CS-343 Background Material

Dr. Christopher Vickery
Queens College of CUNY

Digital computers use logic circuits to operate on Boolean values, but the real world is made up of physical entities that have properties such as mass, color, pitch, duration, and beauty. This chapter introduces some of the considerations that go into mapping elements of the real world into Boolean values so that logic circuits can operate on them.

Measuring Physical and Digital Quantities

The physical world is full of phenomena that we can measure using various units for such properties as length, mass, and time. There is an international standard system of units, called SI (an abbreviation for the French term, "*Le Système International d'Unites*") for measuring physical units, which you can examine in detail at the web site of the US National Institute of Standards and Technology (NIST) [1]. The SI uses the *mks* system of measurement in which the meter, kilogram, and second are the standard units for measuring length, mass, and time. There are also well-defined units for measuring other aspects of the physical world (but not beauty!).

The range of values that physical measurements can take on is typically extremely large. For example, sound is caused by variations in sound pressure level (compression of the air), and the unit of measure for sound pressure level is the dynes per square centimeter (d/cm^2)[1]. The weakest sound pressure level that humans can hear is approximately 0.0002 d/cm^2 and the sound pressure level that is so intense as to cause pain is 2,000,000,000 d/cm^2, a range of 13 orders of magnitude. The term "order of magnitude" is often used informally to mean "big difference," but here we're use it in its strict meaning of "bigger by a factor of ten." Also, you may already be familiar with the decibel (dB) unit for measuring the loudness of sounds, which uses a logarithmic scale to relate sound intensities. Using decibels, the threshold of hearing (0.0002 d/cm^2) is 0 dB and the threshold of pain is 130 dB. One dB, which corresponds to a ratio of $1:10^{0.1}$, is just about the minimum difference in two sound intensities that humans can discriminate between.

Table 1 lists the prefixes used to identify some of the standard multiples and divisions of various physical units of measure over a range of 30 orders of magnitude. For example, you would write 6.25 μsec for 0.00000625 of a second. (Where the Greek letter μ can't be written, such as in ASCII-formatted text, you will typically see the roman letter 'u' substituted.) Also, the standard unit for weight is the kilogram, not the gram, but the prefixes are applied to grams. So 1 mg is 0.001 gram, and 1 kg is 1,000 grams.

---

[1] Actually, there are several units of measure for sound pressure level, which are related by the following set of equalities: 1 d/cm^2 = 1 microbar = 0.1 N/m^2 = 0.1 Pa. The unit abbreviations are *d* for dynes, *N* for Newtons, and *Pa* for Pascals. There are many good web sites with information about physical units of measure, especially with regard to sound measurements, in addition to the NIST sites listed in the text. For example, "Elements of psychoacoustics"[2] at the Broadcast Engineering web site.

Power of 10	Decimal Value	Prefix
15	1,000,000,000,000,000	peta (P)
12	1,000,000,000,000	tera (T)
9	1,000,000,000	giga (G)
6	1,000,000	mega (M)
3	1,000	kilo (k)
-3	0.001	milli (m)
-6	0.000001	micro (μ)
-9	0.000000001	nano (n)
-12	0.000000000001	pico (p)
-15	0.000000000000001	femto (f)

Table 1 Some standard multiples for physical units of measure.

When digital systems, such as computers and Digital Signal Processors (DSPs), operate on the physical world, the physical properties first have to be cast into the digital realm of binary numbers. When working with the binary numbers, it's convenient to adapt the decimal SI prefixes to represent powers of two instead of powers of ten. This practice works because of the coincidence that 2^{10} is approximately equal to 10^3 (1,024 compared to 1,000), and the SI prefix *k* has been adopted to refer to 2^{10} as a convenience. Likewise, 2^{20}, 2^{30}, 2^{40}, and 2^{50} are often abbreviated using the M, G, T, and P prefixes. But using the same abbreviation to mean two different values can be confusing, especially when disk manufacturers use GB to represent 10^9 bytes of storage capacity and DRAM manufacturers use the same abbreviation to represent 2^{30} bytes. (This is not due to a cynical attempt to overstate storage capacities on the part of disk manufacturers, but rather is related to differences in the natural structures underlying disks and DRAMs. See Chapter xxx on memory for the details.) The NIST has provided a set of alternate names for the prefixes to use with values expressed as powers of two rather than ten, as shown in Table 2.

