

● Machines

When someone says, “My machine is down,” there is no doubt what they mean. They are not saying that their car won’t start, or the toaster won’t toast, or the air conditioner is emitting sparks. No, it’s the computer that is not working.

This usage dates back to the time when ‘computers’ were people who did hand calculations and electronic computing machines were a novelty. Nowadays, ‘computer’ has largely lost its old meaning, but ‘machine’ in the sense of a digital computer thrives. It is enshrined in the name of the Association for Computing Machinery.

The topic of this part of this book is the machine—or more properly the hardware side of the hardware/software divide. For many people this distinction does not exist, or exists only in the vague feeling that if they could afford more memory (whatever *that* is) their applications would run faster. But in scientific computing you ignore at your own risk the fact that your programs run on machines.

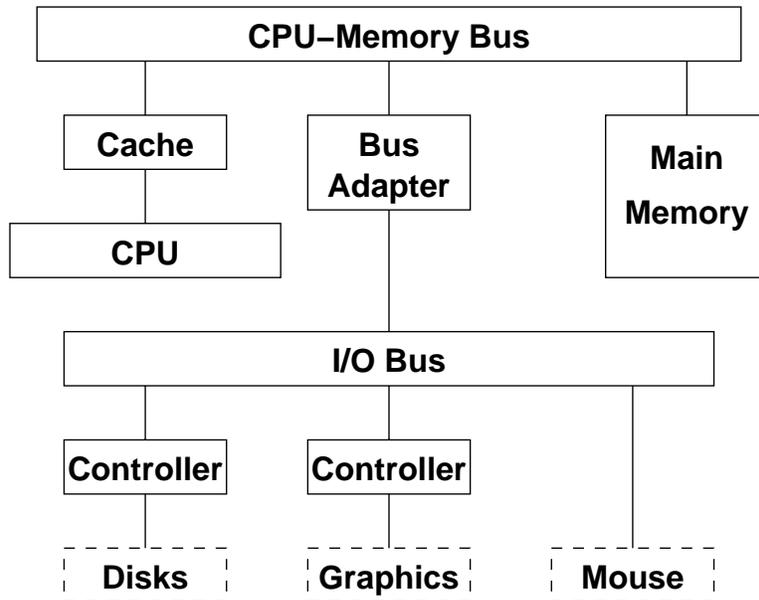


Figure 1.1: A simple computer

1. ANATOMY OF A COMPUTER

Like any machine, a computer consists of a number of interacting parts. Figure 1.1 shows the parts of a simple computer and how they are connected. In this section we are going to describe these parts, just as a medical anatomy describes the organs of the body. In addition we will touch briefly their physiology—how they work—although there will be much more of that in the following sections.

At the end of this section are two digressions. The first is a brief treatment of processes and the operating systems that coordinate them. The second is a discussion of the law of diminishing returns in the form of Amdahl’s law. They are placed here, somewhat awkwardly, because they will be needed throughout this book.

1.1. THE CENTRAL PROCESSING UNIT (CPU)

SYNOPSIS: The basic function of the CPU is to manipulate information, which is encoded as sequences of bits—bytes and words. These are held in registers where they are manipulated by CPU instructions. The data in registers has meaning only in the context of the instructions that operate on them.

The operation of the CPU is synchronized by a clock, whose speed determines how fast the

computer can execute instructions. In real life, however, the performance of a computer also depends on other factors than clock speed.

The central processing unit is the heart and mind of the machine. Its basic function is to manipulate information. This information consists of **bits**. A bit is a single binary unit capable of assuming numerical values of 0 or 1. Because a bit cannot contain much information it is clumped together with other bits—most commonly into **bytes** of 8 bits or **words** of 32 or 64 bits. It is easy to see that there are $2^8 = 256$ distinct bytes; that is, a byte can represent 256 units of information. A 32-bit word can represent $2^{32} = 4,294,967,296 \cong 4.3 \cdot 10^9$ units of information.

Information is stored in **registers** and in main memory. The CPU processes this information by executing sequences of **instructions** that change and combine the contents of its register and of memory. It is important to keep in mind that the data in registers have meaning only in relation to the operations performed on or with them. For example, the same configuration of bits in a register may represent a string of alphanumeric characters or a binary number. In the former case, a program may issue instructions to check if a character is a decimal digit or a letter or something else. In the latter case, the program may add, subtract, multiply, or divide the data in its registers.

The operation of a CPU is rigidly synchronized by a **clock**. The time between consecutive ticks of the clock is called a **cycle**. The speed of the clock determines how fast the computer can execute instructions. Other things being equal, the faster the clock the faster the computer, which is why clock rates feature so prominently in advertising for computers. But, as we shall see, things are frequently not equal, and clock rates are only one factor in determining the performance of a computer.

