the system would perform the *branch* instruction—in this case, sounding an alarm. But, if this test is not met (the result is not zero), the system would forget the branch instruction for now and go about its business.

Tech Talk

Branch: A point in a program where the CPU may have to switch to a different stream of instructions is called a *branch.* A conditional branch is where the stream chosen depends on the result of some computation.

Branches can waste a lot of clock cycles while the CPU dumps everything in the pipeline and finds the new instruction. Branch prediction circuits deal with this by looking ahead in the pipeline (before the ALU) and predicting the result of a branch. If the CPU thinks the branch will be taken (usually based on past performance) it will then go ahead and load the alternative instruction and possibly also new data. This can be done while the ALU is still working on earlier calculations, thereby greatly minimizing the penalty imposed if a branch has to be taken. Speculative execution means that the CPU will use the equivalent of a separate pipeline to follow a possible branch. If it turns out that the branch is taken, the CPU will be ready with the result. If not, it's pitched.

Tech Talk

Speculative execution: A CPU that has circuits designed for speculative execution will execute the instructions after a branch just in case the program needs to go that way.

Techniques like pipelining and superscalar organization, not to mention branch prediction and speculative execution, require that a CPU have lots of extra transistors. That's OK, since manufacturers have been using smaller and smaller design rules that allow more and more traces and transistors. The Intel 8088 that powered the first IBM PC had 29,000 transistors. The Pentium III of today, which is superscalar and superpipelined, supports branch prediction, speculative execution, and a bunch of other clever stuff, has around 7.5 million transistors. Some high-end CPUs, like the HP 8500, have in the vicinity of 140 million transistors (though much of that is in on-board memory, discussed in Chapter 2). Figure 1.9 provides an illustration of a pipelined, superscalar CPU.

ARCHITECTURE: SUITING THE CPU TO THE TASK

As important as things like superscalar techniques and pipelining are, they don't exhaust the opportunities to make CPUs go faster. This section covers three ideas: reorganizing the computer to do a limited number of tasks quickly (RISC); using an instruction that has one opcode working on multiple operands (SIMD); and using an instruction that can have multiple opcodes working on multiple operands (VLIW).

