Chapter 1: Introduction and Overview of Hardware Architecture

Learning Objectives

When you've finished this lesson, you will be able to

- Describe the evolution of computing devices and the way most computer-based devices are organized.
- Make simple conversions between the binary, octal and hexadecimal number systems, and explain the importance of these systems to computing devices.
- Demonstrate the way that the atomic elements of computer hardware, logic gates, are used and detail the rules that govern their operation;

Introduction

Today we often take for granted the impressive array of computing machinery that surrounds us and helps us manage our daily lives. Because you are studying computer architecture and digital hardware you no doubt have a good understanding of these machines, and you've probably written countless programs on your PC's and workstations. However, it is very easy to become jaded and forget the evolution of the technology that has led us to the point where every Nintendo Gameboy® has 100 times the computing power of the computer systems on the first Mercury space missions.

A Brief History of Computing

Computing machines have been around for a long time, hundreds of years. The Chinese abacus, the calculators with gears and wheels and the first analog computers are all examples of computing machinery, in some cases quite complex, that predates the introduction of digital computing systems. The computing machines that we're interested in came about in the 1940s because World War II artillery needed a more accurate way to calculate the trajectories of the shells fired from battleships.

Today, the primary reason that computers have become so pervasive is the advances made in integrated circuit manufacturing technology. What was once primarily orange groves in California, north of San Jose and south of Palo Alto, is today the region know as Silicon Valley. Silicon Valley is the home to many of the companies that are the locomotives of this technology. Intel, AMD, Cypress, Cirrus Logic and so on are household names (if you live in a geek-speak household) anywhere in the world.

About 30 years ago, Gordon Moore, one of the founders of Intel, observed that the density of transistors being placed on individual silicon chips was doubling about every eighteen

months. This observation has been remarkably accurate since Moore first stated it and it has since become known as Moore's Law. Memory capacity, more then anything else, has been an excellent example of the accuracy of Moore's Law. Figure 1.1, is a semi-logarithmic graph of memory capacity versus time. Many circuit designers and device physicists are arguing about the continued viability of Moore's Law. Transistors cannot continue to shrink indefinitely, nor can manufacturers easily afford the cost of the manufacturing equipment required to produce silicon wafers of such minute dimensions. At some point, the laws of quantum physics will begin to alter the behavior of these tiny transistors in very profound ways.

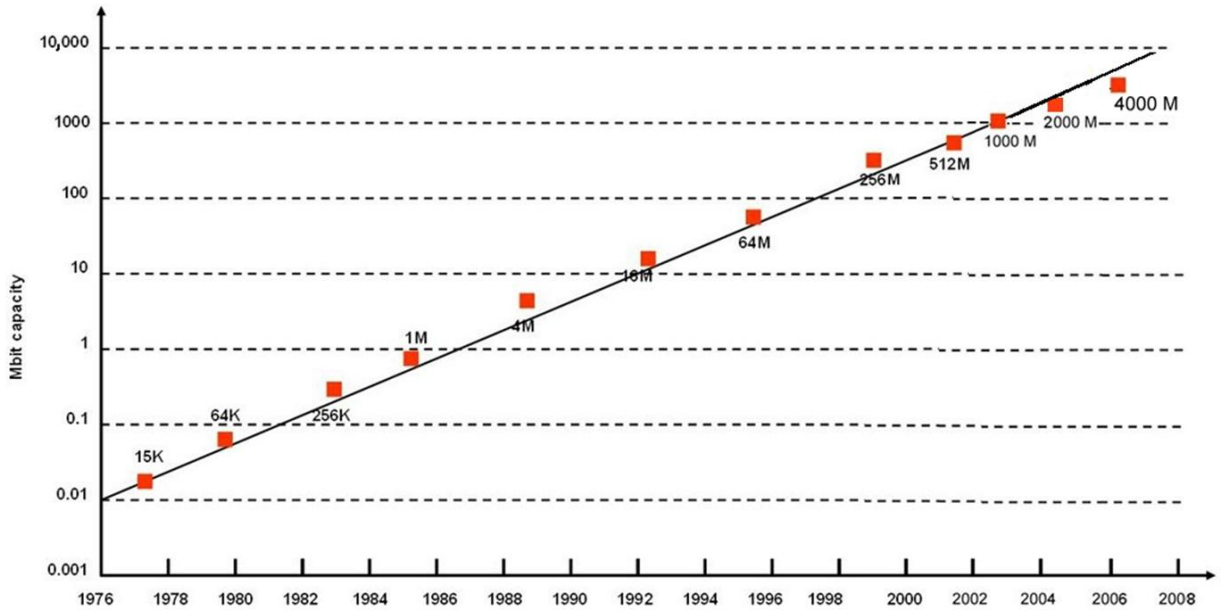


Figure 1.1: The growth in the capacity of dynamic random access memories (DRAM) with time. Note the semi-logarithmic behavior, characteristic of Moore's Law.

Today, we are capable of placing hundreds of millions of transistors (the "active" switching device from which we create logic gates) onto a single perfect piece of silicon, perhaps 2 cm on a side. From the point of view of designing computer chips, the big breakthrough came about when Mead and Conway[1] described a method of creating hardware designs by writing software. Called ***silicon compilation***, it has led to the creation of ***hardware description language***, or HDL. Hardware description languages such as Verilog® and VHDL® enable hardware designers to write a program that looks remarkably like the C programming language, and then to compile that program to a recipe that a semiconductor manufacturer can use to build a chip.

Back to the beginning; the first generation of computing engines was comprised of the mechanical devices. The abacus, the adding machine, the punch card reader for textile machines fit into this category. The next generation spanned the period from 1940–1960. Here electronic devices—vacuum tubes—were used as the active device or switching element. Even a miniature vacuum tube is millions of times larger than the transistor on a silicon wafer. It consumes millions of times the power of the transistor and its useful lifetime

is hundreds or thousands of times less than a transistor. Although the vacuum tube computers were much faster than the mechanical computers of the preceding generation, they are thousands of times slower then the computers of today. If you are a fan of the grade B science fiction movies of the 1950's, these computers were the ones that filled the room with lights flashing and meters bouncing.

The third generation covered roughly the period of time from 1960 to 1968. Here the transistor replaced the vacuum tube, and suddenly the computers began to be able to do real work. Companies such as IBM®, Burroughs® and Univac® built large mainframe computers. The IBM 360 family is a representative example of the mainframe computer of the day. Also at this time, Xerox® was carrying out some pioneering work on the human/computer interface at their Palo Alto Research Center, Xerox PARC. Here they studied what later would become computer networks, the windows-based operating systems, and the ubiquitous mouse. Programmers stopped programming in machine language and assembly language and began to use FORTRAN, COBOL and BASIC.

The fourth generation, roughly 1969-1977, was the age of the minicomputer. The minicomputer was the computer of the masses. It wasn't quite the PC, but it moved the computer out of the sterile environment of the "computer room", protected by technicians in white coats, to a computer in your lab. The minicomputer also represented the replacement of individual electronic parts, such as transistors and resistors, mounted on printed circuit boards (called discrete devices), with integrated circuits, or collections of logic functions in a single package. Here was the introduction of the small and medium scale integrated circuits. Companies such as Digital Equipment Company (DEC), Data General and Hewlett-Packard all built this generation of minicomputer.[2] Also within this timeframe, simple integrated-circuit microprocessors were introduced and commercially produced by companies like Intel, Texas Instruments, Motorola, MOS Technology and Zilog. Early microcomputer devices that best represent this generation are the 4004, 8008 and 8080 from Intel, the 9900 from Texas Instruments and the 6800 from Motorola. The computer languages of the fourth generation were assembly, C, Pascal, Modula, Smalltalk, and Microsoft BASIC.

We are currently in the fifth generation, although it could be argued that the fifth generation ended with the Intel® 80486 microprocessor and the introduction of the Pentium® represents the sixth generation. We'll ignore that distinction until it is more widely accepted. The advances made in semiconductor manufacturing technology best characterize the fifth generation of computers. Today's semiconductor processes typify what is referred to as Very Large Scale Integration, or VLSI technology. The next step, Ultra Large Scale Integration, or ULSI is either here today or right around the corner. Dr. Daniel Mann[3], an AMD Fellow, recently told me that a modern AMD Athlon XP processor contains approximately 60 million transistors.

The fifth generation also saw the growth of the personal computer and the operating system as the primary focus of the machine. Standard hardware platforms controlled by standard operating systems enabled thousands of developers to create programs for these systems. In terms of software, the dominant languages became Ada, C++, Java, HTML, and XML. In

addition, graphical design language, based upon the Universal Modeling Language, UML, began to appear.

Two Views of Today's Computer

The modern computer has become faster and more powerful but the basic architecture of a computing machine has essentially stayed the same for many years. Today we can take two equivalent views of the machine: the hardware view and the software view. The hardware view, not surprisingly, focuses on the machine and does allow for the fact that the software has something to do with its reason to exist. From 50,000 feet, our computer looks like figure 1.2.

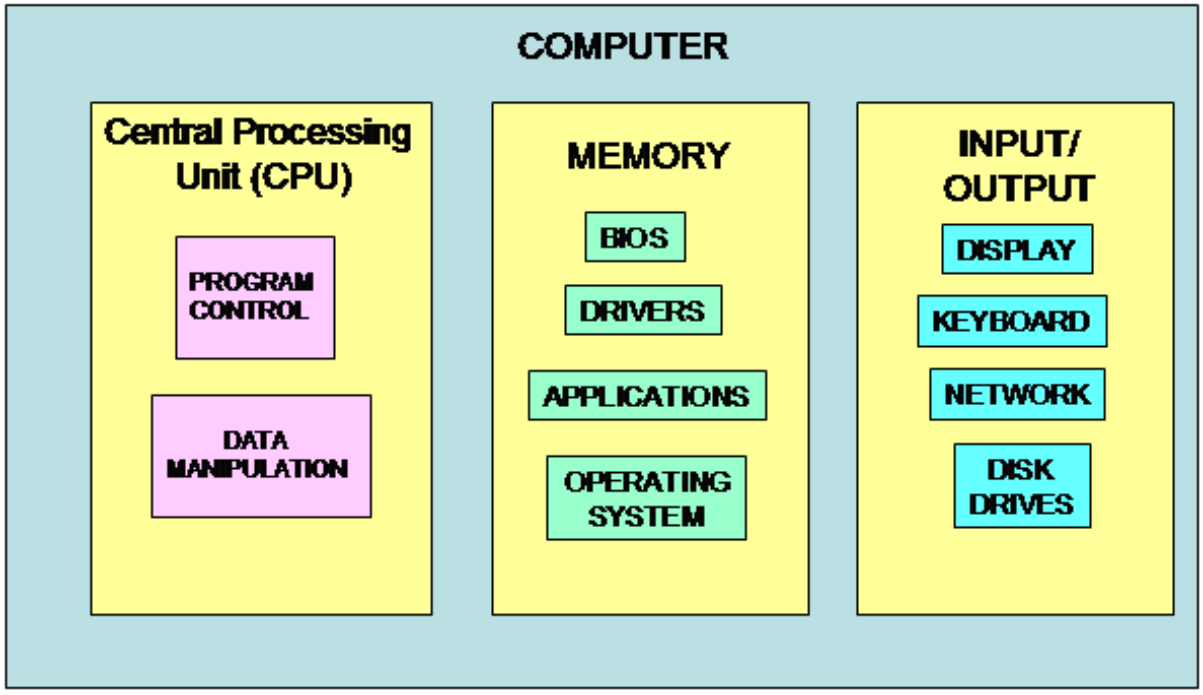


Figure 1.2: Abstract view of a computer. The three main elements are the control and data processor, the input and output, or I/O devices and the program that is executing on the machine.

In this course we'll focus primarily on the CPU and memory systems, with some consideration of the software that drives this hardware. We'll touch briefly on I/O, since a computer isn't much good without it.

The software developer's view is roughly equivalent, but the perspective does change somewhat. Figure 1.3 shows the computer from the software designer's point of view. Note that the view of the system shown in figure 1.3 is somewhat problematic because it isn't always clear that the User Interface communicates directly with the application program. In many cases, the user interface first communicates with operating system. However, let's look at this diagram somewhat loosely and consider it to be the flow of information, rather than the flow of control.