Power of 2	Decimal Value	Prefix
50	1,125,899,906,842,624	pebi (Pi)
40	1,099,511,627,776	tebi (Ti)
30	1,073,741,824	gibi (Gi)
20	1,048,576	mebi (Mi)
10	1,024	kibi (Ki)

Table 2 SI prefixes for binary units of measure.

But the names for the binary prefixes are a bit awkward to write and pronounce compared to the conventional prefixes. As a result, where convention is well established and the

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

meaning is clear (when talking about disk drives and DRAM capacities, for example), the abbreviations in Table 1 continue to be used in practice.

Binary Basic Skills

You will be working a lot with binary numbers, so there are a few things about them with which you should be fluent. "Fluency" means that you should over-learn these facts so you can work with them without thinking. To start, test yourself: how fast can you say the first ten powers of two? If you have to mentally double 128 to get to 256, you know the principle but you aren't yet fluent. And if you have to write the series on paper to be sure it's right, you really aren't there at all yet. You should be able to recite them as fast as you can say the first eleven letters of the alphabet.

You should not only know the sequence of the powers of two, you should be sure you know, fluently, which decimal value goes with which exponent. If you can't answer questions like, "What is 2^9?" or "What power of 2 is 128?" immediately and without doing mental arithmetic, you need to practice until you can.

You should be comfortable with the fact that binary, like decimal, is a *positional* number system, with each position in a number corresponding to a particular power of two. For standard positional number systems, the exponents are laid along a number line as in. Figure 1.

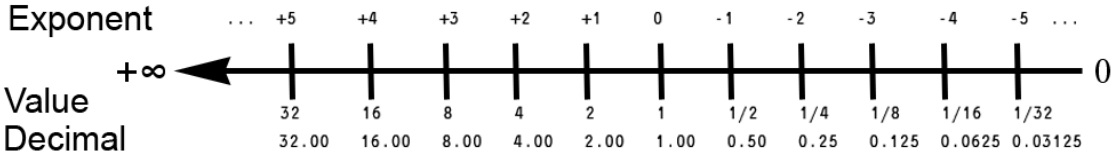

Figure 1. Number line for binary numbers, showing the base 2 exponents with their corresponding values in fractional and decimal forms.

Because the left end of a positional number has the biggest weight, it is often called the *most significant digit* or, in the case of binary numbers, the *most significant bit (msb).* Likewise, the rightmost bit is the *least significant bit (lsb).*

You need to be comfortable working with the logarithmic representation of exponents, which says that $log_{base}(base^{exponent}) \equiv exponent$. That is, the logarithm of a number is the exponent to which a number called the base must be raised to produce the number. Using base two, this concept is simply a shorthand way of expressing your fluency of powers of two: $log_2(8) = 3$, for example is just another way of expressing your knowledge that $2^3 = 8$.

Values with negative exponents are the reciprocals of the corresponding values with positive exponents: $2^{-3} = 1 / 2^3 = 1/8 = 0.125$.

Hexadecimal

Base sixteen numbers are important because they provide a compact way to represent binary numbers compactly. Because 16 is 2^4, every hexadecimal digit represents exactly 4 bits of information. One of the hazards of computing is that you should be able to

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

translate between hexadecimal and binary representations fluently. If you have to think to say that 0xC is 1100_2 or that 1001_2 is 0x9, you're doomed! Especially important in many situations is to recognize immediately that the difference between 0x7 and 0x8 is that the former (and all values below it) has a binary zero in the leftmost position and the latter (and all values above it) has a binary 1 in the leftmost position.

Measuring and Encoding Information

If this were a discussion of physics or psychoacoustics, we'd be happy to deal with measurements of the physical world and of the human response to physical stimuli. But in dealing with digital systems we need some way to be able to cast the physical world into the digital realm. The groundwork for doing this was laid by two men working in different parts of the world during the middle of the twentieth century: Alan Turing (UK) and Claude Shannon (US).