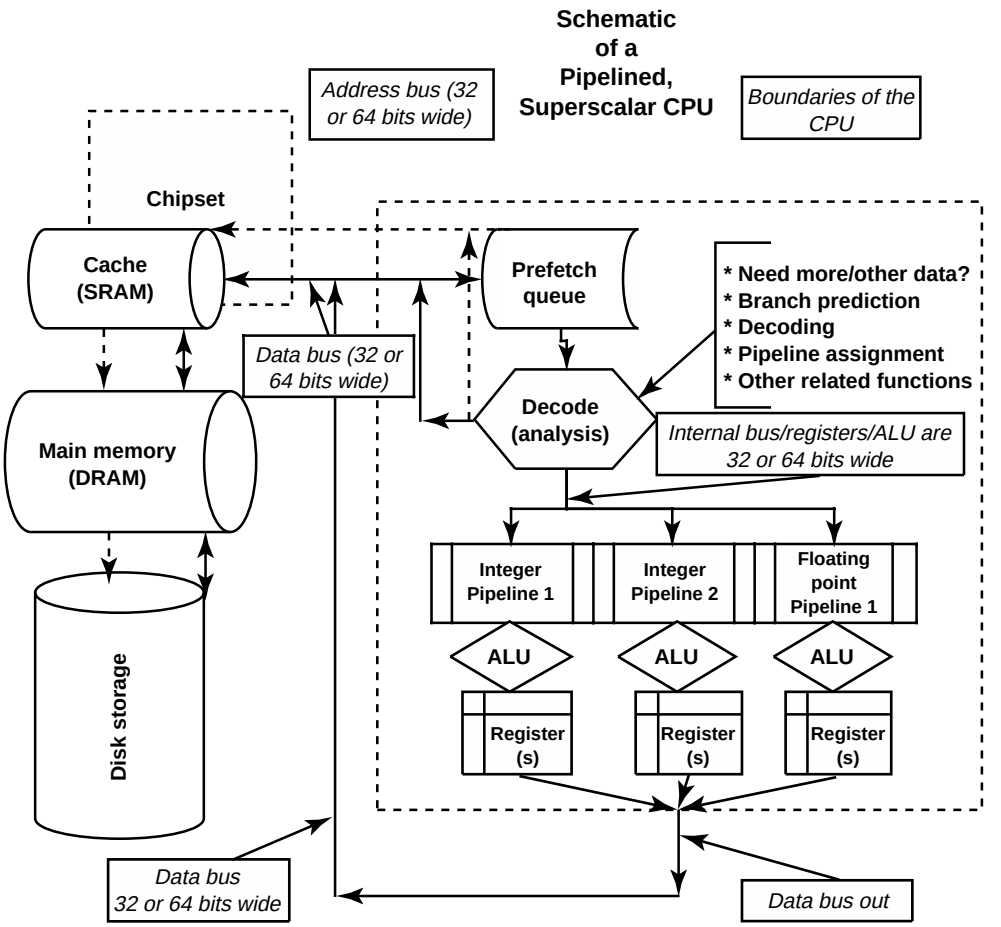


Figure 1.9 A pipelined, superscalar CPU.
This diagram shows the flow of operations within the CPU, as well as the most important external connections. Needless to say, this is highly simplified.

RISC

The best known CPU architectural issue is that of RISC vs. CISC. RISC, which means *reduced instruction set computing,* was developed in the 1970s by IBM researcher John Cocke, who noticed that computers typically didn't use many of the instructions, particularly the more complex ones, that were built into their circuitry. As an alternative, he designed a chip that used fewer and simpler instructions. When needed, the complex tasks could be accomplished by executing the simpler instructions in series (Cocke also made all instructions the same length, which further simplified circuits by comparison to CPUs that

had to analyze the instruction's length before beginning to execute it). The chip real estate that was no longer needed for the complex instructions was used instead for more registers.

Once new software was developed for this approach, Cocke was able to demonstrate that his chip could operate dramatically faster than the alternative style, known as a CISC, for *complex instruction set computing*, processor. IBM wasn't much interested at first, but the idea caught on because graduate students from Berkeley and Stanford who worked in Cocke's lab brought the ideas back to their professors. Once established in the academic-business-venture capital environment of the San Francisco Bay area, RISC technology quickly hit the mainstream. All modern microprocessors now use this approach to some extent.

Tech Talk

RISC: "RISC" stands for "reduced instruction set computer" and refers to a CPU that is structured to gain efficiency through circuits designed to execute relatively few instructions at high speed.

SIMD

A related type of internal optimization is called *single instruction multiple data* (SIMD) in which part of the CPU is designed to do multiplication, or some other math, on many numbers at once (see Figure 1.10). Intel has two versions of this. In the earliest, MMX (which stands for *multimedia extensions*), a 64-bit floating point pipeline can do the same calculation (e.g. multiply) on four 16-bit or eight 8-bit num-

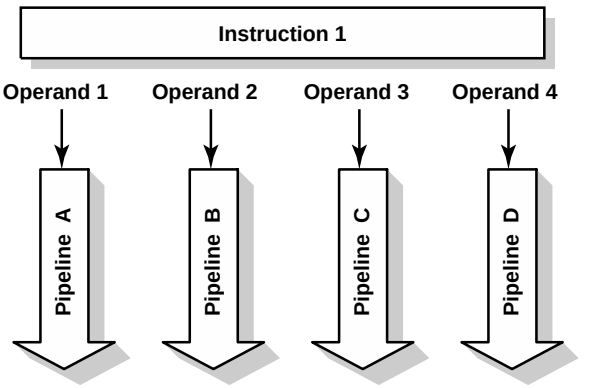


Figure 1.10 Single Instruction Multiple Data (SIMD).
In SIMD, one instruction can work on four 32-bit units of data at one time. Intel's versions often use chunks of data smaller than 32-bits, and one large pipeline rather than four separate ones.

bers simultaneously. The most recent Intel version of SIMD, SSE (which stands for *Streaming Single-Instruction, Multiple-Data [SIMD] Extensions*, uses two floating point pipelines, which are each 64 bits wide, to do the same thing, but on twice as much data at once. It should be obvious that the SIMD approach only works on data that is highly parallel. In fact, Intel added these instructions to deal with the manipulation of graphics, which are notorious for demanding that the CPU do one type of operation on long streams of numbers (see Chapter 3). A variant of SIMD, *vector computation*, is the way traditional supercomputers work. The principal difference is that vector computers can do more than handle a group of numbers at once; they can also carry out multiple parts of a calculation, for example, multiple stages of division, in one clock cycle.