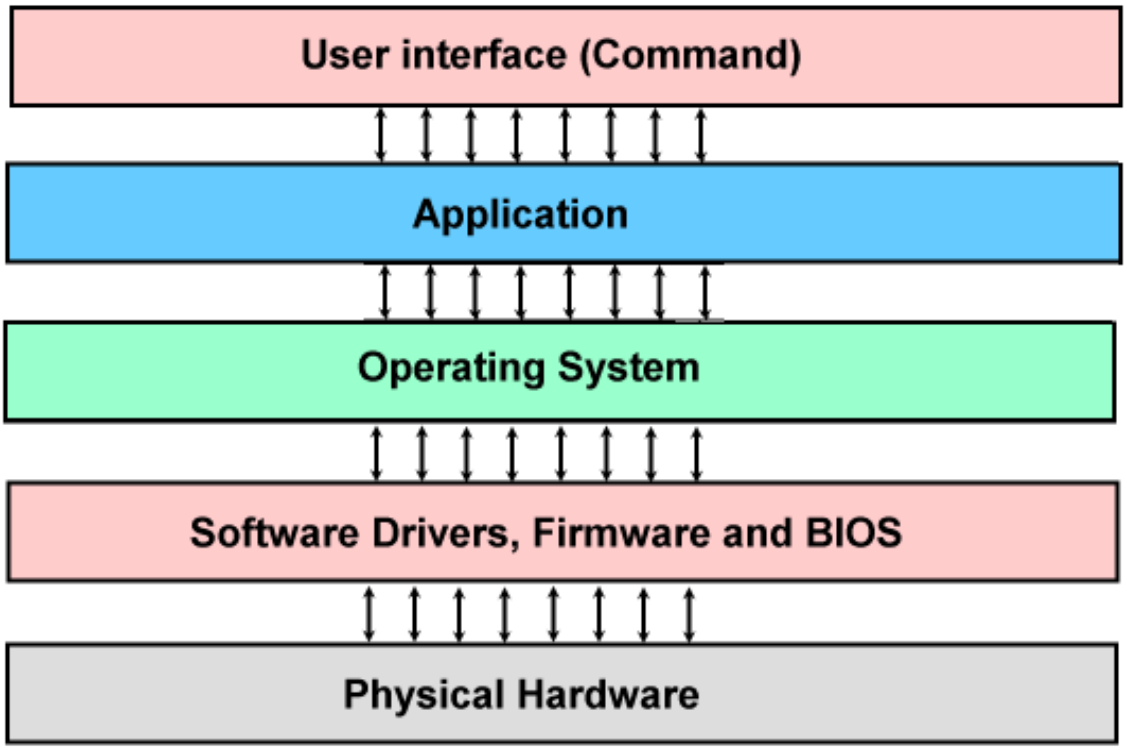


Figure 1.3: Representing the computer in terms of the abstraction levels of the system.

Abstraction Levels

One of the more modern concepts of computer design is the idea of abstraction levels. Each level provides an abstraction of the level below it. At the lowest level is the hardware. In order to control the hardware it is necessary to create small programs, called drivers, which actually manipulate the individual control bits of the hardware.

Sitting above the drivers is the operating system and other system programs. The operating system, or ***OS,*** communicates with the drivers through a standard set of application programming interfaces, or ***APIs***. The APIs provide a structure by which the next level up in the abstraction level can communicate with the layer below it. Thus, in order to read a character from the keyboard of your computer, there is a low-level driver program that becomes active when a key is struck. The operating system communicates with this driver through its API.

At the next level, the application software communicates with the OS through system API's that once again abstract the lower levels so that the individual differences in behavior of the hardware and the drivers may be ignored. However, we need to be a bit careful about taking the viewpoint of figure 1.3 too literally. We could also argue that the Application layer and the Operating System layers should be reversed because the user interacts with the application through the Operating System layer as well. Thus, mouse and keyboard inputs are really passed to the application through the Operating System and do not go directly from the User to the Application. Anyway, you get the picture.

The computer hardware, as represented by a desktop PC, can be thought of as being comprised of four basic parts.

1. Input devices can include components such as the mouse, keyboard, microphone, modem and the network.
2. Output devices are components such as the display, modem, sound card and speakers, and the network
3. The memory system is comprised of internal and external caches, main memory, video memory and disk.
4. The central processing unit, or CPU is comprised of the arithmetic and logic unit (ALU), control system and busses.

Busses

The busses are the nervous system of the computer. They connect the various functional blocks of the computer both internally and externally. Within a computer, a bus is a grouping of similar signals. Thus, your Pentium processor has a 32-bit address bus and a 32-bit data bus. In terms of the bus structure this means that there are two bundles of 32 wires with each bundle containing 32 individual signals, but with a common function. We'll discuss busses much more thoroughly in a later lesson.

The typical computer has three busses: one for memory addresses, one for data, and one for status (housekeeping and control). There are also industry standard busses such as PCI, ISA, AGP, PC-105, VXI and so forth. Since the signal definitions and timing requirements for these industry-standard busses are carefully controlled by the standards associations that maintains them, hardware devices from different manufacturers can generally be expected to work properly and interchangeably. Some busses are quite simple—only one wire—but the signals sent down that wire may be quite complex and require special hardware and software protocols to understand it. Examples of these types of busses are the universal serial bus (USB), the small computer system interface bus (SCSI), Ethernet, and Firewire.

Memory

From the point of view of a software developer, the memory system is the most visible part of the computer. If we didn't have memory, we'd never have a problem with an errant pointer. But that's another story. The computer memory is the place where program code (instructions) and variables (data) are stored. We can make a simple analogy about instructions and data. Consider a recipe to bake a cake. The recipe itself is the collection of instructions that tell us how to create the cake. The data represents the ingredients we need that the instructions manipulate. It doesn't make much sense to sift the flour if you don't have flour to sift.

We may also describe memory as a hierarchy, based upon speed. In this case, speed means how fast the data can be retrieved from the memory when the computer requests it. The fastest memory is also the most expensive, so as the memory access times become slower, the cost per bit decreases, so we can have more of it. The fastest memory is also the memory

that's closest to the CPU. Thus, our CPU might have a small number of on-chip data registers, or storage locations, several thousand locations of off-chip cache memory, several million locations of main memory and several billion locations of disk storage. The ratio of the access time of the fastest on-chip memory to the slowest memory, the hard disk, is about 10,000 to 1. The ratio of the cost of the two memories is somewhat more difficult to calculate because the fastest semiconductor memory is the on-chip cache memory, and you cannot buy that separately from the microprocessor itself. However, if we estimate the ratio of the cost per gigabyte of the main memory in your PC to the cost per gigabyte of hard disk storage (and taking into account the mail-in rebates) then we find that the faster semiconductor storage with an average access time of 20-40 nanoseconds is 300 times more costly than hard disk storage, with an average access time of 1 millisecond.

Today, because of the economy of scale provided by the PC industry, memory is incredibly inexpensive. A single in-line memory module (SIMM) with a capacity of 2 billion storage locations costs approximately $50. PC memory is dominated by a memory technology called dynamic random access memory, or ***DRAM.*** There are several variations of DRAM, and we'll cover them in greater depth later on. DRAM is characterized by the fact that it must be constantly accessed or it will lose its stored data. This forces us to create highly specialized and complex support hardware to interface the memory systems to the CPU. These devices are contained in support chipsets that have become as important to the modern PC as the CPU. Why use these complex memories? DRAMs are inherently very dense and can hold upwards of 512 million bits of information. In order to achieve these densities, the complexity of accessing and controlling them was moved to the chipset.

Static Ram (SRAM)

The memory that we'll focus on is called static random access memory or SRAM. Each memory cell of an SRAM device is more complicated than the DRAM, but the overall operation of the device is easier to understand, so we'll focus on this type of memory in our discussions. The term, static random access memory, or SRAM, refers to the fact that:

1. We may read from the chip or write data to it.
2. Any memory cell in the chip may be accessed at any time, once the appropriate address of the cell is presented to the chip.
3. As long as power is applied to the memory, we are only required to provide an address to the SRAM cell, together with a READ or a WRITE signal, in order to access, or modify, the data in the cell.

Point #3, above, is quite a bit different than the effort required to maintain data integrity in DRAM cells. Well discuss this point in greater detail in a later chapter.

With RAM memory there is no need to search through all the preceding memory cells in order to get to the one you want to read. In other words, data can be read from the last cell of the RAM as quickly as it could be read from the first cell. In contrast, a tape backup device

must stream through the entire tape before the last piece of data can be retrieved. That's why the term "random access" is used. Also note that when we are discussing SRAM or DRAM in the general sense, as listed in items 1 and 2, above, we'll just use the term ***RAM***, without the SRAM or DRAM distinction.

Memory Hierarchy

Figure 1.4 shows the memory hierarchy in real terms, where we can see how the various memory elements grow exponentially in size and decrease exponentially in access time. Closest memory to the CPU is the smallest, typically in the range of 1K Byte of data (approximately 1,000 8-bit characters) to 1 M Byte of data (1,000,000 8-bit characters). These on-chip cache memories can have access times as short as ½ to 1 nanosecond, or one-billionth of a second. As a benchmark, light can travel about one foot through the air in one nanosecond. We call this cache the ***Level 1***, or ***L1*** cache.

Below the L1 cache is the ***Level 2,*** or ***L2,*** cache. In today's Pentium-class processors, the L2 cache is usually on the processor chip itself. In fact, if you could lift the lid of a Pentium or Athlon processor and look at the silicon die itself under a microscope you might be surprised to find that the biggest percentage of chip area was taken up by the cache memories.

Underneath the secondary cache sits your computer's main memory. These are the "memory sticks" that you can purchase in a computer shop to increase your computer's performance. This memory is considerably slower than on-chip cache memory, but you usually have a much larger main memory than the on-chip memory. You might wonder why adding more memory will give you better performance. To see this, consider the next level down from main memory, the hard disk drive. Your hard drive has lots of capacity, but it comes at a price, speed. The hard disk is an electromechanical device. The data is stored on a rotating platter and, even at a rotational speed of 7200 revolutions per minute, it takes precious time for the correct data to come under the disk read heads. Also, the data is organized as individual tracks on each side of multiple platters. The heads have to move from track-to-track in order to access the correct data. This takes time as well.

Now, your operating system, whether it's MAC O/S, Linux or one of the flavors of Windows, uses the hard disk as a handy place to swap out programs or data that can't fit in main memory at a particular point in time. So, if you have several windows open on your computer, and only 64 megabytes of main memory, you might see the hourglass form of the cursor appearing quite often because the operating system is constantly swapping the different applications in and out of main memory. From figure 1.4 we see that the ratio of access times between the hard disk and main memory can be 10,000 to 1, so any time that we go to the hard disk, we will have to wait. The moral here is that the best performance boost you can give to your computer is to add as much memory as it can hold.

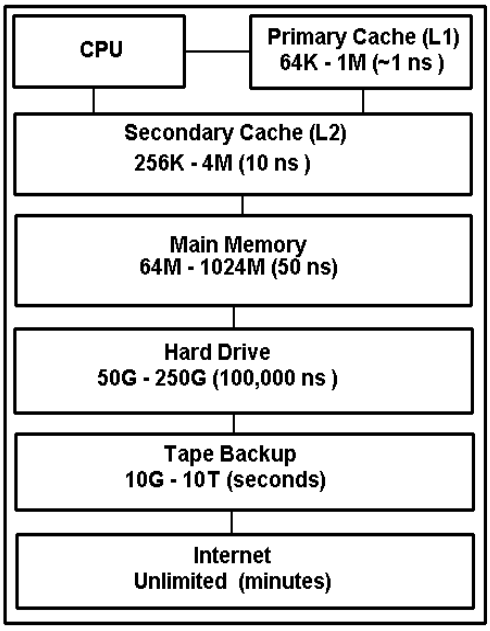


Figure 1.4: The memory hierarchy. Notice the inverse relationship between the size of the memory and the access time.

Finally, there are several symbols used in figure 1.4 that might be strange to you. We'll discuss them in detail in due course, but in case you were wondering what it meant, here's a little preview.

Symbol	Name	Meaning
ns	nanosecond	billionth of a second
K	kilobytes	2^{10} or 1024 8-bit characters (bytes)
M	megabytes	2^{20} or 1,048,576 bytes
G	gigabytes	2^{30} or 1,073,741,824 bytes
T	terabytes	2^{40} or 1,099,511,627,776 bytes

Hopefully, these numbers will soon become quite familiar to you because they are the dimensions of modern computer technology. However, one note of caution; the terms ***kilo, mega, giga*** *and* ***tera*** are often overloaded. Sometimes they are used in the strictly scientific sense of a shorthand notation for a multiplication factor of 10^3, 10^6, 10^9 and 10^{12} respectively. So, how do you know if the computer-speak versions, 2^{10}, 2^{20}, 2^{30}, 2^{40} are being used, or if the traditional science and engineering meanings are being used?

That's a good question. Sometimes it isn't so obvious and mistakes could be made. For example, whenever we are dealing with memory size and memory issues, we are almost always using the base 2 sense of the terms. However, this isn't always the case. Hard disk drives, even though they are storage devices, use the terms in the engineering sense.

