

Measurement and Units of Measure

Introduction

Computer Science deals with three overlapping areas of interest: hardware, software, and theory. Computer Architecture is a subject within computer science that deals with hardware, which means (among other things) that you need to be comfortable with measuring the physical properties of various hardware devices and systems. The purpose of this document is to review some key topics that you probably learned long ago, but may have forgotten.

Units of Measure and Orders of Magnitude

We always measure the physical properties of a thing using particular units. For example, the unit for measuring your body's mass is the *pound* and the unit for measuring your height is *inches*. But, of course, it's not as simple as that. If you use the metric system you measure your mass in kilograms, not pounds, and you measure your height in centimeters. And even if you measure your height in inches, you are probably mix two different units of measure: *feet* and *inches*.

Although we can measure the mass and lengths of the devices and we deal with in computer architecture, we are typically more interested in a third property: *time* (measurement unit: the *second*) and the related properties of *period* and its reciprocal, *frequency*. Furthermore, computers manipulate physical representations of *information*, which has its own measurement unit, the *bit*.

The first issue to deal with is the enormous range of values that occur in the physical and information worlds. To deal with this range, it is convenient to use exponential notation to represent values. We will use the term "order of magnitude" to refer to the base-ten exponent used to represent a value in scientific notation when the integer part of the value is between 1 and 999. For example, the number 123 can be expressed as 1.23×10^2 (or 1.23E2), and we will say that its order of magnitude is 2.

Actually, the term "order of magnitude" is most often used to express the approximate relative sizes of two values. Divide the larger value by the smaller value, convert the quotient to normalized scientific notation, and the exponent can be used to say how many orders of magnitude the larger value is compared to the smaller value. For example $5,280 \div 45$ is $117.3 = 1.173E2$, so "5,280 is two orders of magnitude larger than 45."

In the physical world of length, mass, and time, prefixes are used to indicate multiplication of the basic units (meter, gram, second) by powers of ten. There are prefixes for every three orders of magnitude, as well as several intermediate ones. The multiplication prefixes commonly encountered in computer architecture are:

Multiplier	Power of Ten	Prefix (abbreviation)	Common Name
0.000 000 000 001	10^{-12}	pico (p)	Trillionth
0.000 000 001	10^{-9}	nano (n)	Billionth
0.000 001	10^{-6}	micro (μ)	Millionth
0.001	10^{-3}	milli (m)	Thousandth
1,000	10^{+3}	kilo (K)	Thousand
1,000,000	10^{+6}	mega (M)	Million
1,000,000,000	10^{+9}	giga (G)	Billion
1,000,000,000,000	10^{+12}	tera (T)	Trillion
1,000,000,000,000,000	10^{+15}	peta (P)	Quadrillion

In computer architecture, small numbers are most often associated with *time* (seconds): how many milliseconds (msec) it takes a spinning disk to make one rotation, how many picoseconds (psec) it takes for a gate to change state, etc. Big numbers are most often associated with information storage capacities (how many gigabytes (GB) of main memory or terabytes (TB) of disk a computer has) or the reciprocal of time (how fast is a clock, in GHz). The remainder of this document tries to put all this terminology in order.

Time (period, frequency)

The basic unit of time is the *second*, and we use prefixes to denote fractions of a second. So one second contains 1,000 milliseconds (msec), 1,000,000 microseconds (μ sec), etc. Remember, the prefix names represent three orders of magnitude differences, so there are a thousand picoseconds in a nanosecond, a thousand nanoseconds in a microsecond, a thousand microseconds in a millisecond, and a thousand milliseconds in a second.

Skill: represent the same value using different prefixes.

The same value can be represented in a number of ways. For example, 1 nsec is the same as 1,000 psec as well as 0.001 μ sec. Although it is not a hard and fast rule, the “normal” way to represent a value is to scale it so that the integer part is a number between one and 999. The conversion factor for adjacent prefixes is either 1,000 or 0.001, depending on which way you are going. The trick is not to go the wrong way! If you know the conversion factor between feet and inches is 12, that only applies to converting feet into inches: 5×12 gives

60". To go the other way you have to use 1/12 as the conversion factor, which is the same as dividing by 12 rather than multiplying: $60'' \div 12$ gives 5'. Feet and inches are everyday concepts for us, so we are not likely to make silly mistakes, like multiplying inches by 12 to get feet. (Is 60 inches equal to 720 feet?—no way!) But not so with scientific prefixes: you have to think carefully when converting something like nanoseconds to microseconds to make sure you don't end up with picoseconds by mistake. Just remember to multiply when going to smaller units and to divide when going to larger units.