VLIW

Another microprocessor design that is based on a technique requiring multiple simultaneous operations is known as *very long instruction word* (VLIW). See Figure 1.11. This is a more ambitious relative of SIMD. The idea here is that the processor will accept very wide (e.g., 128-bit) instructions. The CPU accommodates these long instructions with wide pipelines that can execute multiple opcodes on multiple operands at once. It's similar to a superscalar approach, except that the whole long instruction—typically the equal of four normal ones—is fetched and decoded at once rather than serially.

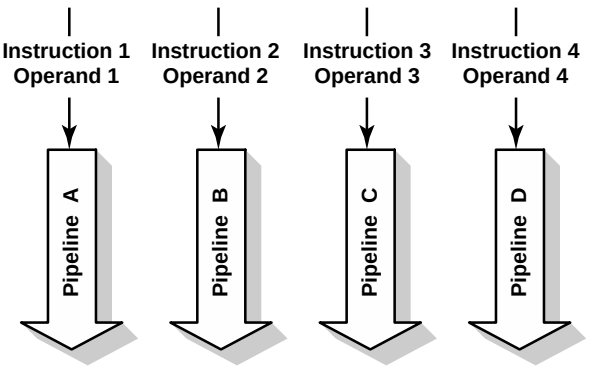


Figure 1.11 Very Long Instruction Word (VLIW).
In VLIW, the entering 128-bit word is divided and used to feed four 32-bit pipes. Unlike SIMD, there can be four separate instructions executing at once. Unlike a simple superscalar scheme, all pipelines are fed on the same clock cycle. The key difference with VLIW, however, is that instructions are organized for parallel execution, by the compiler, before they enter the CPU.

Tech Talk

VLIW: "Very Long Instruction Word" refers to a technique that is similar to SIMD, but supports the equivalent of multiple instructions in one word. In this manner, multiple instructions are feteched at the same time, alleviating the bottleneck of superscalar SIMD.

While this approach has obvious appeal, what makes it work in practice is that the software is organized for the architecture of the chip *before* it enters the CPU. In the ideal implementation, instructions and accompanying data fit neatly into the pipelines, executing smoothly and without dependency. But success in creating this software–hardware relationship also causes some problems for VLIW. The first is that (as with SIMD) not all software will benefit from VLIW. If a program doesn't have significant amounts of code that can be executed in parallel, it might even execute slower in a VLIW machine—it wouldn't do to have a bunch of mutually dependent actions lumped into one word. Another challenge is that the software that does the optimizing for parallel execution, called a *compiler* (see Chapter 8) is extremely difficult to write. A number of VLIW-based startup projects have failed because there wasn't time to create efficient software before the company ran out of money. A final serious problem of VLIW is that its software is intimately tied to the architecture of the CPU. Even minor changes in a chip will require new or substantially revised software (and it will still be hard to write).

Despite the potential drawbacks, the idea of VLIW is promising, and is central to the technology of Intel's seventh generation CPU "Itanium" architecture (also known as IA-64, for *Intel Architecture, 64-bit*). Intel's version of VLIW is called EPIC for *Explicitly Parallel Instruction Computing.*

Linking CPUs

You've probably observed that if a superscalar chip can do two or more operations in one tick of the clock, then two such chips could do twice as many. Three could do three times, and so on. *Multiprocessing,* as this approach is called, is indeed a widely used strategy in computing. Getting such systems to scale—meaning that two CPUs are really twice as fast, three are three times as fast, etc.—is not easy, though. Hardware has to be organized so that the communications channels between CPUs and between CPUs and memory don't get clogged. At a minimum, this is expensive. Finally, as with SIMD and VLIW, even the best multiprocessing hardware design won't scale if the software isn't optimized for it.

Making systems scale better is a challenging but potentially very important area of computing. Multiprocessing is discussed in some depth in the chapter on operating systems (Chapter 5).