Therefore, an old 1gigabyte hard drive does not hold the same amount of data as 1 gigabyte of memory because the "giga" term is being used in two different senses. In any case, you are generally safe to assume the base 2 versions of the term in this text, unless I specifically state otherwise. In the real world, ***caveat emptor.***

Hard Disk Drive

Let's look a bit further into the dynamics of the hard disk drive. Disk drives are a marvel of engineering technology. For years, electronic industry analysts were predicting the demise of the hard drive. Economical semiconductor memories that were as cost effective as a hard disk were always "a few years away". However, the disk drive manufacturers ignored the pundits and just continued increase the capacity and performance, improve the reliability and reduce the cost of the disk drives. Today, an average disk drive costs about 60 cents per gigabyte of storage capacity.

Consider a modern, high-performance disk drive. Specifically, let's look at the Model ST3146807LC from Seagate Technology® [4]. The following are the relevant specifications for this drive:

- Rotational Speed: 10,000 rpm
- Interface: Ultra320 SCSI
- Discs/Heads: 4/8
- Formatted Capacity (512 bytes/sector) GB: 146.8
- Cylinders: 49855
- Sectors per drive: 286,749,488
- External transfer rate: 320 Mbytes per second
- Track-to-track Seek Read/Write (msec): 0.35/0.55
- Average Seek Read/Write (msec): 4.7/5.3
- Average Latency (msec): 2.99

What does all this mean? Consider figure 1.5. Here we see a highly simplified schematic diagram of the Seagate ***Cheetah®*** disk drive discussed above. The hard disk is comprised of 4 aluminum platters. Each side of the platter is coated with magnetic material that records the stored data. Above each platter is a tiny magnetic pick-up, or ***head***, that floats above the disk platter (disc) on a cushion of air. In a very real sense, the head is flying over the surface of the platter, a distance much less than the thickness of a human hair. Thus, when you get a disk crash, the results are quite analogous to when an airplane crashes. In either case, the airplane or the read/write head loses lift and hits the ground or the surface of the disk. When that happens the magnetic material is scraped away and the disk is no longer usable.

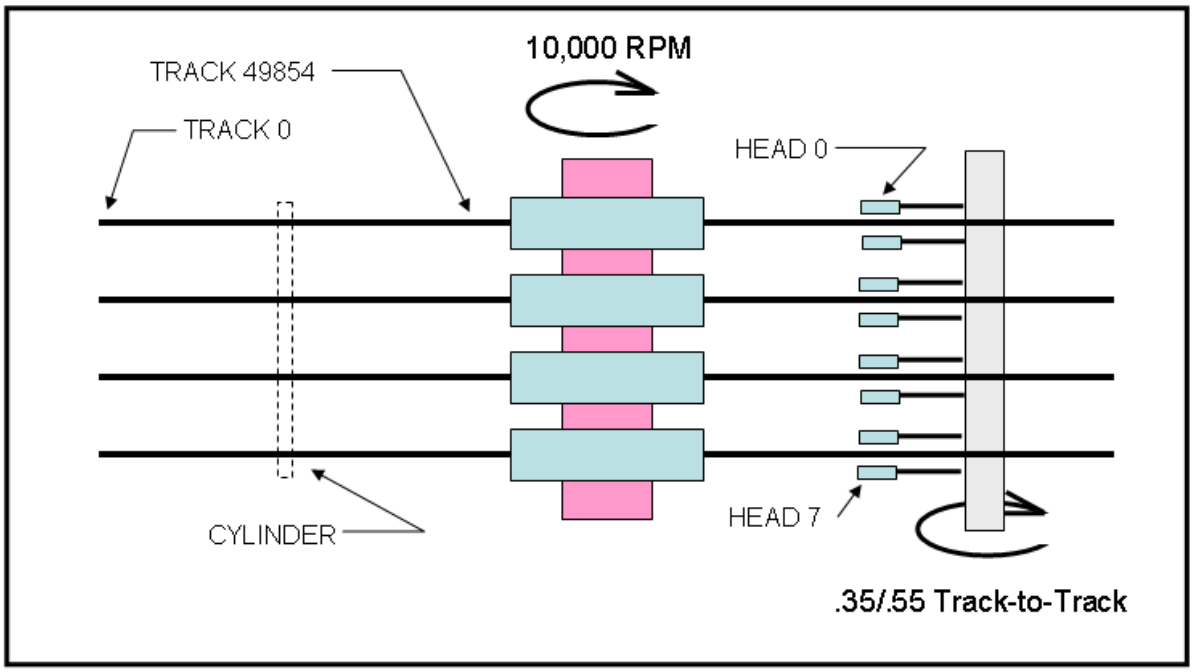


Figure 1.5 Schematic diagram of the Seagate ***Cheetah***® hard disk drive

Each surface contains 49,855 concentric tracks. Each track is discrete. It is not connected to the adjacent track, as a spiral. Thus, in order to move from one track to an adjacent track, the head must physically move a slight amount. All heads are connected to a common shaft that can quickly and accurately rotate the heads to a new position. Since the tracks on all the surfaces are in vertical alignment, we call this a ***cylinder***. Thus, cylinder 0 contains 8 tracks from the 4 platters.

Now, how do we get a 146.8 GB drive? First, each platter contains 2 surfaces and we have 4 platters, so that the total capacity of the drive will be 8 times the capacity of one surface. Each sector holds 512 bytes of data. By dividing the total number of sectors by 8, we see that there are 35,843,686 sectors per surface. Dividing again by 49,855, we see that there are approximately 719 sectors per track. It is interesting that the actual number is 718.96 sectors per track. Why isn't this value a whole number? In other words, how can we have a fractional number of sectors per track?

There are a number of possibilities and we need not dwell on it. However, one possibility is that the number of sectors per track is not uniform across the disk surface because the sector spacing changes as we move from the center of the disk to the outside. In a CD-ROM or DVD drive this is corrected by changing the rotational speed of the drive as the laser moves in and out. But, since the hard disk rotates at a fixed rate, changing the recording density as we move from inner to outer tracks makes the most sense.

Anyway, back to our calculation. If each track holds 719 sectors and each sector is 512 bytes, then each track holds 368,128 bytes of data. Since there are 8 tracks per cylinder, each cylinder holds 2,945,024 bytes. Now, here's where the big numbers come in. Since we have 49,855 cylinders, our total capacity is 146,824,171,520 bytes, or 146.8 GB.

Before we leave the subject of disk drives, let's consider one more issue. Considering the hard drive specifications, above, we see that access times are measured in units of milliseconds (ms), or thousandths of a second. Thus, if your data sectors are spread out over the disk, then accessing each block of 512 bytes can easily take seconds of time. Comparing this to the time required to access data stored in main memory, it is easy to see why the hard drive is 10,000 times slower than main memory.

Complex Instruction Set Architecture, and RISC, or Reduced Instruction Set Computer

Today we have two dominant computer architectures, complex instruction set computer architecture, or ***CISC*** and reduced instruction set computer architecture, or ***RISC***. CISC is typified by what is referred to as the Von Neumann Architecture, invented by John Von Neumann of Princeton University. In the Von Neumann architecture, both the instruction memory and the data memory share the same physical memory space. This can lead to a condition called the ***Von Neumann bottleneck***, where the same external address and data busses must serve double duty, transferring instructions from memory to the processor for program execution, and moving data to and from memory for the storage and retrieval of program variables.

When the processor is moving data, it can't fetch the next instruction, and vice versa. The solution to this dilemma, as we shall see later, is the introduction of separate on-chip instruction and data caches. Motorola's 68000 processor and its successors, and Intel's 8086 processor and its successors are all characteristic of CISC processors. We'll take a more in-depth look at the differences between CISC and RISC processors again later when we study pipelining.

Note:
You will often see the processor represented as 80X86, or 680X0. The "X" is used as a placeholder to represent a member of a family of devices. Thus, the 80X86 (often written as X86) represents the 8086, 80186, 80286, 80386, 80486 and 80586 (the designation of the first Pentium processor). The Motorola processors are the 68000, 68010, 68020, 68030, 68040 and 68060.

Howard Aiken of Harvard University designed an alternate computer architecture that we commonly associate today with the Reduced Instruction Set Computer, or RISC architecture. The classic Harvard Architecture computer has two entirely separate memory spaces, one for instructions and one for data. A processor with these memories could operate much more efficiently because data and instructions could be fetched from the computer's memory when needed, not just when the busses were available. The first popular microprocessor that used the Harvard Architecture was the Am29000 from AMD. This device achieved reasonable popularity in the early Hewlett-Packard laser printers, but later fell from favor because designing a computer with two separate memories was just not cost effective. The obvious performance gain from the two memory spaces was the driving force to inspire the CPU designers to move the memory spaces onto the chips, in the form of the instruction and data caches that are common in today's high-performance microprocessors.

The gains in CPU power are quite evident if we look at advances in workstation performance over the past few years. It is interesting that just four years later, a high-end PC, such as a 2.4 GHz Pentium 4 from Intel could easily outperform the DEC Alpha 21254/600 workstation.

Figure 1.6[5] is a graph of workstation performance over time.

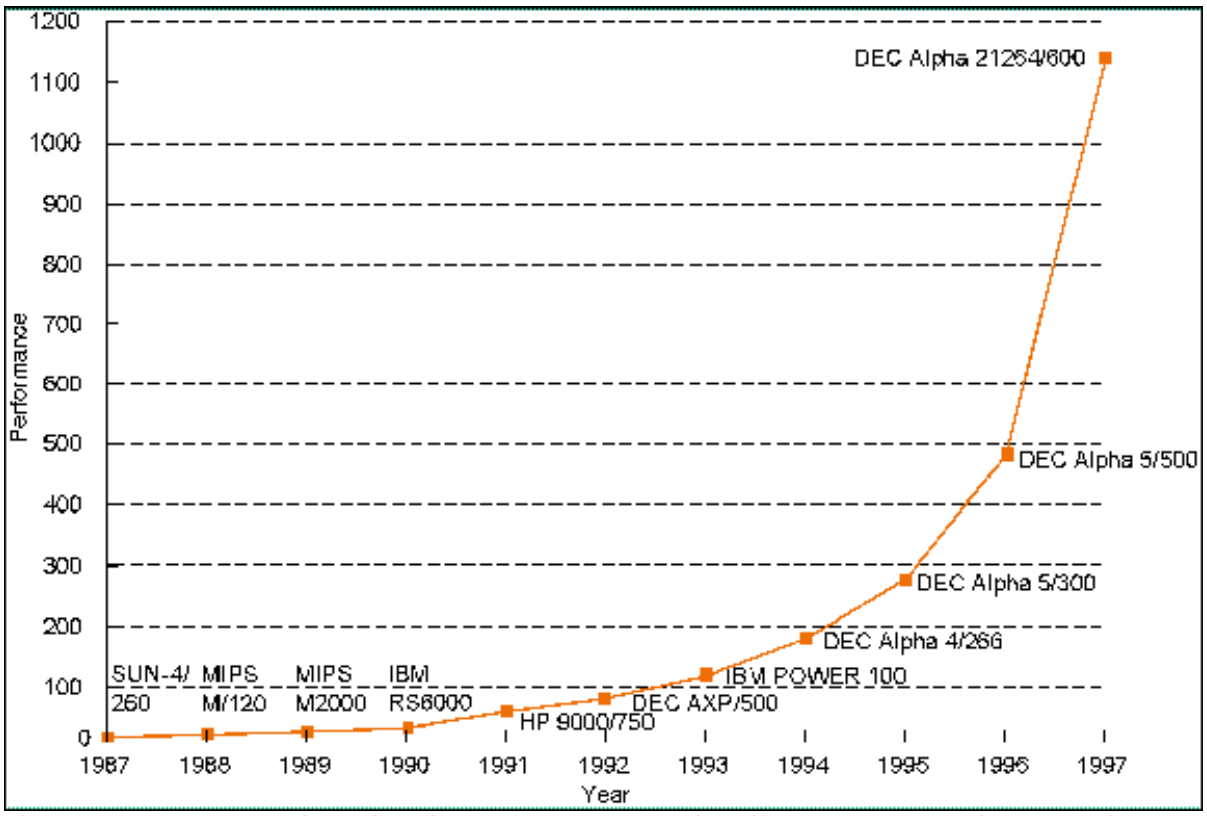


Figure 1.6: Improvement in workstation performance over time (*from Patterson and Hennessy)*

The relative performance of these computers was measured using a standard set of programs called ***benchmarks***. In this case, the industry standard SPECbase_int92 benchmark was used. It is difficult to compare performance based on these numbers with more modern performance measurements because the benchmark changed along the way. Today, the benchmark of choice is the SPECint95, which does not directly correlate with the earlier SPECint92 benchmark[6]. However, according to Mann, a conversion factor of about 38 allows for a rough comparison. Thus, according to the published results[7], a 1.0GHz AMD Athlon processor achieved a SPECint95 benchmark result of 42.9, which roughly compares to a SPECint92 result of 1630. The Digital Equipment Corporation (DEC) AlphaStation 5/300 is one workstation that has published results for both benchmark tests. It measures about 280 in the graph of figure 1.6 and 7.33 according to the SPECint95 benchmark. Multiplying by 38, we get 278.5, which is in reasonable agreement with the earlier result. We'll return to the issue of performance measurements in a later chapter.