Arguably the two most significant developments in computer science during the twentieth century were Alan Turing's Turing Machine [4] and Claude Shannon's Information Theory [3]. The Turing Machine told us that any calculation can in principle be computed by a relatively simple "finite state" machine. Digital systems are often designed and implemented as finite state automata. Information theory says all information can be represented using binary numbers. At this point we are concerned with representing the physical world using the basis provided by information theory. We'll deal more with performing calculations using binary numbers in chapter 00, and we'll cover digital circuits based on finite state automata in chapter 00.

Shannon, who worked for Bell Laboratories, was interested in measuring the capacity of telephone lines for carrying voice messages. His insight was to measure information in terms of *uncertainty* and to use the *bit* as the unit of measure for uncertainty. Specifically, he defined one bit of information as the amount of uncertainty that is reduced by answering one yes-no question. Of course binary numbers were well known long before Shannon did his work in the mid-twentieth century, and the credit for coining the term *bit* to represent one *bi*nary dig*it* goes to Shannon's colleague at Bell Labs, J. W. Tukey. But the notion that something as abstract as "information" could be measured precisely using the bit as the unit of measure was Shannon's genius.

What Shannon expressed was that we gain information by reducing the number of alternatives that might exist. Shannon couched his work in terms of messages being sent over a communications channel (think telephone lines). If you want to know whether to take an umbrella when you leave the house you don't need much information. But if you want to know the expected temperature and probability of rain for the next five days, there is much more uncertainty, and you need more information to resolve it. Here are some basic examples of how to measure information based on the number of possible alternatives:

> **Example 1**. "I'm thinking of a number between 0 and 7." How much information would you need in order to know which number it is?
>
> **Solution:** There are 8 choices, and the answer is *three bits*, which you can obtain by any of a variety of binary search techniques. At each step, you ask a yes-no question that divides the amount of uncertainty in half. For example, the first

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

question could ask whether the number is even. Whether the answer is yes or no, you've reduced the number of possibilities from 8 to 4. With a total of three questions you can always know exactly which number I was thinking of, so you can say there were three bits of information in the original situation.

Example 2. "I'm thinking of one of the days of the week." How many yes-no questions do you have to ask me to find out what day I'm thinking of?

Solution: Since 7 is not a power of 2, you will sometimes need three questions, but sometime will need only 2. If your first question is, "Is it Monday, Tuesday, Wednesday, or Thursday?" and the answer is "no" and your second question is, "Is it Friday or Saturday?" and the answer is "yes," then you have to ask a third question to find out whether the answer is Friday or Saturday. But if the answer to the second question is "no" you don't have to ask any more questions because you know the answer is "Sunday." So how many bits of information are there when there are 7 unknowns? The answer is $log_2(7)$, which equals 2.808…. In the previous example, there were $log_2(8)$, which is exactly 3 bits of information. If the idea of fractions of a bit isn't intuitive to you, you can try the following experiment: Play the guessing game for days of the week with a partner. Have the person doing the thinking pick days of the week randomly and/or have the person doing the guessing use randomly chosen strategies for doing the binary search. Record how many times it takes two questions to get the answer and how many times it takes three. Compute the average of all the two's and three's and see what the answer is. For example, if you played the game 20 times and required 3 questions 16 times and 2 questions 4 times, the average would be (3*16+2*4)/20 = 2.8. (People are notoriously bad at doing things randomly, so you might prefer writing a program that generates the days or the guesses randomly.)

Although information theory gives us a way of measuring information, in itself it doesn't tell us how to *encode* information. For example, a thinker and a guesser could agree to encode the numbers 0-7 using standard binary notation: 000, 001, 010, 011, … , 111. In that case, a workable strategy for solving the first example would be to ask questions like, "Is the leftmost bit a 0?" Once the guesser knows the three binary values, he or she can simply use knowledge of the encoding mechanism to give the decimal value.