- Convert 1.250 nsec to psec: Multiply, giving 1,250 psec.
- Convert 1.250 nsec to μ sec: Divide, giving 0.001250 μ sec.
- How many picoseconds in 3.7 μ sec? Multiply by 10^{+6} , giving 3,700,000 psec.

Use seconds to measure *latency*—how long it takes something to happen. Examples are how long it takes a gate to change state; how long it takes to execute an instruction (or a program); how long it takes to get a copy of some information that is in main memory or on disk, etc.

Note that multiples of a second are normally given using conventional units (minutes, hours, days, years, centuries, millennia, etc.) rather than the scientific prefixes. But you can always impress your friends by reminding them that there are 86.4 Ksec in a day.

Time is also intimately related to various *rate* measures—see below.

Information

Computers are all about manipulating digital information, represented as *binary digits*; John Tukey of Bell Laboratories coined the name *bit* to mean one binary digit in 1946. But the digital age owes much of its binary underpinnings to another person working at Bell Labs in the 1940's: Claude Shannon, who proposed the bit as the unit for measuring information. Because Bell Labs was part of the telephone system, Shannon addressed the issue of how to measure the information capacity of a communication channel, such as a telephone line. Simply stated, Shannon defined the bit as the amount of uncertainty that is reduced by answering one yes/no question. For example, if I do not know whether it is raining outside and I look out the window, my uncertainty is reduced by one bit— I now know whether it is actually raining or not. Of course, I will undoubtedly obtain other information by looking out the window, but my uncertainty about whether or not it is raining has been reduced by one bit.

Key to understanding information measurement is the notion of the number of possibilities our uncertainty covers. If I ask you to pick a number between 1 and 100, my uncertainty spans 100 possibilities. The number of bits of uncertainty, however, is just $\log_2(100)$, because I can determine which number you picked by getting the answer to that many yes/no questions. My strategy is to use each question to eliminate half of the possible outcomes. Rather than ask, "Is it 1? ... Is it 2? ..." until you say "yes," I might start by asking, "Is it an odd number?" Regardless of your answer, I will have eliminated half of the possible outcomes and reduced my uncertainty by one bit. Since $\log_2(100)$ is approximately 6.644, my "binary search" strategy should reduce my uncertainty after

just 6-7 questions rather than the 50 it would take on average if I just asked about each possible number individually.

The number 6.644 in this example raises some important points. Obviously, you can't ask a fraction of a yes/no question; either you ask the question or you don't, and if you do ask it, the answer has to be either "yes" or "no." First, let's look at a way in which the fractional number of bits actually does make sense. Then we'll look at how information is stored in a computer, where fractional bits don't exist any more than fractional yes/no questions.

To understand fractional bits of information, think in terms of the *average* number of yes/no questions you would have to ask, assuming that you get to play the guessing game lots of times. To start, consider these two sequences of questions and answers:

Sequence 1		Sequence 2	
Question	Answer	Question	Answer
Bigger than 50?	No	Bigger than 50?	No
Bigger than 25?	No	Bigger than 25?	No
Bigger than 12?	No	Bigger than 12?	No
Bigger than 6?	No	Bigger than 6?	No
Bigger than 3?	No	Bigger than 3?	No
Bigger than 2?	No	Bigger than 2?	Yes

After the six questions in Sequence 1, we still don't know what the number is that we are searching for: it could be either 1 or 2, and we would have to ask a seventh question to find out which it is. But in the second sequence we got a different answer to the sixth question and we immediately know that the answer is 3, which is the only number bigger than 2 that is not bigger than 3. The point is that, depending on the number you are trying to guess and on how you choose to divide the uncertainty in half, sometimes it will take six questions and sometimes it will take seven. You could try playing the game lots of times with a patient friend (or a computer programmed to act like a patient friend), and count how often it takes six questions and how often it takes seven. OK, I just played the game 1,000 times and found that I got the secret number after six guesses 356 times, but the rest of the time it took seven guesses. What is the average number of guesses? It is a *weighted average*. The two numbers being averaged are six and seven, but instead of just adding them together and dividing by two, you multiply the value six by a weight of 356 and multiply seven by the weight (1,000 - 356), then you add the two products and divide by the number of cases (1,000). That's $((356 \times 6) + (644 \times 7)) \div 1,000 = 6.644 = \log_2(100)$. Of course if this the game is played truly randomly, it won't always come out exactly 6.644, but if you play it enough times (a) you will get very bored and/or lose all your friends and (b) will come closer and closer to the actual fractional number of bits.