Number Systems

How do you represent a number in a computer? How do you send that number, whatever it may be, a ***char***, an ***int***, a ***float*** or perhaps a ***double*** between the processor and memory, or within the microprocessor itself? This is a fair question to ask, and the answer leads us naturally to an understanding of why modern digital computers are based on the binary (base 2) number system. In order to investigate this, consider figure 1.7

In figure 1.7 we'll do a simple-minded experiment. Let's pretend that we can place an electrical voltage on the wire that represents the number we would like to transmit between two functional elements of the computer. The method might work for simple numbers, but I wouldn't want to touch the wire if I was sending 2000.456! In fact, this method would be extremely slow, expensive and would only work for a narrow range of values.

However, that doesn't imply that this method isn't used at all. In fact, one of the first families of electronic computers was the ***analog computer***. The analog computer is based upon ***linear amplifiers***, or the kind of electronic circuitry that you might find in your stereo receiver at home. The key point is that variables (in this case the voltages on wires) can assume an infinite range of values between some limits imposed by the nature of the circuitry. In many of the early analog computers this range might be between -25 volts and +25 volts. Thus, any quantity that could be represented as a steady, or time varying voltage within this range could be used as a variable within an analog computer.

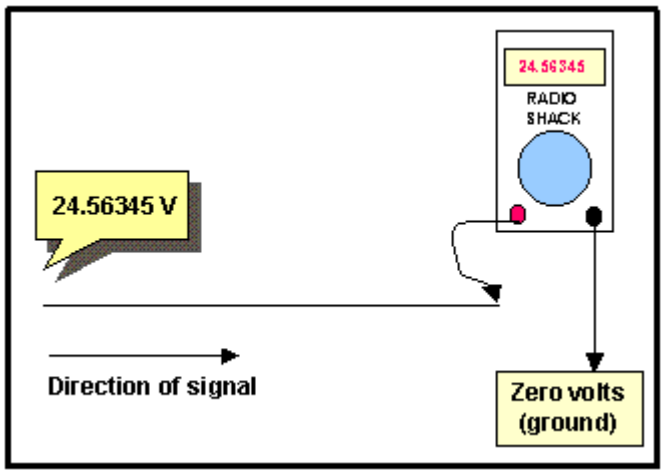


Figure 1.7 - Representing the value of a number by the voltage on a wire.

The analog computer takes advantage of the fact that there are electronic circuits that can do the following mathematical operations:

- Add / subtract
- Log / anti-log
- Multiply / divide
- Differentiate / integrate

By combining these circuits one after another with intermediate amplification and scaling, real-time systems could be easily modeled and the solution to complex linear differential equations could be obtained as the system was operating.

However, the analog computer suffers from the same limitations as does your stereo system. That is, its amplification accuracy is not infinitely perfect, so the best accuracy that could be hoped for is about 0.01%, or about 1 part in 10,000. Figure 1.8 shows an analog computer of the type used by the United States submarines during World War II. The Torpedo Data Computer, or TDC, would take as its inputs:

- The compass heading and speed of the target ship
- The heading and speed of the submarine
- The desired firing distance

The correct speed and heading was then sent to the torpedoes and they would track the course, speed and depth transmitted to them by the TDC.

Figure 1-8: An analog computer from a WWII submarine. *Picture courtesy of www.fleetsubmarine.com*

Thus, within the limitations imposed by the electronic circuitry of the 1940's, an entire family of computers was based upon a system where the inputs and outputs were continuous variables. In that sense, your stereo amplifier is an analog computer. An amplifier ***amplifies***, or boosts, an electrical signal. An amplifier with a ***gain*** of 10, has an output voltage that is, at every instant of time, 10 times greater than the input voltage. Thus, $V_{out} = 10V_{in}$. Here we have an analog computing block that happens to be a multiplication block with a constant multiplier.

Anyway, let's get back to discussing the number systems. We might be able to improve on this method by breaking the number into more manageable parts and send a more limited signal range over several wires at the same time (in parallel). Thus, each wire would only need to transmit a narrow range of values. Figure 1.9 shows how this might work.

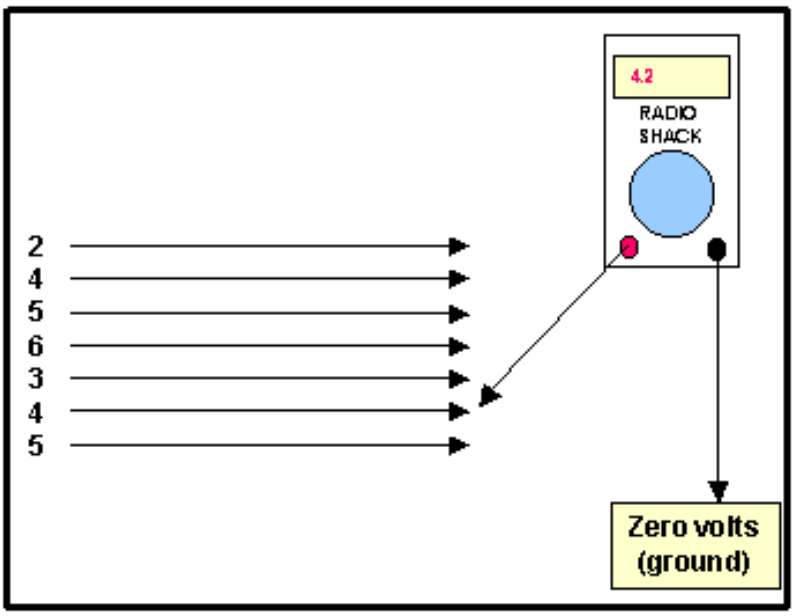


Figure 1.9 - Using a parallel bundle of wires to transmit a numeric value in a computer. The wire's position in the bundle determines its numeric weight. Each wire carries between 0 volts and 9 volts.

In this case each wire in the bundle represents a decimal decade and each number that we send would be represented by the corresponding voltages on the wires. Thus, instead of needing to transmit potentially lethal voltages, such as 12,567 volts, the voltage in each wire would never become greater than that of a 9-volt battery. Let's stop for a moment because this approach looks promising.

How accurate would the voltage on the wire have to be so that the circuitry interprets the number as 4, and not 3 or 5? In figure 1.9 our voltmeter shows that the second wire from the bottom measures 4.2V, not 4 volts. Is that good enough? Should it really be 4.000 +/- .0005 volts? In all probability, this system might work just fine if each voltage increment has a "slop" of about 0.3 volts. So we would only need to send 4 volts +/- 0.3 volts (3.7 - 4.3 volts) in order to guarantee that the circuitry received the correct number. What if the circuit erred and sent 4.5 volts instead? This is too large to be a 4 but too small to be a 5. The answer is that we don't know what will happen. The value represented by 4.5 volts is ***undefined***. Hopefully, our computer works properly and this is not a problem.

The method proposed in figure 1.9 is actually very close to reality, but it isn't quite what we need. With the speed of modern computers, it is still far too difficult to design circuitry that is both fast enough and accurate enough to switch the voltage on a wire between 10 different values. However, this idea is being looked at for the next generation of computer memory cells. More on that later, stay tuned!

Modern transistors are excellent switches. They can switch a voltage or a current on or off in trillionths of a second (picoseconds). Can we make use of this fact? Let's see. Suppose we extend the concept of a bundle of wires but let's restrict even further the values that can exist on any individual wire. Since each wire is controlled by a switch; we'll switch between

nothing (0 volts) and something (~3 volts). This implies that just two numbers may be carried on each wire, 0 or something (not 0). Will it work? Let's look at figure 1.10.

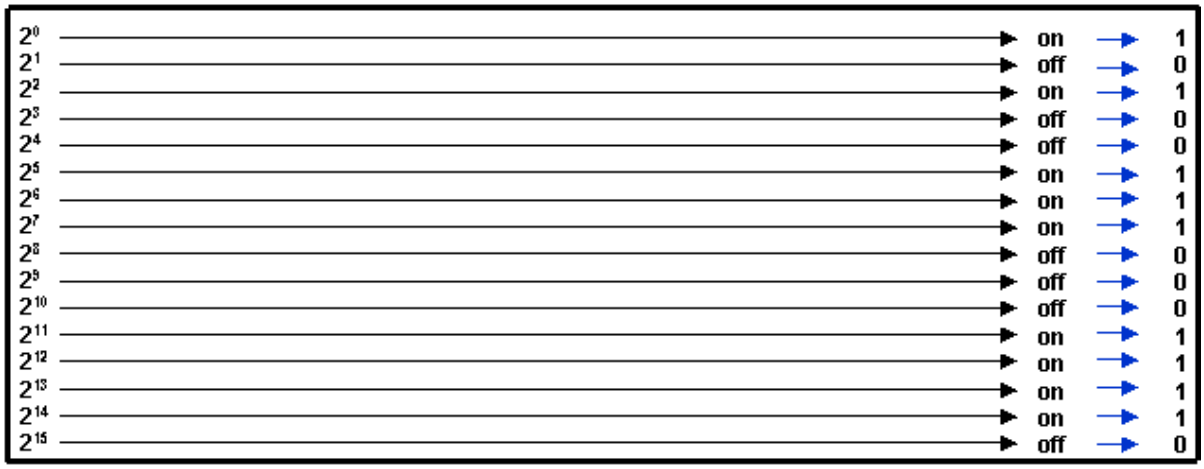


Figure 1.10: Sending numbers as binary values. Each arrow represents a wire with the arrowhead representing the direction of signal transmission. The position of each wire in the bundle represents its numerical weight. Each row represents an increasing power of 2.

In this scenario, the amount of information that we can carry on a single wire is limited to nothing, 0, or something (let's call something "1" or "on"), so we'll need a lot of wires in order to transmit anything of significance. Figure 1.10 shows 16 wires, and as you'll soon see, this limits us to numbers between 0 and 65,535 if we are dealing with unsigned numbers, or the signed range of -32,768 to +32,767. Here the decimal number 0 would be represented by the binary number 0000000000000000 and the decimal number 65,535 would be represented by the binary number 1111111111111111.

Note: For many years most standard digital circuits used 5 volts for a 1. However, as the integrated circuits became smaller and denser, the logical voltage levels also had to be reduced. Today, the core of a modern Pentium or Athlon processor runs at a voltage of around 1.7-1.8 volts, not very different from a standard AA battery.

Now we're finally there. We'll take advantage of the fact that electronic switching elements, or transistors, can rapidly switch the voltage on a wire between two values. The most common form of this is between almost 0 volts and something (about 3 volts). If our system is working properly, then what we define as "nothing", or 0, might never exceed about ½ of a volt. So, we can define the number 0 to be any voltage less than ½ volt (actually, it is usually 0.4 volts). Similarly, if the number that we define as a 1, would never be less than 2.5 volts, then we have all the information we need to define our number system. Here, the number 0 is never greater than 0.4 volts and the number 1 is never less than 2.5 volts. Anything between these two ranges is considered to be undefined and is not allowed.

It should be mentioned that we've been referring to "the voltage on a wire". Just where are the wires in our computer? Strictly speaking, we should call the wires "electrical conductors. They can be real wires, such as the wires in the cable that you connect from your printer to

the parallel port on the back of your computer. They can also be thin conducting paths on printed circuit boards within your computer. Finally, they can be tiny aluminum conductors on the processor chip itself. Figure 1.11 shows a portion of a printed circuit board from a computer designed by the author.

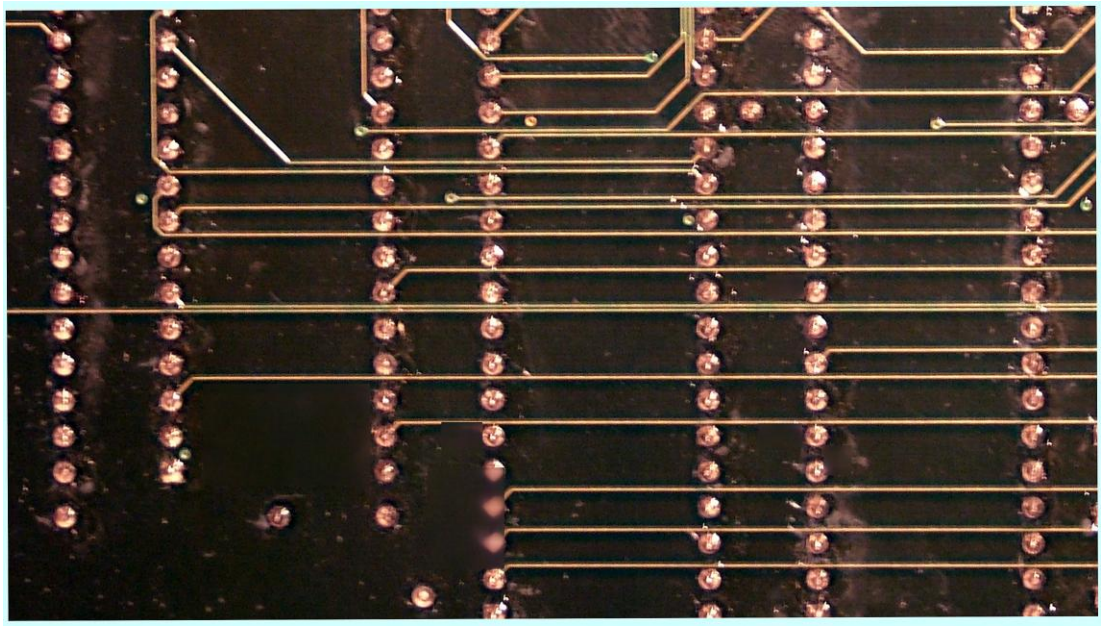


Figure 1.11: Printed wires on a computer circuit board. Each wire is actually a copper ***trace*** approximately 0.08 mm wide. Traces can be as close as 0.08 mm apart from each other. The large spots are the soldered pins of the integrated circuits coming through from the other side of the board.