1.2. THE MEMORY SYSTEM

SYNOPSIS: Main memory lies outside of the CPU and typically consists of a sequence of bytes that have unique addresses. The totality of possible addresses is called the address space, and it is usually larger than the size of the memory. Main memory contains not only data but also program instructions.

The rate at which main memory can be read or written is considerably slower than the CPU cycle. Consequently, between the CPU and main memory is a buffer, called a cache, of high speed memory. Blocks of data are transferred from main memory to the cache as needed. There they can be read and written at high speed.

When data and instructions overflow main memory, they must be written out to a slower storage device such as a disk. By treating main memory as a cache for the disk, a virtual memory system is able to manage the transfer of data between main memory and the disk invisibly to the user.

Although registers are the working memory of the CPU, there are not enough of them

000	xxxxxxxx
001	xxxxxxxx
010	xxxxxxxx
011	xxxxxxxx
100	xxxxxxxx
101	xxxxxxxx
110	xxxxxxxx
111	xxxxxxxx

Figure 1.2: A small memory

to store the massive amounts of data the computer must manipulate. Consequently, the computer has a capacious **main memory** to hold the bits it must manipulate. Typically, but not always, these bits are organized into bytes of eight bits. The bytes are arranged linearly in memory, each byte having a unique **address**. Figure 1.2 shows a small memory consisting of eight bytes. The addresses of the bytes on the right are written in binary. A memory with addresses containing n bits can have up to $2^n - 1$ bytes. The range of addresses from 0 to $2^n - 1$ is called the **address space** of the memory.

Typically, the address space is larger than the physical memory. For example, a 32-bit address is equivalent to about 4 GB (gigabytes) of memory — considerably more than is found on your garden-variety PC. However, a recurring mistake in the design of computers has been to make the address space too small. A few years ago, 4 GB of memory would have been inconceivable. Yet today there are computers with more memory than that. And tomorrow, memories will be even larger.

On general purpose computers the main memory not only contains the data the computer processes but the instructions that it executes. Such a machine is called a **stored program machine**, since its programs are stored in main memory. It is also called a **von Neumann machine** after John von Neumann, who first proposed the idea. As we shall see in a moment the presence of instructions in main memory is a potential roadblock to their fast execution.

Memories and CPU's do not march to the same drummer. Generally the time to read or write a single memory address—the memory cycle—is considerably greater than the CPU cycle. Since the execution of an instruction requires that it be fetched from memory, it would seem then that the CPU cannot execute instructions faster than the main memory cycle—a serious problem. A related problem is that of feeding data to the CPU at a sufficiently fast rate.

One way of circumventing this problem is to install faster memory. Unfortunately, the cost of memory grows with its speed, so that superfast memories are found only on

expensive supercomputers.

A second solution is to observe that a significant part of the memory cycle is devoted to setting up the memory for a transaction. For example, it takes time to prepare the memory to move information to or from a specific memory address. But once the memory is set up, contiguous blocks of data can be transmitted at a rate faster than the memory cycle. This suggests that the CPU be provided with a small (and therefore affordable) **cache** of fast memory which buffers blocks from main memory. The blocks of data can be transmitted at a rate faster than the main memory cycle, and once they are in cache, they can be read by the CPU at the rate of cache memory.

Cache memories are implemented in hardware so that their existence is invisible to the programmer. Nonetheless, it is possible to write code that forces unnecessary reads and writes of blocks. The only way to avoid these inefficiencies is to understand the workings of cache memory.

When the data, including instructions, for a program exceeds the capacity of main memory, it must be stored on a slow secondary storage device—usually a disk. **Virtual memory** is a system for managing the transfer of data between disk and main memory so that it is invisible to the user. It does this by making main memory a cache for the virtual memory residing on the disk. Note that the virtual memory system does not have a place in Figure 1.1. That is because the system is scattered through the components of the machine. In addition, the virtual memory system has a substantial software component, unlike cache memory, which is implemented in hardware.

1.3. BUSES AND PERIPHERALS

SYNOPSIS: Buses convey information between parts of the machine. Since several devices can be connected to a bus, busses must have hardware protocols for deciding which device gets use the bus and for two devices to communicate.

Peripheral devices, like disks, mice, and monitors, lie at the edge of the machine and serve to communicate with the outside world. Peripherals are generally connected to an I/O bus, which is in turn connected to the memory bus via a special piece of hardware.

Buses are the circulatory system of a computer, moving bits between its parts. Conceptually a bus consists of lines along which information can be transferred. Each line corresponds to one bit. Thus to transmit a 32 bit word in one step, the bus must have 32 data lines. The number of lines in a bus is called the **width** of the bus. The **length** of the bus is the physical length of its lines. The rate at which information can be transferred over the bus is the **bandwidth** or **throughput** of the bus. Peak bandwidths (the highest possible bandwidth) typically vary from 100 to over 1000 MB/sec (megabytes per sec).