Weighted averages show up a lot in this course. A shortcut (sometimes) is to adjust all the weights so they add up to 1.0, eliminating the need for division at the end. In the example, divide each weight by the sum of the weights (giving 0.356 and 0.644), then just multiply and add. There is an example below.

The implication of information theory is profound: *any kind of information can be represented using binary digits*. The rules for mapping particular sets of binary digits to particular members of a set of choices is called *encoding*, a major topic of its own. Figuring out how to divide information into sets of discrete values that can be encoded is called *quantization*, another big topic.

A key point about encoding is that, just as there is no such thing as part of a yes/no question, there is no such thing as a part of a bit in computer hardware. To return to our pick-a-number example, the choices would have to be encoded using an integer number of bits. In this case, it would be $\lceil \log_2(100) \rceil = 7$ bits. If you knew that some of the possible choices occur more often than others, you could make up a code that uses fewer bits for those choices and more bits for the less-frequent choices, reducing the average number of bits needed to encode the different choices to the 6.644 lower limit. But for our purposes, we will concentrate on codes that use a fixed number of bits for each possible choice.

Some examples of information codes:

- Characters: ASCII (7 bits per character), ISO-Latin-1 (8 bits per character), Unicode (16 bits per character), representing 128, 256, or 65,536 different characters respectively.
- Pixel colors: most computer monitors use 24 bits to represent the color of a pixel. The 24 bits are divided into three sets of 8 bits representing the intensities of red, green, and blue “channels.” (The colors red, green, and blue is an example of a set of *primary colors*: they can be mixed in different proportions to produce all other colors.) Some digital cameras use 12 bits per channel, and Photoshop can work with images having 16 bits per channel, so a pixel can have 2^{24} , 2^{36} , or 2^{48} different colors depending on the device or software being used.
- CD audio: two stereo audio channels, each with 12 bits representing the sound pressure level of the sound at a particular moment in time (repeated 44,000 times per second).
- Numbers: fixed-point (unsigned, or signed using codes two’s complement, sign–magnitude, or biased codes—the (signed) *int* data type in C, C++, C#, and Java uses two’s complement fixed-point encoding); floating-point (IEEE-754 standard encoding normally used for *float* and *double* data types).

Logarithms and Exponents

A little review of logarithms and exponents is in order. “A logarithm is an exponent” is another way of saying that $\log_{base}(x) = y$ is the same as $base^y = x$. For our example, $\log_2(100) = 6.644$ is the same as saying $2^{6.644} = 100$.

When multiplying or dividing numbers using exponential notation with the same base, add or subtract exponents respectively: $1.2 \times 10^3 \times 2.0 \times 10^1 = 2.4 \times 10^4$; $10^9 \div 10^3 = 10^6$; $2^{30} \div 2^{20} = 2^{10}$.

Memorize the first eleven powers of two:

N	0	1	2	3	4	5	6	7	8	9	10
2^N	1	2	4	8	16	32	64	128	256	512	1024

Note that 2^{10} (1,024) is approximately equal to 10^3 (1,000). This means that $2^{10} \times 2^{10} = 2^{20}$ is approximately equal to $10^3 \times 10^3 = 10^6$, 2^{30} is approximately equal to 10^9 , etc. The convention is to use scientific prefixes K, M, G, T, and P (kilo, mega, giga, tera, and peta) for binary numbers with exponents of 10, 20, 30, 40, and 50 respectively.

Finally, just as we have conventional names for time periods longer than a second (minutes, hours, etc.), there is a conventional name for $2^3 = 8$ bits: a *byte*. Lower- and uppercase letter are used for abbreviations: 1 B is 8 b.

Example: How many bits are there in 2 MB? **Solution:** $2 \text{ MB} = 2^1 \times 2^{20} \times 2^3 = 2^{24}$ bits, which is the same as $2^4 \times 2^{20} = 16 \times 2^{20}$, or 16 Mb.

Rates (speed, bandwidth)

We mentioned earlier that *latency* tells how long something takes, and that it is measured in *seconds*. As you can imagine, the related concept of *speed* is important for evaluating the performance of computer systems. Speed is always a *ratio* of something divided by a unit of time. The terms *rate* and *speed* are equivalent. For example:

- MPH, the speed of a car, bicycle, airplane, etc. Miles divided by hours.
- RPM, the speed at which something, like a car's crankshaft or the platters of a disk drive rotate. Revolutions divided by minutes.