HEY BUDDY, CAN YOU SPARE A PROCESSOR CYCLE?

Here's one way to make use of those CPU cycles that go unused while your computer just sits around. A group affiliated with the SETI Institute (Search for Extraterrestrial Intelligence) lets you use the Internet to download software that will analyze signals from a slice of the universe to see if it contains anything unexpected that might portend alien activity. When finished, your computer sends its work back and asks for more. This makes an otherwise impossible task manageable. Computer science students used a similar scheme to show that an important encryption code was breakable. We're likely to see lots more of this kind of multiprocessing in the future.

CPUs FOR SPECIAL PURPOSES

The term CPU normally means general purpose processor. There are a lot of other kinds of microprocessors that are optimized for specific kinds of instructions and data. We'll talk about three categories: application-specific integrated circuits (ASICs); digital signal processors (DSPs), and media processors. We'll also discuss a new technology, the *field programmable gate array* (FPGA), that is attracting a lot of attention.

ASICs

Many microprocessors are designed to do just a limited number of tasks. They lack the full instruction set of the general purpose CPU, which means they also have much simpler circuitry. The general term for these chips is ASIC—an *Application-Specific Integrated Circuit.* An example of an ASIC is a chip that receives information about vehicle operations and makes a decision as to whether the brakes should be applied, to which wheels, and how hard. An alternative to using an ASIC for this chore would be to have a general purpose CPU that would handle a bunch of tasks in addition to braking, for example, analyzing fuel flow and optimizing combustion to reduce pollution. This would likely be cheaper, but one advantage of an ASIC is that it is single-tasking. There's no competition for its attention. It can make that braking decision *right now*.

Increasingly, ASICs are built by connecting some special purpose circuits to a standard CPU *core* on the same piece of silicon. This strategy, which limits the cost of both design and fabrication, has powerful economic appeal. More on this and related approaches later. By the way, today's automobiles have both ASICs *and* general purpose CPUs.

PAPER OR PLASTIC?

There are a variety of ways in which CPUs and other chips are connected to the outside world. In the normal approach, a wire bonding machine creates pins that extend the input and output points of the chip. The combined device is then put in a plastic carrier for stability. Finally, the CPU is plugged into a socket in the computer. There are a variety of alternative approaches. All types of packaging are expensive, both for the process and for the testing that has to follow. It's easy to see, therefore, that making chips with fewer wires is a good way to lower cost. And yes, designers are talking about putting chips into paper devices.

DSPs

The fastest growing segment of microprocessors is that of the digital signal processor (DSP). DSPs generally work on data that begins life as analog—principally audio and/or video signals. DSPs make themselves useful by doing the many calculations necessary to, for example, remove interference from a video signal (television). These calculations are generally not complex, but the huge volume associated with such "streaming" data means that a general purpose CPU often would not be able to do them fast enough.

Think of DSPs as processors that can do a variety of tasks, but only for a certain class of applications. Their simpler structure can translate to a lot of speed, however. Texas Instruments, the premier producer of these chips, has them operating well into the GHz range. Since these processors have the ability to do multiple computations in one clock cycle (SIMD-style), their effective speed is in the billions of operations per second. DSPs are used in computer hard drives, where they take the analog signal from the read head and turn it into digital data. Similar applications, and the fastest growing, include audio devices such as cell phones and CD and DVD music players. The biggest coming market is in television set-top boxes, but for these the DSPs are likely to be integrated in multi-purpose chips.

Media Processors

A fast developing class of microprocessors is one that combines high-end DSP capabilities and some pipelines that have been optimized for graphics, together with relatively limited general-purpose integer functions. These chips, known as media processors, are aimed at the game machine and television set-top box markets. The thinking here, as with the development of RISC, is that chip real estate should be focused to the task at hand.

To give you the most realistic possible gore, your 3D game machine needs to process thirty or so complex graphical images (frames) in a second. Each of these images will have many objects, each of which will require a vast amount of computation. Originally, there were two ways to build these systems. One was to market a box whose silicon was dedicated to running games. Another was to add fast graphics processing ability to PCs. The newest alternative is to do the reverse—add some PC-type functions to a game-type chip. Of course, this kind of system won't run Word as fast as a Pentium III. However, in many cases, the slower speed won't matter since, when it comes to writing, the machine's principal users are typically still in the crayon stage. The first generation of chips to be described as *media processors* included some rather dramatic failures. As a result, the term has become somewhat unpopular. Thus, Sony and Toshiba, who created the first of a new class of super powerful media processors for the Sony Playstation, chose to call their chip the *emotion engine.*

TWO FOR ONE?