Notice that some of the integrated circuit's (IC's) pins appear to have wires connecting them to another device while others seem to be unconnected. The reason for this is that this printed circuit board is actually a sandwich made up of 5 thinner layers with wires printed on either side, giving a total of 10 layers. The 8 inner layers also have a thin insulating layer between them to prevent electric short circuits. During the manufacturing process the five conducting layers and the 4 insulating layers are carefully aligned and bonded together. The resultant 10-layer printed circuit board is approximately 2.5 mm thick.

Without this multilayer manufacturing technique it would be impossible to build complex computer systems because it would not be possible to connect the wires between components without having to cross a separate wire with a different purpose.

Figure 1.12 shows us just what's going on with the inner layers. Here is an x-ray view of another computer system hardware circuit. This is about the same level of complexity that you might find on the motherboard of your PC. The view is looking through the layers of the board and the conductive traces on each layer are shown in a different color.

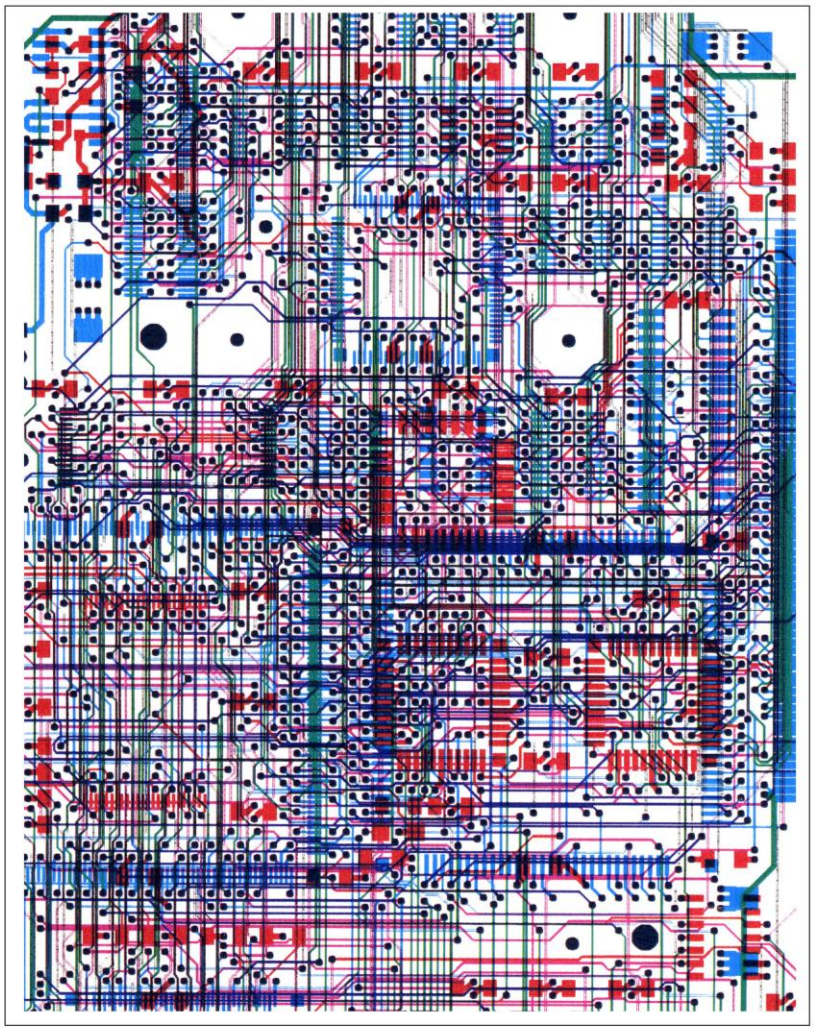


Figure 1.12: An x-ray view of a portion of a computer systems board.

While this may appear quite imposing, most of the layout was done using Computed-Aided Design (CAD) software. It would take altogether too much time for even a skilled designer to complete the layout of this board. Figure 1.13 is a magnification of a smaller portion of figure 1.12. Here you can clearly see the various traces on the different layers. Each printed wire is approximately 0.03mm wide.

If you look carefully at figure 1.13 you'll notice that certain colored wire touch a black dot and then seem to go off in another direction as a wire of a different color. The black dots are called ***vias*** and they represent places in the circuit where a wire leaves its layer and traverses to another layer. Vias are vertical conductors that allow signals to cross between layers. Without multiple layers and vias, wires couldn't cross each other on the board without short-circuiting to each other. Thus, when you see a green wire crossing a red wire, the two wires are not in physical contact, but are passing over each other on different layers of the board.

This is an important concept to keep in mind because we'll soon be looking at and drawing our own electronic circuit diagrams, called ***schematic diagrams***, and we'll need to keep in mind how to represent wires that appear to cross each other without being physically connected, and those wires that are connected to each other.

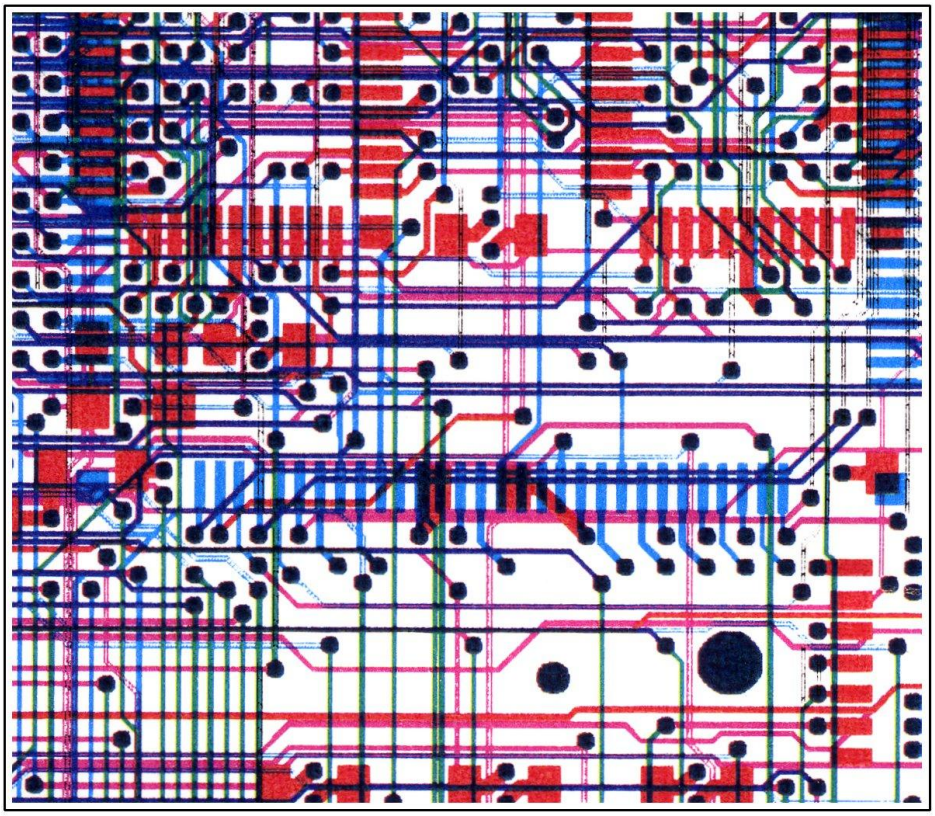


Figure 1.13: A magnified view of a portion of the board shown in figure 1.10B.

Let's review what we've just discussed. Modern digital computers use the binary (base 2) number system. They do so because a number system that has only two digits in its natural sequence of numbers lends itself to a hardware system which utilizes switches to indicate if a circuit is in a "1" state (on) or a "0" state (off). Also, the fundamental circuit elements that are used to create complex digital networks are also based on these principles as logical expressions. Thus, just as we might say, logically, that an expression is TRUE or FALSE, we can just as easily describe it as a "1" (TRUE) or "0" (FALSE). As you'll soon see, the association of 1 with TRUE and 0 with FALSE is completely arbitrary, and we may reverse the designations with little or no ill effects. However, for now, let's adopt the convention that a binary 1 represents a TRUE or ON condition, and a binary 0 represents a FALSE or OFF condition. We can summarize this in the following table:

Binary Value	Electrical Circuit Value	Logical Value
0	OFF	FALSE
1	ON	TRUE

A Simple Binary Example

Since you have probably never been exposed to electrical circuit diagrams, let's dive right in. Figure 1.14 is a simple schematic diagram of a circuit containing a battery, two switches, labeled A and B, and a light bulb, C. The positive terminal on the battery is labeled with the plus (+) sign and the negative battery terminal is labeled with the minus (-)sign. Think of a typical AA battery that you might use in your portable MP3 player. The little bump on the end is the positive terminal and the flat portion on the opposite end is the negative terminal. Referring to Figure 1.14, it might seem curious that the positive terminal is drawn as a wide line, and the negative terminal is drawn as a narrow line. There's a reason for it, but we won't discuss that here. Electrical Engineering students are taught the reason for this during their initiation ceremony, but I'm sworn to secrecy.

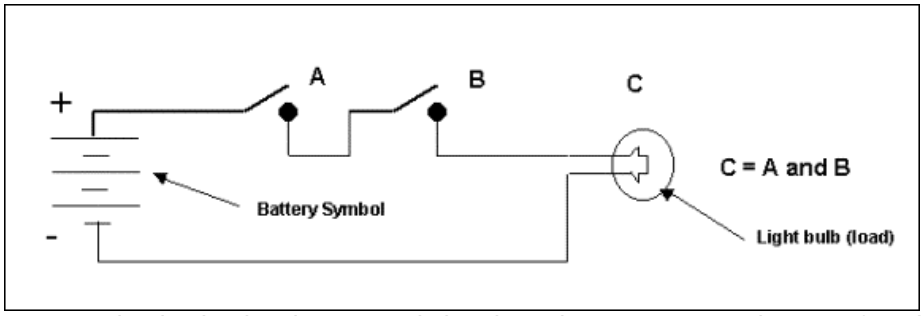


Figure 1.14: A simple circuit using two switches in series to represent the AND function.

The light bulb, C, will illuminate when enough current flows through it to heat the filament. We assume that in electrical circuits such as this one, that current flows from positive to negative. Thus, current exits the battery at the + terminal and flows through the closed switches (A and B), then through the lamp, and finally to the – terminal of the battery. Now, you might wonder about this because, as we all know from our high school science classes, that electrical current is actually made up of electrons and electrons; and electrons, being negatively charged, actually flow from the negative terminal of the battery to the positive terminal, the reverse direction.

The answer to this apparent paradox is historical precedent. As long as we think of the current as being positively charged, then everything works out just fine.

Anyway, in order for current to flow through the filament, two things must happen: switch A must be closed (ON) *and* switch B must be closed (ON). When this condition is met, the output variable, C, will be ON (illuminated). Thus, we can talk about our first example of a logical equation:

C = A AND B

This is a very interesting result. We've seen two apparently very different consequences of using switches to build computer systems. The first is that we are led to having to deal with numbers as binary (base 2) values and the second is that these switches also allow us to create logical equations. For now, let's keep item 2 as an interesting consequence. We'll deal with it more thoroughly in the next chapter. Before we leave figure 1.14 we should point out that the switches, A and B, are actuated mechanically. Someone flips the switch to turn it on or off. In general, a switch is a ***three terminal device***. There is a control input that determines the signal propagation between the other two terminals.