In computer architecture there are some special rates that you need to know about. One is *clock speed* or *frequency*. Logic circuits are synchronized using an electronic signal called a *clock*, which is a wire that alternates at a fixed rate between whatever two voltages are used to represent boolean 1 and boolean 0. If the voltage on a wire goes from one value to another and back again, it is called a *pulse*, and the speed of a clock is measured in pulses per second (pulses divided by seconds). The unit of measure for pulses per second is the Hertz, abbreviated Hz. A clock that operates at 1,000 pulses per second has a frequency of 1 KHz. Clock speeds of processors typically range from 1-4 GHz.

The reciprocal of a clock's frequency (pulses per second) is its *period* (seconds per pulse), which tells the latency from the start of one pulse to the start of the next one. Frequencies are always measured in Hertz, and periods are always measured in seconds. The period of a 1 GHz clock is 1 nsec. Because they are reciprocals, increasing one always decreases the other: doubling 1 GHz to 2 GHz clock halves the period from 1 nsec to 0.5 nsec (500 psec).

A key concept in computer architecture is that the maximum latency with which gates can change state is what determines the minimum clock period of a system, and hence the maximum clock speed that can be used. Hopefully, that concept will make intuitive sense to you by the time you complete the first part of the course!

A second important rate measure for computing is *bandwidth*, which is the rate at which bits can be transferred from one place to another, measured in bits per second. When you buy a “broadband” connection to the internet from a cable or telephone company, how much you pay depends in large part on how much bandwidth the company promises to provide you. An example from computer architecture is copying information from a disk to main memory. The latency tells how long it takes to access the first bit of information on the disk, but how long it takes to transfer the information after the initial latency depends on the bandwidth of the communication channel between the disk and memory.

Example: How long does it take to transfer a 100 MB program from disk to memory if the initial latency is 10 msec and the bandwidth of the channel between disk and memory is 500 Mbps? **Solution:** Start by getting all the units to line up: 100 MB is 800 Mb because there are 8 bits (b) in a byte (B). The total transfer time will be the initial latency plus the time it takes to transfer the program over the channel. The initial latency is given as 0.010 sec. You need to calculate the number of seconds to transfer 800 Mb over a 500 Mb/sec (500 Mbps) channel, which is $800/500 = 1.6$ sec. So the total time is $1.6 + 0.01 = 1.61$ sec.

Keeping track of units takes practice. In the example, we had to combine the size of the program (unit: bits) and the bandwidth of the channel (unit: b/s) to give the time for the transfer (unit: sec). The correct operation was to divide: $\text{bits} \div (\text{bits} \div \text{sec})$ is the same as $\text{bits} \times (\text{sec} \div \text{bit})$, which means the “bits” units cancel out, leaving an answer measured in seconds.

The third important rate measure for computer architecture is the speed at which programs get executed. The unit of measure here is programs per second, although the raw data we work with is usually the reciprocal: the time it takes to execute a program, seconds per program. The direct measure of execution speed is normally called *throughput*, with jobs per hour being one unit traditionally used. Not to telegraph the textbook too much, the actual formula for execution time is:

$$s/p = i/p \times c/i \times s/c$$

Here, s/p is the seconds per program, i/p is the instructions per program, c/i is the (average) clock pulses per instruction, and s/c is the period of the clock. You can see that the units on the right side of the equation cancel out to give the units on the left, notably that the number of seconds per clock pulse times the number of clock pulses gives the number of seconds.

Example: How long does it take to execute a million instructions on a processor with a 2 GHz clock and an average CPI of 1.5? (CPI is the standard abbreviation for c/i in the above equation). **Solution:** i/p is 1×10^6 , c/i is 1.5, and

s/c is $1 \div (2 \times 10^9)$. Remember, the units for the clock speed is c/s and we need s/c for the equation. $1 \div (2 \times 10^9)$ is 0.5×10^{-9} , so the answer is $1 \times 10^6 \times 1.5 \times 0.5 \times 10^{-9} = 0.75 \times 10^{-3} = 0.75 \text{ msec} = 750 \mu\text{sec}$.

Weighted Averages (CPI)

We mentioned earlier that weighted averages show up a lot in this course. Here is an example problem for computing clocks per instruction:

Example: A computer takes 3 clock cycles to execute memory access instructions, 1 clock cycle to execute branch instructions, and 2 clock cycles to execute all others types of instructions. A program consists of 20% memory access instructions and 10% branch instructions. What is the average CPI?
Answer: $0.2 \times 3 + 0.1 \times 1 + 0.7 \times 2 = 2.1$ clocks per instruction. Note that the sum of the percentages has to add up to 100%, and that converting percentages to proportions makes the sum of the weights 1.0, so the weighted average is easy to compute.