Buses are not just passive conduits for information. Several devices can be connected to a bus. Hence the bus must coordinate who gets to use the bus. The bus must also provide a protocol by which two devices can talk to each other, and each device must have hardware to use this protocol. An example of such hardware is the **bus adapter** or **bridge** in Figure 1.1, which allows the I/O bus to use the memory bus.

Peripheral devices are the way the computer communicates with the outside world. A mouse, a printer, a disk are all peripherals. We are going to treat peripherals in greater detail later. However, it is worth noting in advance that in our simple computer, the peripherals are connected to a I/O bus, which in turn is connected to the memory bus. This means that a peripheral can communicate directly with memory, saving the CPU the job of managing much of the computer I/O. Complicated, high-speed peripherals like a disk generally have a **controller** — a small computer in its own right that manages I/O transactions.

1.4. PROCESSES AND THE OPERATING SYSTEM

SYNOPSIS: A process is a program in action on the machine. More than one process can run on a machine, but only one at a time. One of the jobs of the operating system is to coordinate concurrent processes, stopping and starting them as required. To do this the operating system runs in a privileged mode that allows it to do things ordinary processes cannot do.

Interrupts are the way of directing the operating system's attention to conditions that require immediate attention. When an interrupt occurs, the current process is suspended and control turned over to an interrupt handler that does whatever is necessary.

Hardware cannot be divorced from software. Although these initial sections are devoted primarily to the architecture of a computer, we will often have to make reference to programs running on the machine. Hence we need to introduce some terminology.

The basic unit of computation is a **process**. You may think of a process as a program, but it is more than that. It is a program in execution. That is not to say that the execution is continuous. There can be several processes in existence — somewhere between initiation and termination — but only one can use the CPU at a time. Processes that exist at the same time are called **concurrent processes**.

Concurrent processes must be coordinated. That is the job (actually only one of the jobs) of the **operating system**. For example, when a process makes a request for input, it generally cannot proceed until the input data arrives. Rather than tie up the CPU, the operating system (which also handles I/O requests) suspends the execution of the requesting process, and starts another process. In due course, after the data arrives, the operating system reactivates the original process. This swapping of processes is called a **context shift**.

It is tempting to regard the operating system as an entity that stands above the computer directing its operations. In fact, the operating system runs on the CPU as process in its own right. Consequently, the CPU runs in two modes. User programs run in **user mode**; the operating system runs in a privileged **executive** or **system mode**. To get from user mode to executive mode the machine uses a system of **interrupts**. For example, when an I/O device needs the attention of the operating system, it sends a signal that causes the CPU to stop what it is doing and transfer control to a program called an **interrupt handler**, which takes appropriate action on the condition that caused the interrupt. When an interrupt occurs the status of the machine must be saved so that the CPU can return to business after the interrupt handler has finished. Moreover, some interrupts are more important than others and their handlers must be given priority. How these problems are resolved by a combination of hardware and software is highly system dependent and will not concern us here. Fortunately, a knowledge of these details is rarely needed in scientific computing.

1.5. AMDAHL'S LAW

SYNOPSIS: When a task can be divided into two independent subtasks, speeding up one of the subtasks will speed up the original task. Amdahl's law is a formula that shows the limits of speeding up a subtask and suggests which one to work on. In its form it also embodies the law of diminishing returns.

—

Here is a problem that arises time and again in computer science—not to mention elsewhere. Suppose we have a task that takes time T to complete. Suppose further that this task can be subdivided into two tasks A and B that require time T_A and T_B , so that the total time is

$$T = T_A + T_B.$$

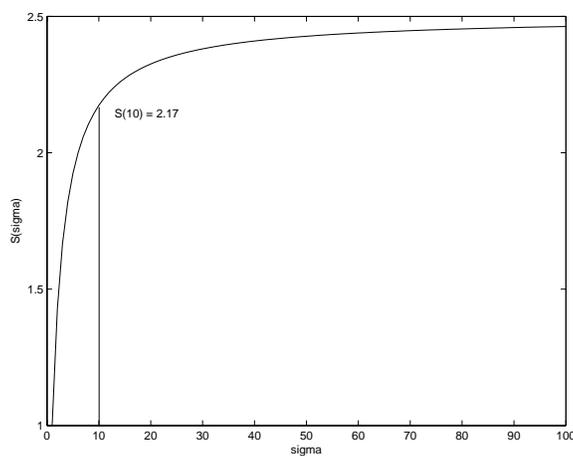
How much do we gain by altering task B to reduce the time T_B ? A classic example of this problem is when T represents the time to execute a program, T_A is the CPU time spent actually spent executing the program, and T_B is the time when the CPU is idle waiting for input.