Bases

Let's return to our discussion of the binary number system. We are accustomed to using the decimal (base 10) number system because we had ten fingers before we had an iMAC®. The base (or ***radix***) of a number system is just the number of distinct digits in that number system. Consider the table, below:

Base 2	0,1	Binary
Base 8	0,1,2,3,4,5,6,7	Octal
Base 10	0,1,2,3,4,5,6,7,8,9	Decimal
Base 16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	Hexadecimal

Look at the hexadecimal numbers in the table above. There are 16 distinct digits, 0 through 9 and A through F, representing the decimal numbers 0 through 15, but expressing them in the hexadecimal system.

Now, if you've ever had your PC lock-up with the "blue screen of death" you might recall seeing some funny looking letters and numbers on the screen. That cryptic message was trying to show you an address value in hexadecimal where something bad has just happened. It may be of little solace to you that from now on the message on the blue screen will not only tell you that a bad thing has happened and you've just lost four hours of work, but with your new-found insight, you will know where in your PC's memory the illegal event took place.

When we write a number in binary, octal, decimal or hexadecimal, we are representing the number in exactly the same way, although the number will look quite different to us, depending upon the base we're using. Let's consider the decimal number 65,536. This happens to be 2^{16}. Later, we'll see that this has special significance, but for now, it's just a number.

Figure 1.15, shows how each digit of the number, 65,536, represents the column value multiplied by the numerical weight of the column. The leftmost, or ***most significant digit,*** is the number 6. The rightmost digit, or ***least significant digit***, also happens to be 6. The column weight of the most significant digit is 10,000 (10^4) so the value in that column is 6 x 10,000, or 60,000. If we multiply out each column value and then arrange them as a list

of numbers to be added together, as we've done on the right side of figure 1.15, we can add them together and get the same number as we started with. OK, perhaps we're overstating the obvious here, but stay tuned, because it does get better. This little example should be obvious to you because you're accustomed to manipulating decimal numbers. The key point is that the column value happens to be the base value raised to a power that starts at 0 in the rightmost column and increases by 1 as we move to the left. Since decimal is base 10, the column weights moving leftward are 1, 10, 100, 1000, 10000 and so on.

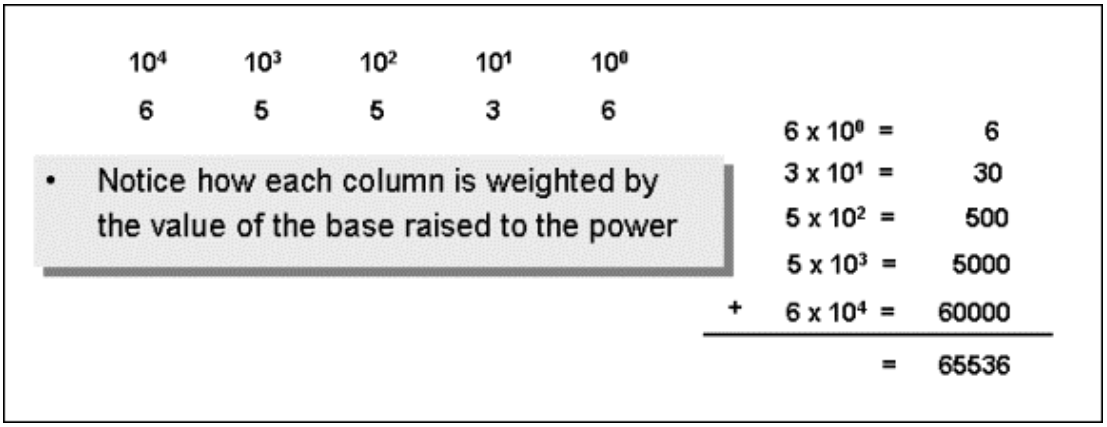


Figure 1.15: Representing a number in base 10. Going from right to left, each digit multiplies the value of the base, raised to a power. The number is just the sum of these multiples.

If we can generalize this method of representing numbers, then it follows that we could use the same method to represent numbers in any other base.

Translating numbers between bases

Let's repeat the above exercise, but this time we'll use a binary number. Let's consider the 8-bit binary number 10101100. Because this number has 8 binary digits, or bits associated with it, we call it an 8-bit number. It is customary to call an 8-bit binary number a ***byte*** (in C or C++ this is a ***char*****).** It should now be obvious to you why binary numbers are all 1's and 0's. Aside from the fact that these happen to be the two states of our switching circuits (transistors), they are the only numbers available in a base 2 number system.

The byte is perhaps most notable because we measure storage capacity in byte-sized chunks (sorry). The memory in your PC is probably at least 256M bytes (256 million bytes) and your hard disk has a capacity of 40G bytes (40 billion bytes), or more.

Consider figure 1.16. We use the same method as we used in the decimal example of figure 1.15. However, this time the column weight is a multiple of base 2, not base 10. The column weights go from 2^7, or 128, the most significant digit, down to 2^0, or 1. Each column is smaller by a factor of 2. To see what this binary number is in decimal, we use the same process as we did before; we multiply the number in the column by the weight of the column.

Figure 1.16: Representing a binary number in terms of the powers of the base 2. Notice that the bases of the octal (8) and hexadecimal (16) number systems are also powers of 2.

Thus, we can conclude that the decimal number 172 is equal to the binary number 10101100. It is also noteworthy that the bases of the hexadecimal (Hex) number system, 16, and the octal number system, 8, are also 2^4 and 2^3, respectively. This might give you a hint as to why we commonly use the hexadecimal representation and the less common octal representation instead of binary when we are dealing with our computer system. Quite simply, writing binary numbers gets extremely tedious very quickly and is highly prone to human errors.

To see this in all of its stark reality, consider the binary equivalent of the decimal value:

2,098,236,812

In binary, this number would be written as:

11111010001000010001101100011100

Now, binary numbers are particularly easy to convert to decimal by this process because the number is either 1 or 0. This makes the multiplication easy for those of us who can't remember the times tables because our PDA's have allowed significant portions of our cerebral cortex to atrophy.

Since there seems to be a connection between the bases 2, 8 and 16, then it is reasonable to assume that converting numbers between the three bases would be easier than converting to decimal, since base 10 is not a natural power of base 2. To see how we convert from binary to octal consider figure 1.17.

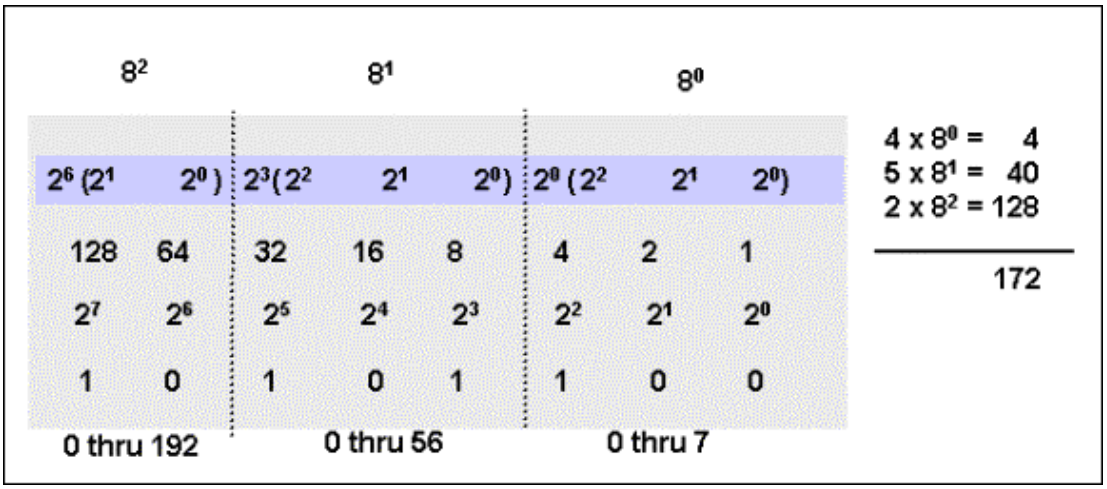


Figure 1.17: Translating a binary number into an octal number. By factoring out the value of the base, we can combine the binary number into groups of three and write down the octal number by inspection.

Figure 1.17 takes the example of figure 1.16 one step further. Figure 1.17 starts with the same binary number, 10101100, or 172 in base 10. However, simple arithmetic shows us that we can factor out various powers of 2 that happen to also be powers of 8. Consider the purple highlighted stripe in figure 1.17. We can make the following simplifications.

Since any number to the 0 power = 1,

$(2^2\ 2^1\ 2^0) = 2^0 \times (2^2\ 2^1\ 2^0)$

Thus,

$2^0 = 8^0 = 1$

We can perform the same simplification with the next group of three binary numbers:

$(2^5\ 2^4\ 2^3) = 2^3 \times (2^2\ 2^1\ 2^0)$

since 2^3 is a common factor of the group.

However, $2^3 = 8^1$, which is the column weight of the next column in the octal number system.

If we repeat this exercise one more time with the final group of two numbers, we see that:

$(2^7\ 2^6) = 2^6 \times (2^1\ 2^0)$

since 2^6 is a common factor of the group. Again, $2^6 = 8^2$, which is just the column weight of the next column in the octal number system.

Since there is this natural relationship between base 8 and base 2, it is very easy to translate numbers between the bases. Each group of three binary numbers, starting from the right side (least significant digit) can be translated to an octal digit from 0 to 7 by simply looking at the binary value and writing down the equivalent octal value. In figure 1.17, the rightmost group of three binary numbers is 100. Referring to the column weights this is just 1*(1*4 + 0*2 + 0*1), or 4.

The middle group of three gives us 8* (1*4 + 0* 2 + 1*1), or 8*5 (40). The two remaining numbers gives us 64*(1* 2 + 0* 1) or 128. Thus, 4 + 40 + 128 = 172, our binary number from figure 1.16. But where's the octal number? Simple, each group of 3 binary numbers gave us the column value of the octal digit, so our binary number is 254. Therefore,

10101100 in binary is equal to 254 in octal, which equals 172 in decimal.

Neat! Thus, we can convert between binary and octal as follows:

- If the number is in octal, write each octal digit in terms of 3 binary digits. For example:
  256773 = 10 101 110 111 111 011
- If the number is in binary, then gather the binary digits into groups of three, starting from the least significant digit and write down the octal (0 through 7) equivalent. For example:
  110001010100110111_2 = 110 001 010 100 110 111 = 612467_8
- If the most significant grouping of binary digits has only 1 or 2 digits remaining, just pad the group with 0's to complete a group of three for the most significant octal digit.

Today, octal is not as commonly used as it once was, but you will still see it used occasionally. For example, in the UNIX (Linux) command ***chmod 777***, the number 777 is the octal representation of the individual bits that define file status. The command changes the file permissions for the users who may then have access to the file.

We can now extend our discussion of the relationship between binary numbers and octal numbers to consider the relationship between binary and hexadecimal. Hexadecimal (hex) numbers are converted to and from binary in exactly the same way as we did with octal numbers, except now we use 2^4 as the common factor rather than 2^3. Referring to figure 1.18, we see the same process for hex numbers as we used for octal.

In this case, we factor out the common power of the base, 2^4, and we're left with a repeating group of binary numbers with column values represented as

2^3 2^2 2^1 2^0

It is simple to see that the binary number 1111 = 15_{10}, so groups of four binary digits may be used to represent a number between 0 and 15 in decimal, or 0 through F in hex. Referring

back to figure 1.16, now that we know how to do the conversion, what's the hex equivalent of 10101100? Referring to the leftmost group of 4 binary digits, 1010, this is just 8 + 0 + 2 + 0, or A. The rightmost group, 1100 equals 8 + 4 + 0 + 0, or C. Therefore, our number in hex is AC.

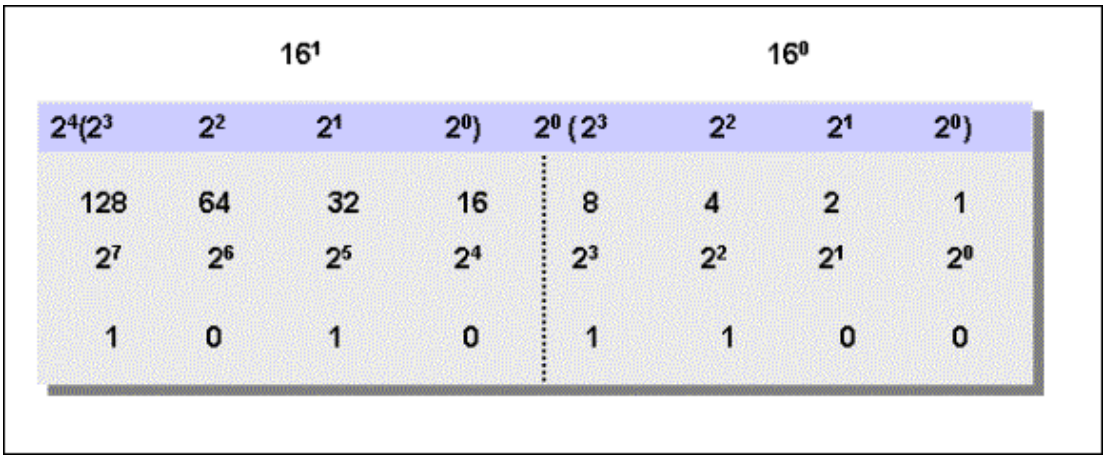


Figure 1.18: Converting a binary number to base 16 (hexadecimal). Binary umbers are grouped by four, starting from the least significant digit, and the hexadecimal equivalent is written down by inspection.