Comparisons (ratios, percentages)

Numbers measure properties of things in different ways. In 1946, a psychophysicist (a person who measures the relationship between physical properties of stimuli and our responses to them) named S. S. Stevens introduced the principle that there are four scales of measurement: *nominal*, *ordinal*, *interval*, and *ratio*. Nominal scales can be used only to tell that things have different names from one another, but say nothing about numerical relationships. For example, the “2” in “type 2 diabetes” says nothing about the relationship between that kind of diabetes and other kinds, just that they have different names. Ordinal scales allow you to talk about predecessor and successor functions; what comes before or after something else. Numbering the steps in an algorithm would be an example of an ordinal scale; the numbers tell you the sequence in which to do things, but nothing more. Interval scales let you do addition and subtraction operations, with Celsius and Fahrenheit temperature scales being classic examples: the difference between 30 degrees and 40 degrees is the same as the difference between 60 degrees and 70 degrees. But you never hear the weatherman say something like, “today was twice as hot as yesterday.” It is ratio scales, which support meaningful multiplication and division operations, that allow the strongest types of comparisons between things. When comparing the performance of computer systems, we will always work with ratio scales. We’ll return to the significance of this point below.

As the textbook points out, we often want to compare the performance of one processor to another (which one is faster and by how much), or to compare the performance of a processor after a design change is made to what it was before the change was made (how much *speedup* the change caused). To be meaningful, these comparisons will always be done based on two measured (or calculated) execution times. The issue here is how you talk about such comparisons. If Computer A takes 10 seconds to run a program and Computer B takes 5 seconds to run the same program, which one is faster or slower, and by how much? The problem is to make a meaningful English sentence that compares the two numbers of seconds. Consider some possibilities:

1. Computer A is half as fast as Computer B.
2. Computer B is twice as fast as Computer A.
3. Computer B is ___% faster than Computer A.
4. Computer A is ___% slower than Computer B.
5. Computer B is ___ times faster than Computer A.
6. Computer A is ___ times slower than Computer B.

The first two sentences are unambiguous. The third and fifth are directly computable. It's the fourth and sixth that cause some problems in English.

The correct answer to #3 is "100% faster." Take the ratio of the slower time to the faster time ($10 \div 5 = 2.0$), subtract 1, and multiply by 100 to get "percent faster." If subtracting 1 is hard to remember, just think what it would mean if the execution times were equal: the ratio would be 1.0 but the percentage difference would zero. You have to subtract out the 1.0 you would get if the times were equal before multiplying by 100.

The correct answer to #5 is "2.0 times faster." It's just the ratio of the longer time over the shorter time. Just like #2, "2.0 times faster" is the same as "twice as fast."

For #4, the answer should be "50% slower" and for #6 the answer should be "two times slower." We are comparing speeds, and if one computer has half the speed of the other, saying it is 50% slower is unambiguous: the only issue is that you need two different formulas for percentages depending on whether the answer is "slower" or "faster." (There is no 1.0 to subtract out of the ratio before multiplying by 100 when dealing with slower speeds.)

But there is a language problem with #6: when we multiply (use the word "times" in English) we normally make things bigger (the exception, of course, is when we multiply by a number between 0 and 1.0). So to make sense, we mentally convert "X times slower" into "1/X times as fast," and we might as well have given our answer in one of the forms #1, #2, #3, or #5 in the first place.

For simplicity's sake, avoid expressing speed comparisons using sentences like #4 and #6.

To return to the point about measurement scales mentioned above, the English word "times" definitely means you are talking about a ratio scale. Because percentages are based on ratios, "times" is implied when using percentage to relate two measurements ("X is 50% of Y" means "X is 0.5 times Y.") On the other hand, the English words "less" and "more" are appropriate when using interval scales ("Today is 10 degrees warmer (more warm) than yesterday.") Add 10 to yesterday's temperature to get today's. The language issue comes when you use interval-scale terminology with ratio-scale units, like percentages. You can do it meaningfully, but it introduces an added step for understanding the statement. If X is 50% more (less) than Y, you first have to take 50% of Y by multiplying by 0.5 (which is all right to do because X and Y are presumably measured using a ratio scale), and then you have to perform the interval-scale addition (subtraction) operation to find out what the value of X is. But if you had just said that X is 50% of Y, all you have to do is multiply Y by 0.5 to know what X is. This point is just another way of saying the same thing mentioned in the previous paragraph: if you are

working with ratio scales (and you are when dealing with time and rate), use “ratio-talk” rather than “interval-talk” to keep things simple and clear.