Microprocessor vendors, led by IBM, are planning to use the extra transistors made available by 0.18 micron technology to put multiple CPUs on a single chip. While a single large processor might seem to make more sense, there are a number of reasons for this new approach. Interestingly, one concern is that if all circuits were combined in one CPU on a single chip, distances could become so great that a clock pulse could not get all the way across the chip in one cycle.

FPGAs

An interesting new variation on processor design is the *field programmable gate array* (FPGA). These chips use special transistors (SRAM cells, see Chapter 2) at critical junction points in their structure. The junction points can be switched, effec-

tively altering the architecture of the chip. A relatively dumb kind of FPGA can only be programmed once, but more sophisticated kinds can be reoriented on the fly. As an example of what this technique can do, the military uses FPGA chips in target recognition computers. A chip might be programmed to look for twenty types of objects. After the system has scanned through the full set of possibilities and rejected some of them (e.g., there are no tanks present), the junction points are reset so that the entire capability of the processor is focused on analyzing the remaining options, for example, the more difficult task of finding artillery pieces. The problem with FPGAs is that the junction points use a lot of space, meaning that for the same computational capacity, they will always be quite a bit bigger than a normal CPU. Still, for some applications, FPGAs will be an increasingly popular choice. And the general concept appears to be at the heart of some of the most advanced processors now on the drawing boards (see Chapter 4).

NEW APPROACHES TO COMPUTING . . . . . .

There are a lot of radical ideas out there about how to make calculation much faster than it is today. We'll just mention a few.

Supercooling

Electrical circuits operate faster and more efficiently when they are cold—*very* cold, like 90° Kelvin (around –300° Fahrenheit). The biggest benefit of the CPU-in-the-deep-freeze is that it makes much higher clock speeds available. Unfortunately, there are a lot of hurdles to overcome before your desktop will feature a really cool computer. One is that the technology currently needed to supercool electronics is both expensive and bulky. If that problem is solved, there is the fact that both the CPU and the circuitry that it connects to will have to be redesigned to operate successfully in this temperature range. Still, these are not huge barriers. Supercooling is already being used in some high-end computers (supercomputer class), and a variety of companies are working to bring it to the midrange of mainframe competitors.

On a slightly different front, engineers are exploring the use of materials in the chip itself that have the property of carrying heat away. While this approach won't create dramatic changes in speed, the expected 100 percent or so improvement would be significant.

Optical Computing

As will be mentioned frequently in this book, electrical signals are vulnerable to interference from many sources. Reducing this vulnerability to zero is probably impossible; even getting close would require an enormous amount of bulky shielding. The whole concept of "micro" would disappear. Computers compensate for the probable interference by using complicated algorithms that analyze information sent from "A" to "B" in order to determine if it arrived correctly. This is true even within the CPU; it becomes a major exercise as data flows off the CPU to other parts of the computer.

An optical computer would be a great solution to this problem. Light in a vacuum is faster than electricity in a wire (~30 cm/sec. vacuum vs. ~20 cm./sec), but with current technology, this difference is of relatively little value. More important for the moment is that it is much easier to guard against interference in optics, which means that data carried by light can be transmitted with less error checking. The throughput (the real vs. the theoretical speed) of optical switches is therefore much greater than with data carried by electrons. Research has demonstrated that internal connections in a computer, say from the CPU to memory or to disk, could be accomplished much faster by light than by electricity. And it is feasible to build such a connection. So why isn't it done?

The first reason is that there isn't yet a compelling need for such speed in mainstream commercial machines. Existing computers are not yet hobbled by the problem of moving data on their very short, relatively error free, internal paths. This will likely change once the annual sixty percent or so increase in CPU speeds starts to go away—perhaps around 2015. The more important reason why optical components are not yet used inside production systems is that the increases they would bring are, at the moment, largely theoretical. The problem is that only an all-optical device will be significantly faster—translating from electronic data to optical data is a serious bottleneck. The electronics that do this are complicated and not especially fast.