Let's do a 16-bit binary number conversion example.

Binary number:	0101111111010111
Octal:	0 101 111 111 010 111 = 057727 (grouped by threes)
Hex:	0101 1111 1101 0111 = 5FD7 (grouped by fours)
Decimal:	To convert to decimal, see below:

Octal to decimal	Hex to Decimal
$7 \times 8^0 = 7$	$7 \times 16^0 = 7$
$2 \times 8^1 = 16$	$13 \times 16^1 = 208$
$7 \times 8^2 = 448$	$15 \times 16^2 = 3840$
$7 \times 8^3 = 3584$	$5 \times 16^3 = 20480$
$5 \times 8^4 = 20480$	
24,535	24,535

Definitions

Before we go full circle and consider the reverse process, converting from decimal to hex, octal and binary, we should define some terms. These terms are particular to computer numbers and give us a shorthand way to represent the size of the numbers that we'll be working with later on. By size, we don't mean the magnitude of the number, we actually mean the number of binary bits that the number refers to. You are already familiar with this concept because most compilers require that you declare the type of a variable before you can use it. Declaring the type really means two things:

1. How much storage space will this variable occupy, and
2. What type of assembly language algorithms must be generated to manipulate this number?

The table below summarizes the various groupings of binary bits and defines them.

bit	The simplest binary number is 1 digit long
nibble	A number comprised of four binary bits. A NIBBLE is also one hexadecimal digit
byte	Eight binary bits taken together form a byte. A byte is the fundamental unit of measuring storage capacity in computer memories and disks. The byte is also equal to a ***char*** in C and C++.
word	A word is 16 binary bits in length. It is also 4 hex digits in length or 2 bytes in length. This will become more important when we discuss memory organization. In C or C++, a word is sometimes called a ***short***.
long word	Also called a LONG, the long word is 32 binary bits or 8 hex digits. Today, this is an ***int*** in C or C++.
double word	Also called DOUBLE, the double word is 64 binary bits in length, or 16 hex digits.

From the table you may get a clue as to why the octal number representation has been mostly supplanted by hex numbers. Since octal is formed by groups of three, we are usually left with those pesky remainders to deal with. We always seem to have an extra 1, 2, or 3 as the most significant octal digit. If the computer designers had settled on 15 and 33 bits for the bus widths instead of 16 and 32 bits, perhaps octal would still be alive and kicking. Also, hexadecimal representation is a far more compact way of representing numbers, so it has become today's standard. Figure 1.19 summarizes the various sizes of data elements as they stand today and as they'll probably be in the near future.

Today we already have computers that can manipulate 64-bit numbers in one operation. The Athlon64® from Advanced Micro Devices Corporation is one such processor. Another example is the processor in the Nintendo N64®. Also, if you consider yourself a ***PC Gamer***, and you like to play fast action video games on your PC, then you likely have a high-

performance video card in your game machine. It is likely that your video card has a video processing computer chip on it that can process 128 bits at a time. Is a 256-bit processor far behind?

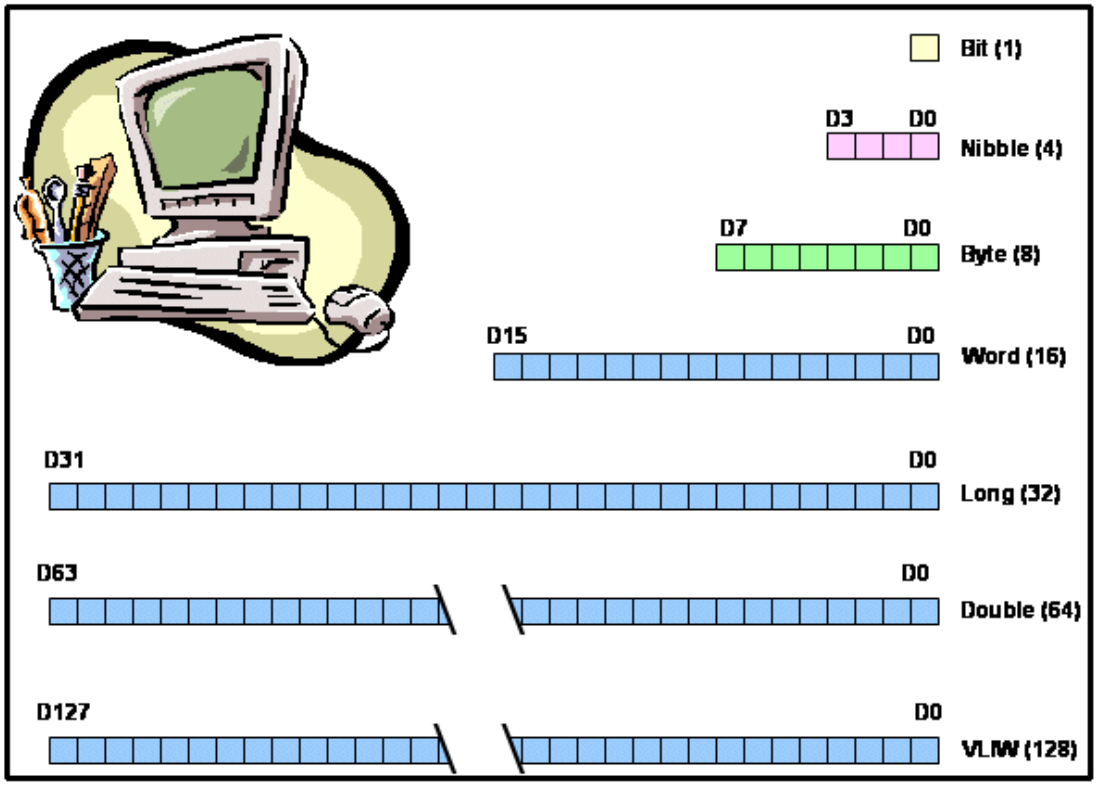


Figure 1.19: Size of the various data elements in a computer system

Fractional Numbers

We deal with fractions in the same manner as we deal with whole numbers. For example, consider figure 1.20.

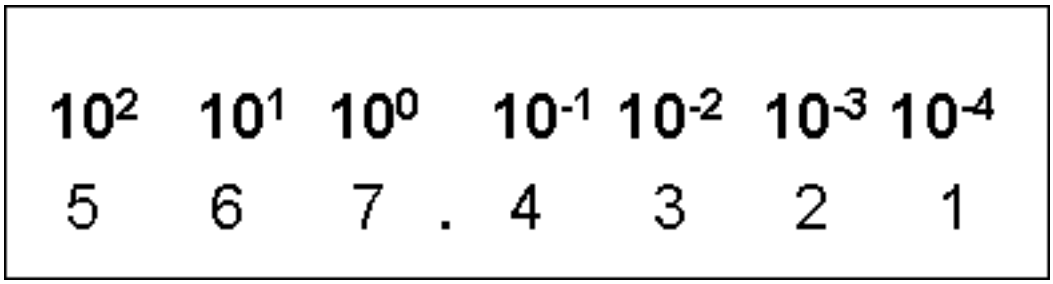


Figure 1.20: Representing a fractional number in base 10

We see that for a decimal number, the columns to the right of the decimal point go in increasingly negative powers of ten. We would apply the same methods that we just learned for converting between bases to fractional numbers. However, having said that, it should be mentioned that fractional numbers are not usually represented this way in a computer. Any fractional number is immediately converted to a floating point number, or ***float.*** The floating-point numbers have their own representation, typically as a 64-bit value consisting of a mantissa and an exponent. We will discuss floating point numbers in a later chapter.

Binary Coded Decimal

In the early days of computers, when there was a transition taking place from instrumentation based strictly on digital logic to instrumentation based on embedded microprocessors, it was convenient to represent numbers in a form called ***binary coded decimal***, or BCD, because standard digital logic circuits were also available to support counting and displaying numbers in the BCD format. A BCD number was represented as 4 binary digits, just like a hex number, except the highest number in the sequence is 9, rather than F. Devices like counters and meters used BCD because it was a convenient way to connect a digital value to some sort of a display device, like a 7-segment display. Figure 1.21 shows the digits of a seven-segment display.

The seven-segment display consists of 7 bars and usually also contains a decimal point and each of the elements is illuminated by a light emitting diode (LED). Figure 1.21 shows how the 7-segment display can be used to show the numbers 0 through 9. In fact, with a little creativity, it can also show the hexadecimal numbers A through F.

BCD was an easy way to convert digital counters and voltmeters to an easy to read display. Imagine what your reaction would be if the Radio Shack® voltmeter read 7A volts, instead of 122 volts. Figure 1.21 shows what happens when we count in BCD. The numbers that are displayed make sense to us because they look like decimal digits. When the count reaches 1001 (9), the next increment causes the display to roll around to 0 and carry a 1, instead of displaying A. Many microprocessors today still contain vestiges of the transition period from BCD to hex numbers by containing special instructions, such as ***decimal add adjust***, that are used to create algorithms for implementing BCD arithmetic.

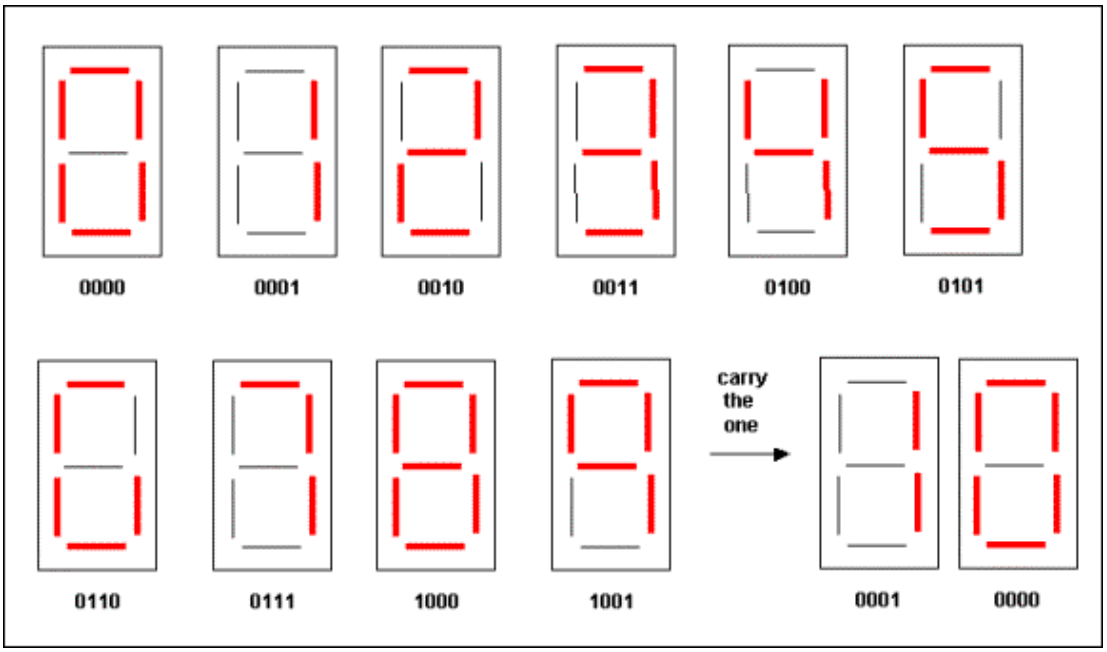


Figure 1.21: Binary coded decimal (BCD) number representation. When the number exceeds the count of 9, a carry operation takes place to the next most significant decade and the binary digits roll around to zero.

Converting Decimals to Bases

Converting a decimal number to binary, octal or hex is a bit more involved because base 10 does not have a natural relationship to any of the other bases. However, it is a rather straightforward process to describe an algorithm to use to translate a number in decimal to another base. For example, let's convert the decimal number 38,070 to hexadecimal.

1. Find the largest value of the base (in this case 16), raised to an integer power that is still less than the number that you are trying to convert. In order to do the conversion, we'll need to refer to the table of powers of 16, shown below. From the table we see that the number 38,070 is greater than 4,096 but less than 65,536. Therefore, we know that the largest column value for our conversion is 16^3.