But how do you map days of the week to binary numbers? The answer is that there are many ways to do so. You could call either Sunday or Monday the "first" day, and you could use either 000 or 001 as the "first" number, but there is no single right way to assign numbers to days, and there will always be one 3-bit binary number that doesn't represent anything. The process of mapping specific binary numbers to a set of objects is called *encoding*. From an information viewpoint, as long as everyone uses the same mapping (the same code), the actual numbers used for individual objects really doesn't matter. But some codes are better than others. For example, using binary numbers that increase by one for successive days of the week might make it easier to write a program that steps through the days in sequence compared to using a random ordering.

Picking good encoding schemes is a big topic that has generated a lot of research. In this chapter we'll look at some specific types of information and some of the encoding

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

schemes commonly used with them. But we won't be covering how to generate good encoding schemes, only how some that are used actually work. We begin by reviewing how binary numbers can be used to represent numerical values. "How to use numbers to represent numbers" may seem like a simple-minded task, but it turns out to be a surprisingly complex (no pun intended) matter.

Representing Numbers

As the guessing game in Example 1 indicated, assigning binary numbers to decimal values can be a simple exercise. But two important characteristics of numbers quickly make the situation complicated: negative values and fractions.

Signed Numbers

If you have *n* bits available for representing integers, you can represent a maximum of 2^n different integer values. If the integers are unsigned, it would be natural to encode the decimal values from 0 to 2^{n-1} using the equivalent binary numbers. But if you are going to allow for negative numbers, some of the information you encode will be the sign of the value. If all numbers were either positive or negative, it would take exactly one bit to determine the sign, but there is a slight inaccuracy here because there is a special value that is neither negative or positive: *zero*.

There are four ways of encoding signed values in binary in common use today: *two's complement, biased, sign-magnitude,* and *packed decimal.* Table 3 lists the sixteen possible binary codes for the case of $n = 4$, along with the corresponding unsigned, two's complement, biased, and sign-magnitude values. The *one's complement* values, although not commonly used in practice, are included in the table for purposes of discussion.

Binary Code	Unsigned	Two's Complement	One's Complement	Bias-8	Sign-Magnitude	Packed Decimal
0000	0	0	+0	-8	+0	0
0001	1	+1	+1	-7	+1	1
0010	2	+2	+2	-6	+2	2
0011	3	+3	+3	-5	+3	3
0100	4	+4	+4	-4	+4	4
0101	5	+5	+5	-3	+5	5
0110	6	+6	+6	-2	+6	6
0111	7	+7	+7	-1	+7	7
1000	8	-8	-7	0	-0	8
1001	9	-7	-6	+1	-1	9
1010	10	-6	-5	+2	-2	(+)
1011	11	-5	-4	+3	-3	(-)
1100	12	-4	-3	+4	-4	+

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

1101	13	-3	-2	+5	-5	-
1110	14	-2	-1	+6	-6	(+)
1111	15	-1	-0	+7	-7	(+)

Table 3 Signed Number Encodings

Before reviewing how each of these encoding schemes "works," there are a couple of features of Table 3 to notice. The first is that one's complement and sign-magnitude both have two representations for zero, marked +0 and –0 in the table. This feature makes them awkward to work with because any testing for zero has to involve two tests when using these encodings. And having two representations for zero makes these codes inefficient because they cannot represent as many different values as the other codes.

The second thing to notice is that two's complement values increase in an orderly fashion as the unsigned binary codes increase in magnitude, except for the big jump between +7 and –8. Biased numbers, on the other hand, increase in strict binary order as the binary codes increase.

The third thing to recognize, and this is a common misperception among many people, *all* the values listed in the first column are "two's complement numbers" (or biased, or sign-magnitude, or one's complement for the other columns). That is, 0011_2 is just as much a two's complement number as 1101_2 when two's complement encoding is being used. The first number is the two's complement code for +3 and the second one is the two's complement code for –3. A number doesn't have to be negative to be represented using two's complement (or any other) encoding.