So why not make the whole thing, including the CPU, optical? That's a powerfully attractive idea, and lots of scientists are working on it. Optical computers exist in the lab, but making a purely optical system, one that is able to avoid all the problems of electricity, is not yet possible on a commercial scale. Indeed, researchers appear to be quite some distance from achieving that. They need, as they say, a few "breakthroughs." Of course, breakthroughs are things that may or may not occur. Experience says that they will in this case, but when—two years, ten years, twenty years—is anybody's guess.

In the meantime, computer companies are getting lots of experience with optics. Because interference, and therefore reduced throughput, has been a real problem in computer networking, optical interconnects (fiber optics) are in wide use there and are rapidly becoming ubiquitous.

Even More Exotic Stuff

The electronic logic gate has a long way to go—to around 2010–2015—before it begins to lose momentum as the center of computation. Even so, scientists are actively exploring other, radically different techniques. One approach is to directly manipulate atoms so that the basic unit of measure becomes individual electrons (quantum computing). Expect to hear a lot about *carbon nanontubes* in the near future. Another concept is to use biochemical reactions, as in combining DNA, to execute enormously complex calculations at speeds that electronic systems could not hope to duplicate. Indeed, scientists are exploring pretty much everything that mirrors the ability of a transistor to act as a binary switch. As interesting as these ideas are, they aren't likely to have much of an impact in the near future—most definitely including the useful lifetime of this edition.

CONCLUSION . . . . . .

This chapter began with some basics of computation: computers work with instructions (opcodes) and data (operands). As these are fed into the CPU, the CPU goes through a standard series of operations—fetch, decode, execute, store. However, although these basics apply to all CPUs, there is considerable diversity after that. If the CPU is thought of in terms of the standard desktop system, we can observe that there have been considerable changes in architecture over a fairly short period of time.

The original complex instruction set (CISC) design has given way to one that uses, at least to some extent, a reduced instruction set (RISC) approach. Other organizational changes include the addition of multiple pipelines (superscalar) and specialized pipelines for such things as general floating point mathematics and for streaming media. Designers are also finding ways to optimize parallel execution with approaches such as having one instruction work on multiple pieces of data at once (SIMD) and including multiple instructions in one word (VLIW). At the same time that there is an increasing variety of approaches to the architecture of the standard desktop computer's CPU, there are more radical concepts for different kinds of devices. Application-specific integrated circuits (ASICs) can be added to a standard CPU "core" to make it more efficient at a limited range of tasks, while digital signal processors (DSPs) are being used to handle the increased need to work with information, such as video or audio signals, that are inherently analog rather than digital. Finally, the media processor presents a hybrid that is optimized for games and other entertainment-related computing. Given that we have seen so much change in only a quarter century or so, and given that designers have a vast increase in the number of transistors they can play with, it's reasonable to expect that we are just at the begin-

ning of thinking about how to design CPUs. In the future, a knowledgeable person is no more likely to say that a particular computer has "a CPU" than she would be to say the she drives "a vehicle."

We'll talk at some length about directions in CPU design and manufacture in Chapter 4. In the meantime, let's remember that the CPU can't do its work alone and that its effective speed depends on what happens in other parts of the computer. In the next chapter, we'll cover where instructions and data come from (memory and storage) and how they are moved into and out of the system (I/O). Next, Chapter 3 will discuss the increasingly important question of displays—how the computer manages to take streams of data and convert them into complex graphics that are visible on a screen or a printed page.

TEST YOUR UNDERSTANDING . . . . . .

1. What are the four stages of CPU operation?
2. What does an interrupt do?
3. What is the *width* of CPUs for contemporary desktop computers?
4. Why have clock speeds increased so rapidly since the microprocessor was invented?
5. Why is pipelining important?
6. What is distinctive about a superscalar processor?
7. What is the difference between fetch/execute and load/store?
8. What is the difference between RISC and CISC?
9. What is the key reason for Moore's Law?
10. What is the difference between VLIW and SIMD?
11. What is the principal advantage of optical computing? What is the principal challenge to realizing it?
12. What are examples of special purpose CPUs?