$16^0 = 1$	$16^1 = 16$	$16^2 = 256$	$16^3 = 4096$
$16^4 = 65{,}536$	$16^5 = 1{,}048{,}576$	$16^6 = 16{,}777{,}216$	$16^7 = 268{,}435{,}456$

2. Perform an integer division on the number to convert:
    a. 38,070 DIV 4096 = 9
    b. 38,070 MOD 4096 = 1206
3. The most significant hex digit is 9. Repeat step 1 with the MOD (remainder) from step 2. 256 is less than 1206 and 4096 is greater than 1206.
    a. 1206 DIV 256 = 4
    b. 1206 MOD 256=182

4. The next most significant digit is 4. Repeat step 2 with the MOD from step 3. 182 is greater than 16 but less than 256.
    a. 182 DIV 16 = 11 (B)
    b. 4b 182 MOD 16 = 6
5. The third most significant digit is B. We can stop here because the least significant digit is, by inspection, 6. Therefore: $38{,}070_{10} = 94B6_{16}$

Before we move on to the next topic, logic gates, it is worthwhile to summarize why we did what we did. It wouldn't take you very long, writing 32-bit numbers down in binary, to realize that there has to be a better way. There is a better way, hex and octal. Hexadecimal and octal numbers, because of their bases, 16 and 8 respectively, have a natural relationship to binary numbers, base 2. We can simplify our number manipulations by gathering the binary numbers into groups of three or four. Thus, we can write a 32-bit binary value such as 10101010111101011110000010110110 in hex as AAF5E0B6.

However, remember that we are still dealing with binary values. Only the way we choose to represent these numbers is different. As you'll see shortly, this natural relationship between binary and hex also extends to arithmetic operations as well. To prove this to yourself, perform the following hexadecimal addition:

0B + 1A = 25 (Remember, that's 25 in hex, not decimal. 25 in hex is 37 in decimal)

Now convert 0B and 1A to binary and perform the same addition. Remember that in binary 1 + 1 = 0, with a carry of 1.

Since it is very easy to mistake numbers in hex, binary and octal, assemblers and compilers allow you to easily specify the base of a number. In C and C++ we represent a hex number, such as AA55, by 0xAA55. In assembly language, we use the dollar sign. So the same number in assembly language is $AA55. However, assembly language does not have a standard, such as ANSI C, so different assemblers may use different notation. Another common method is to precede the hex number with a zero, if the most significant digit is A through F, and append an "H" to the number. Thus $AA55 could be represented as 0AA55H by a different vendor's assembler.

Engineering Notation

While most students have learned the basics of using scientific notation to represent numbers that are either very small or very large, not everyone has also learned to extend scientific notation somewhat to simplify the expression of many of the common quantities that we deal with in digital systems. Therefore, let's take a very brief detour and cover this topic so that we'll have a common starting point. For those of you who already know this, you may take your bio break about 10 minutes earlier.

Engineering notation is just a shorthand way of representing very large or very small numbers in a format that lends itself to communication simplicity among engineers. Let's start with an example that I remember from a nature program about bats that I saw on TV a

number of years ago. Bats locate insects in absolute blackness by using the echoes from ultrasonic sound waves that they emit. An insect reflects the sound bursts and the bat is able to locate dinner. What I remember is the narrator saying that the nervous system of the bat is so good at echo-location that the bat can discern sound pulses that arrive less than a few millionths of a second apart. Wow!

Anyway, what does a "few millionths" mean? Let's say that a few millionths is 5 millionths. That's 0.000005 seconds. In scientific notation 0.000005 seconds would be written as 5×10^{-6} seconds. In engineering notation it would be written as 5 μs and pronounced 5 microseconds. We use the Greek symbol, μ (mu) to represent the micro portion of microseconds. What are the symbols that we might commonly encounter? The table below lists the common values:

TERA = 10^{12} (T)	**PICO = 10^{-12} (p)**
GIGA = 10^{9} (G)	**NANO = 10^{-9} (n)**
MEGA = 10^{6} (M)	**MICRO = 10^{-6} (μ)**
KILO = 10^{3} (K)	**MILLI = 10^{-3} (m)**
	FEMTO = 10^{-15}(f)

So, how do we change a number in scientific notation to an equivalent one in engineering notation? Here's the recipe:

1. Adjust the mantissa and the exponent so that the exponent is divisible by 3 and the mantissa is not a fraction. Thus, 3.05×10^{4} bytes becomes 30.5×10^{3} and not 0.03×10^{6} .
2. Replace the exponent terms with the appropriate designation. Thus, 30.5×10^{3} bytes becomes 30.5 Kbytes, or 30.5 Kilobytes.

About 99.99% of the time the unit will be within the exponent range of +/- 12. However, as computers get ever faster, we will be measuring times in the fractions of a picosecond, so its appropriate to include the femtosecond on our table. As an exercise, try to calculate how far light will travel in one femtosecond, given that light travels at a rate of about 6 inches per nanosecond on a printed circuit board.

Although we discussed this earlier, it should be mentioned again in this context that we have to be careful when we use the engineering terms for ***kilo, mega and giga.*** That's a problem that computer folk have created by misappropriating standard engineering notations for their

own use. Since $2^{10} = 1024$, computer "Geekspeakers" decided that it was just too close to 1000 to let it go, so they overloaded the K, M and G symbols to mean 1024, 1048576 and 1073741824, respectively, rather than 1000, 1000000 or 1000000000, respectively.

Fortunately, we rarely get mixed up because the computer definition is usually confined to measurements involving memory size, or byte capacities. Any time that we measure anything else, such as clock speed or time, we use the conventional meaning of the units.

Summary of Chapter One

- The growth of the modern digital computer progressed rapidly, driven by the improvements made in the manufacturing of microprocessors made from integrated circuits.
- The speed of computer memory has an inverse relationship to its capacity. The faster a memory, the closer it is to the computer core.
- Modern computers are based upon two basic designs: CISC and RISC.
- Since an electronic circuit can switch on and off very quickly, we can use the binary numbers system, or base 2, as the natural number system for our computer.
- Binary, octal and hexadecimal are the natural number bases of computers and there are simple ways to convert between them and decimal.

Bibliography and References for Chapter 1

1- Carver Mead, Lynn Conway, ***Introduction to VLSI Sysyems***, ISBN 0-201-04358-0, Addison-Wesley, Reading, MA, 1980
2- For an excellent, and highly readable description of the creation of a new super minicomputer see ***The Soul of a New Machine*** by Tracy Kidder, ISBN-0-316-49170-5, Little, Brown and Company, 1981
3- Daniel Mann, Private Communication
4- http://www.seagate.com/ww/v/index.jsp%3Fvgnextoid%3D8e050debd6f78110VgnVCM100000f5ee0a0aRCRD
5- David A. Patterson and John L. Hennessy, ***Computer Organization and Design: Second Edition,*** ISBN 1-55860-428-6, Morgan-Kaufmann Publishers, San Francisco, Pg. 30
6- Daniel Mann, ibid
7- www.specbench.org/cgi-bin/osgresults?conf=cint95

Exercises for Chapter 1

1- Define ***Moore's Law.*** What is the implication of Moore's Law in understanding trends in computer performance? Limit your answer to no more than two paragraphs.

2- Suppose that in January, 2004, AMD announces a new microprocessor with 100 million transistors. According to Moore's Law, when will AMD introduce a microprocessor with 200 million transistors?

3- Describe an advantage and a disadvantage of the organization of a computer around abstraction layers.

4- What are the industry standard busses in a typical PC?

5- Suppose that the average memory access time is 35 nanoseconds (ns) and the average access time for a hard disk drive is 12 milliseconds (ms). How much faster is semiconductor memory than the memory on the hard drive?

6- What is the decimal number 357 in base 9?

7- Convert the decimal number 68,456 to base 13. Note that the numbers in a base 13 system would be 0 through D.

8- Convert the decimal number 73,503 to base 17. Note that the numbers in a base 17 system would be 0, 1, 2, 3..... D, E, F, G.

9- Convert the following hexadecimal numbers to decimal.

(a) 0xFE57 (b) 0xA3011 (c) 0xDE01 (d) 0x3AB2

10- Convert the following decimal numbers to binary.

(a) 510 (b) 64,200 (c) 4,001 (d) 255

11- Convert the following numbers to HEX

(a) $011001100001101_{\mathbf{2}}$ (b) $375667_{\mathbf{8}}$ (c) $4{,}095_{\mathbf{10}}$ (d) $253{,}879_{\mathbf{10}}$

12- Convert the following decimal numbers to hexadecimal. Show your work! I have an HP calculator that does it too! The calculator doesn't teach number systems, it just calculates.

a- 549 b- 4099 c- 1,400,654 d- 15,789,555 e- 2,105,409,071,090

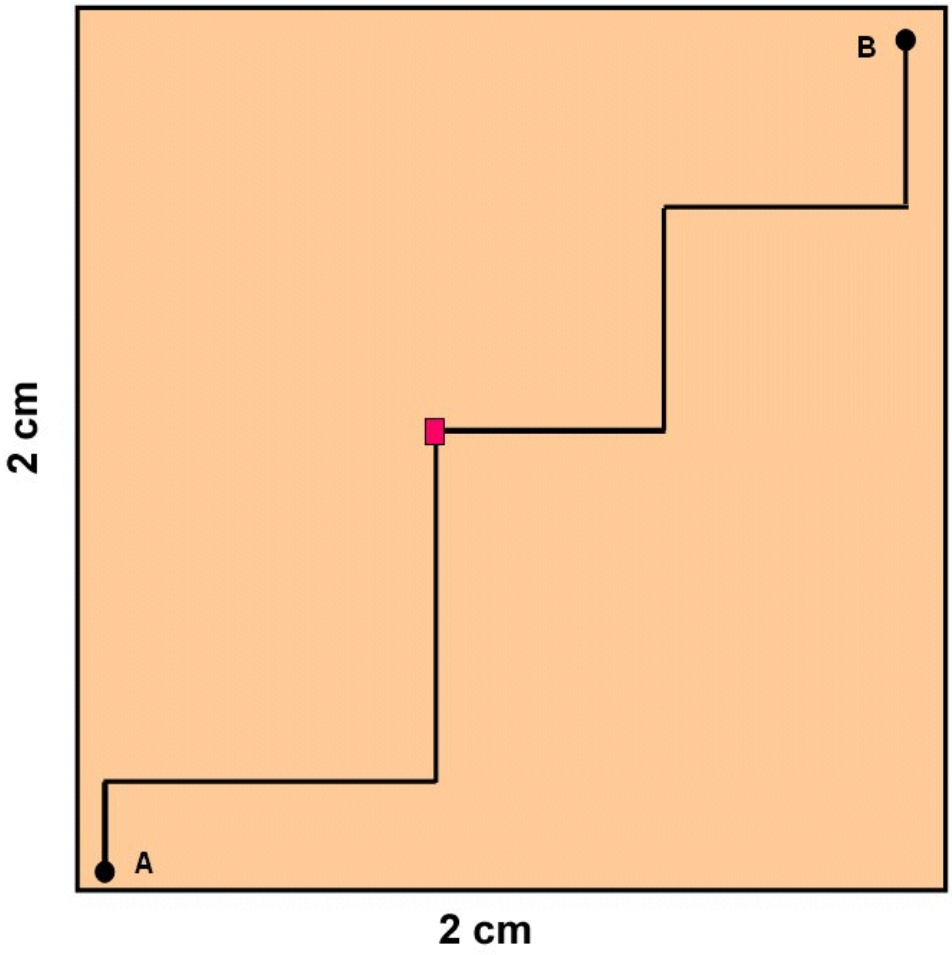
13- Convert the following binary numbers to decimal

a- 10101010 b- 111010000011 c- 11000001011111011 d- 0101101011110011
e- 00110000111110100011010101111100

14- Suppose that you were traveling at 14 furlongs per fortnight. How fast are you going in feet per second? Express your answer in engineering notation.

15- One of the most important considerations in the design of an integrated circuit is the problem of the speed of light. The speed of light in free space is approximately 3×10^{10} cm per second. However, the speed of light on an integrated circuit chip is only 1/2 that of free space. Because of this universal law, we have to be very careful about the time delay (propagation delay) between various locations on the chip.

The path from point A to point B on the figure shown below represents the longest signal path on the chip. The red square represents a booster amplifier which introduces an additional 8 picosecond propagation delay.



What is the time delay for a signal traveling along the path from A to B? Express your answer in engineering units.

Hint: The simplest way to get an approximate value for the path lengths is to print the above diagram and then use a ruler to scale the lengths from the dimensions that you do know. Also, some drawing programs, such as Paint Shop Pro© give you the coordinates of the cursor position.

16- Consider the simplified diagram of an integrated circuit shown below. The path from point A to point B on the figure represents one typical path on the chip for the clock signal. The clock must be able to arrive at all points of the chip at roughly the same time, otherwise, a problem called "clock skew", or time differences between the synchronization signals may arise. The red squares represents a booster amplifier which introduces an additional 8 picosecond propagation delay and the black dots represents a point where the signal path takes a branch and divides into multiple signals. This design is called a "clock fan-out network".

What is the time delay for a signal traveling along the path from A to B? Express your answer in engineering units.