Finally, the packed-decimal column is different from the other encoding schemes because it uses four bits to represent the sign of the number, with 0xC (1100_2) and 0xD (1101_2) being the "preferred" representations, and alternate plus and minus codes indicated in parentheses. Using packed decimal, there are four representations for "positive zero" and two representations for "negative zero!" Actually, it's even worse than that; see the section on packed decimal below for more information.

Two's Complement Encoding

Most computer science students already know that negative two's complement numbers have a 1 in the leftmost bit position and that you take the two's complement of a number by "flipping the bits and adding 1." You may also know that $-x$ is represented as an n-bit two's complement number as $2^n - x$. For example, the 4-bit two's complement representation of –3 is $16 - 3 = 13 = 1101_2$.

Another way to look at two's complement encoding is that it is the same positional number system as unsigned binary numbers, but with a negative weight for the leftmost bit. Thus, for 4-bit two's complement numbers, the value 1101 would represent:

And 0011 would represent:

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

Because positive numbers and zero have a 0 bit in the leftmost position, the negative weight of that bit doesn't get added into the value, and these numbers can be evaluated the way unsigned binary numbers are.

A negative weight for the msb of two's complement numbers gives rise to an important concept of *sign-extension*, which in turn provides a shortcut for figuring out the values of many two's complement numbers.

When a digital system works with numbers, there is always a fixed number of bits that the logic circuits work with. In a high level programming language, you see this as the "size of a variable." Java is a very clean language in this respect: if you declare a variable to be of type *int*, it will be represented using 32 bits. Variables of type *byte* are 8 bits, of type *short* are 16 bits, and of type *long* are 32 bits. At the other extreme, languages like Handel-C allow you to specify exactly how many bits each variable occupies. But the net result is the same: every variable occupies a fixed number of bits. Sign-extension is the mechanism that allows the value of a variable of one size to be stored in a variable of another size without changing the value represented. For example, if a 4-bit two's complement variable holds 1101_2 (-3_{10}) and it is to be copied into a 5-bit variable without changing the value, the result will be 11101_2, which is obtained by making a copy of the sign bit on the left end of the new variable. The value of the new variable is still –3:

By extending the sign bit one place to the left we've added negative 16 to the encoded value, but at the same time the negative 8 in the 2^3 position has now become positive 8, an increase of +16 that exactly offsets the value of the new sign bit. Every bit position in a binary number has exactly twice the weighted value as its neighbor to the right, so this process of sign extension can be repeated indefinitely: 101 = 1101 = 11101 = 111101 … . The process works just as well for positive values as for negative values, with the simplification that all the leftmost zeros in a positive number have no effect on its value: 011 = 0011 = 00011 = 000011 … .

You can use the concept of sign extension to simplify the process of evaluating a negative two's complement number. You can safely ignore a continuous string of one's on the left end of a two's complement number except the rightmost one. So if you are given the number $11111111111111111111111111111110_2$ to evaluate, you can just collapse all the leftmost ones to a single one: 10_2 and simply evaluate

. (What value does the two's complement number 1_2 represent?)

This technique only works for a continuous string of ones on the left, though. Just as you couldn't ignore the leftmost one in 0100000000000000001, you can't ignore the leftmost zero in 10111111111111111110.

Remembering from your hexadecimal fluency that 0xF is 1111_2 makes two's complement values like 0xFFFFFF80 particularly easy to evaluate. (Do you have to read this sentence to know it's –128?)

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

Finally, coming back to the common misperception mentioned earlier about only negative numbers being "two's complement," let's make a distinction between "two's complement encoding" and the "two's complement operation." Encoding simply means that the value can be determined by assigning a negative weight to the leftmost bit, and applies equally well to both negative numbers (where the leftmost bit is one) and zero or positive numbers (where the leftmost bit is zero). The two's complement *operation*, that business of flipping the bits and adding one, is more properly thought of as *negation.* You can negate a positive number to get the equivalent negative number. But you can just as easily negate a negative number to get the corresponding positive number. And you can negate a value without paying attention to its sign by performing exactly the same steps of flipping the bits and adding one:

0011 Original value, +3

1100 Flip the bits

0001 Add one

1101 Negation of +3 is –3

0010 Flip the bits of -3

0001 Add one

0011 Negation of –3 is +3

Numerically, flipping the bits and adding one is the same as subtracting a number from 2^n. In our example, n is 4, and the encoding of –3 is 16-3 = 13 = 1101_2. Going the other way, 16- -3 is 19, or 10011_2. But because we are working with n=4, the leftmost of those five bits is discarded, leaving the correct encoding of +3, which is 0011_2. It is a general characteristic of two's complement numbers that adding, subtracting, and negation operations will result in an n+1 bit answer, and that the leftmost bit in such cases is simply ignored. This topic will be covered in more detail in Chapter 00 when we discuss carry and overflow while performing two's complement arithmetic.

So, what happens if you negate a two's complement zero? There's only one zero, so do you get zero, or something else? (Try it.)

Since there is only one two's complement representation of zero, and since that representation has the same value of its sign bit as all the positive values, and since there has to be an even number of values that can be encoded in *n* bits no matter what the value of *n* is, it follows that there has to be one more negative two's complement number than there are positive numbers. In Table 3 the extra negative value is –8. In general it is -2^{n-1} and the range of values that can be represented using two's complement is -2^{n-1} to $+2^{n-1}-1$.

So, what happens if you try to negate -2^{n-1}? (Try it for $n=3$.) The moral is that the extra negative value, when using two's complement, is an anomaly. If you are developing a system that people's lives or money depends on and it's at all possible for that maximum

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

negative value to occur, you have to check for it and design your system to respond appropriately when it occurs.

One final point about two's complement encoding: it's almost universally used for encoding signed integers because the negation operations (flipping bits and adding one) can be done very easily using digital circuits. This means that instead of having two separate circuits for addition and subtraction, a digital system can achieve the same functionality using a single addition circuit coupled with simple negation logic. Whether the subtrahend is positive or negative in value doesn't matter, just add its negation to the minuend to get the answer[2].

One's Complement Encoding

With two's complement under your belt, one's complement is trivial to understand. But also pretty useless! One's complement negation is even simpler than two's complement negation: just flip the bits. And you don't have to worry about that extra negative value when doing arithmetic. Unfortunately the two zeros complicate things enormously, and you can't subtract simply by negating the subtrahend and adding.

But understanding the name "one's complement" itself can be useful. One way to think of the "flip the bits" operation is to subtract the n-bit number to be negated from the number consisting of *n* ones. There are only two possible cases: 1-0=1 and 1-1=0, and in both cases the resultant bit is the opposite value of the bit being subtracted. So "one's complement" means that you negate by subtracting from all ones.

Biased Encoding

In the context of encoding integers, a bias is a value that is subtracted from an unsigned binary number to get a signed value. In Table 3, the bias chosen was 8, and the encoded values ranged from –8 to +7. But the bias could be any value. For example, if the bias were 1, the encoded values would have ranged from –1 to +14. A bias equal to half the range gives the best balance between the number of positive and negative values that can be represented, but there are situations where it may be advantageous to use a bias that isn't exactly half the range. You'll see such a situation when we discuss floating-point numbers below.

Biased encoding makes addition and subtraction difficult. Basically, if you add two numbers, you have to subtract the value of the bias from the result to get the correct answer, which leads to complicated arithmetic circuits. But biased notation has two big advantages over other systems. It's the only encoding that preserves the numerical order of the unsigned code words in the encoded values, and except for two's complement it's the only encoding with a single zero. These two properties make it very easy to design digital circuits for comparing biased numbers. Again, you'll see more about why this is important in the discussion of floating-point numbers below.

---

[2] The minuend is the one on top and the subtrahend is the one on the bottom in a paper and pencil subtraction problem.

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

Sign-Magnitude Encoding

The name of this encoding scheme says it all: a signed number is represented using one bit to indicate the sign (typically, 0 means positive and 1 means negative), and the remainder of the bits represent the absolute value (magnitude) of the number. Negation is as easy as flipping the value of the sign bit. But using sign-magnitude means that there is no easy way to do arithmetic operations as simple as addition and subtraction; there is a considerable amount of overhead required to determine the signs of the operands, whether to add or subtract, and what the correct sign of the result is. This overhead is way too much for efficient integer calculations, but floating-point numbers require such complicated digital circuits for doing their arithmetic, that sign-magnitude encoding fits in well with that situation.