To make the notion of 'gain' precise, let T_B be reduced to T_B/σ , where $\sigma > 1$. Then the new total time is

$$T_\sigma = T_A + \sigma^{-1}T_B.$$

The ratio

$$S(\sigma) = \frac{T}{T_\sigma} = \frac{T_A + T_B}{T_A + \sigma^{-1}T_B}. \quad (1.1)$$

Figure 1.3: Amdahl's law for $f_B = 0.6$.

is called the **speedup**. If it is equal to ten, then the new combined task runs in one-tenth the time of the original; that is, it runs ten times faster. Since $\sigma > 1$, the speedup is always greater than one. Note that σ itself can be regarded as the speedup in passing from doing task B in time T_B to doing it in time T_B/σ .

To see how large the speedup can become, let us introduce the quantities

$$f_A = \frac{T_A}{T} \quad \text{and} \quad f_B = \frac{T_B}{T}.$$

Thus f_A is the fraction of the total time accounted for by task A, and similarly for f_B . These quantities are not independent but satisfy the relation

$$f_A + f_B = 1.$$

Now divide the numerator and denominator of (1.1) by T to get

$$S(\sigma) = \frac{1}{f_A + \sigma^{-1}f_B}. \tag{1.2}$$

Then as $\sigma \rightarrow \infty$, the speedup $S(\sigma)$ approaches

$$S(\infty) = \frac{1}{f_A}$$

The graph in Figure 1.3 illustrates the approach of $S(\sigma)$ to its asymptotic value of 2.5 for $f_B = 0.6$.

Equation (1.2) is called **Amdahl's law**.¹ It puts a limit on how much you can speed up a task by speeding up one of its components. Moreover, it tells you which of two components to select for speedup: namely, the one with the largest fraction f . For if $f_B < f_A$, then speeding up task A gives an asymptotic speedup of $1/f_B$, which is greater than $1/f_A$, the result of speeding up task B.

All this assumes that the costs of speeding up tasks A and B are the same. If it costs more to speed up task A, then it may pay to work on task B even though f_B is less than f_A .

The graph in Figure 1.3, which is a hyperbola, quickly gets in the neighborhood of its horizontal asymptote of 2.5 rather quickly but then stagnates. Speeding up B by 10 gives a total speed up of about 2.17, which is 87% of 2.5. Increasing the speedup to twenty gives a percentage 94% — an increase of only 7%. In other words, the first ten units of speedup in B gives you 87%, while the next ten units gives you only 7% more. Since those next ten units are bound to cost at least as much as the first ten units, the extra speedup is probably not worth the effort. This is known as the **law of diminishing returns**.

This behavior of **Amdahl's hyperbola** is as important as the value of $S(\infty)$. The approach of a hyperbola to its asymptote is initially quick but soon tapers off. Thus while $S(\infty)$ tells us improvement is limited, the hyperbolic behavior warns us not to try too hard for it.

In summary, Amdahl's law is useful in determining the maximum benefits of speeding up a task, deciding which task to speed up, and deciding when to stop. As we said above you have to factor in the costs of these speedups. But used intelligently, Amdahl's law can be an important design tool.

EXERCISES

1. In scientific notation prefixes kilo, mega, giga, and tera stand for 10^3 , 10^6 , 10^9 , and 10^{12} . Because 1000 is well approximated by $1024 = 2^{10}$ to calling a memory of size 1024 a kilobyte memory. This convention has been extended to the other prefixes, so a 3 GB memory contains 3×2^{30} bytes. For a kilobyte this approximation has a relative error of

$$\frac{2^{10} - 10^3}{10^3} = 0.024$$

or about 2.4 percent. Compute the percent errors for the other prefixes. What do you conclude?

2. Suppose that main memory can deliver bytes to the CPU at 500 MB/sec and an instruction consists of 4 bytes. Assuming that the CPU has a 1 ns cycle, how many cycles does it require

¹Actually, Amdahl proposed his law to analyze parallel computation. In his formulation task B is the part of a computation that can be parallelized while task A is the part that must be performed serially. Task B can be speeded up by adding processors, but the total speed up is limited by the serial task A.

to fetch an instruction.

3. In this exercise we use the notation of Amdahl's law. Suppose you can speed up task A by σ_A and task B by σ_B , but only one of them. Let $\rho = f_B/fa$. Show that if

$$\rho < \frac{\sigma_A \sigma_B - \sigma_B}{\sigma_A \sigma_B - \sigma_A} \equiv \rho_{\text{crit}}.$$

then you should speed up A. If $\rho > \rho_{\text{crit}}$, then you should speed up B. If $\rho = \rho_{\text{crit}}$, it does not matter which you speed up.