Packed Decimal Encoding

Although we are including packed decimal in our discussion of integer encoding, it actually finds its major use in calculations involving decimal fractions. It is particularly useful when the computer must accurately mimic paper and pencil calculations that must be accurate to a certain number of decimal places, such as the hundredth part of a dollar. Even though packed decimal does not actually provide explicit support for fractions, the calculations can be done in units of pennies (hundredths of a dollar), and the results presented to the person reading them with a decimal point printed between the dollar and penny parts. It is also particularly easy to convert between the character representation of a decimal number and it's packed decimal representation, and *vice-versa.*

The basic idea of packed decimal encoding is to use hexadecimal digits to represent decimal digits. As Table 3 indicates, the six hexadecimal digits that don't correspond to decimal values are used to represent the sign of the value, which normally occupies the rightmost hexadecimal position. Thus 0x01234C is $+1234_{10}$ and 0x01234D is -1234_{10}. In a computer with a byte-structured memory, packed decimal numbers would be stored two digits per byte, requiring all values to have an odd number of decimal digits. If there is an even number of decimal digits, as in the example value of +1234, the packed decimal would be padded with a leading zero to make the number of digits odd and the total number of bits a multiple of 8.

Although converting between the textual representation of numbers and the equivalent packed-decimal representation is quite straightforward, packed decimal arithmetic is quite complex, and requires digital circuits to simulate the processes of paper and pencil calculations quite literally.

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

Numbers With Fractions

Fixed-Point Numbers

Floating-Point Numbers

Text

Images

Sound

Exercises

1. Browse the Physics Laboratory of NIST's web site [[1]] and answer the following questions:
    a. What is the range of values for which the NIST lists named prefixes?
    b. Are all named units of measure separated by three orders of magnitude (multiples of 10^3), like the ones listed in Table 1?
    c. The text mentions mass, length, and time. What is the full list of physical attributes for which there are SI standard units?
    d. Using Wikipedia.com or Encyclopedia.com, find out the relationship between the *mks* and *cgs* systems of measurement.
2. Converting from one unit to another can lead to wrong results if you aren't careful. For example, you know there are 12 inches in a foot, so the "conversion factor" between inches and feet is 12. Or is it 1/12? Answer the following questions:
    a. One inch = _____ feet.
    b. One foot = _____ inches.
    c. To convert inches to feet, multiply the number of inches by _____.
    d. To convert feet to inches, multiply the number of feet by _____.
    e. One microsecond = _____ nanoseconds.
    f. One nanosecond = _____ microseconds.
    g. To convert microseconds to nanoseconds, multiply the number of microseconds by _____.
    h. To convert nanoseconds to microseconds, multiply the number of nanoseconds by _____.
3. An audio tone has a frequency of 1,760 Hz. What is its period,
    a. In seconds?
    b. In milliseconds?

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.

c. In microseconds?

d. In nanoseconds?

4. In discussing one's complement encoding we saw that "one's complement means that you negate by subtracting from all ones." Why is there an apostrophe in the "one's" at the beginning of that sentence and not in the second "ones?" The correct answer is more than an exercise in grammatical pedantry, and requires you to do some research on one's and two's (and nine's and ten's) complement notations.

References

[1] NIST. *International System of Units from NIST.* http://physics.nist.gov/cuu/Units/.

[2] Robin, Michael. *Transition to Digital: Elements of Psychoacoustics.* http://artistoftheyear.broadcastengineering.com/ar/broadcasting_transition_digital_elements/.

[3] Shannon, C. E. "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379-423 and 623-656, July and October 1948.

[4] Turing, Alan. "On computable numbers, with an application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society*, Series 2, 42 (1936), pp 230–265.

Copyright © 2005-2006 by Christopher Vickery, all rights reserved.