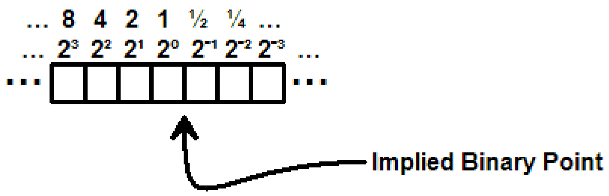
Lecture #13 – February 27, 2004

Introduction to Floating Point

Up to this point we have considered only integer representations of numbers. While a lot can be done with integers, the need arises from time to time to deal with real numbers, or numbers with fractions. As we will see, the limitations of storing numbers with fractions into a limited number of bits in computer memory prevent us from using all possible real numbers. In particular, nearly none of the "interesting" numbers can be stored perfectly into computer memory with full precision (which would be infinite). The best we can do is to store *approximations* to those numbers. The difference between the true value of a number and the approximation we are forced to use is the ***round-off error***, and we must monitor our calculations carefully to insure that the ***cumulative round-off error*** does not grow so large over time that it swamps our expected answer.

Conversions

In an integer, the rightmost bit has the value 2^0, or 1, and bits continuously double in value as you proceed to the left. Similarly, starting at some arbitrary bit position in an integer, bits continuously halve in value as you proceed to the right, until the rightmost bit is reached, which again has the value 1. If you assume that the decimal point, or more properly the ***binary point***, is just to the right of the 2^0 bit, then continuing the halving process gives us bits 2^{-1}=½, 2^{-2}=¼, 2^{-3}=⅛, etc. This is illustrated in the following diagram:



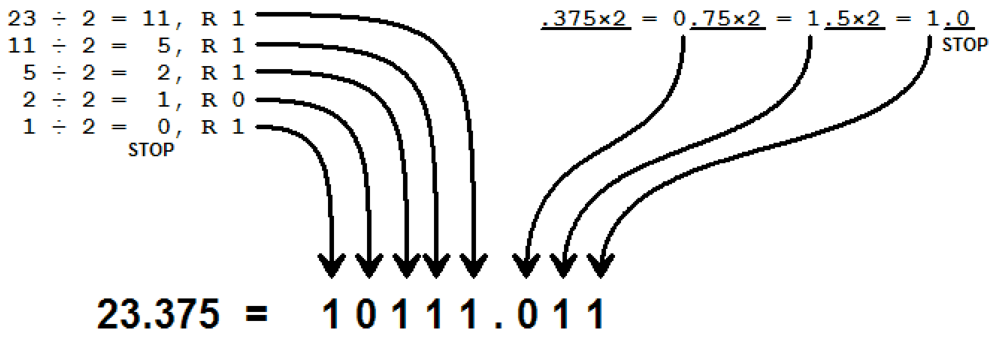
With this information we can generate simple binary fractions. For example, the decimal number 23.375 has 10111.011 as its equivalent binary representation. Converting the whole portion of the number to binary is the same here as for any integer: 23 = 1×16 + 0×8 + 1×4 + 1×2 + 1×1 = 16+0+4+2+1 = 23.

Binary fraction bits have values 2^0=½=0.5, 2^{-1}=¼=0.25, 2^{-2}=⅛=0.125, etc. The decimal fraction 0.375, which is the same as the rational number ⅜, is therefore the sum of 0×½ + 1×¼ + 1×⅛, or 0.011 in binary.

The process of converting from decimal to binary is very straightforward if you remember that multiplying by two in any base is the same as a left-shift of one binary bit, and that dividing by two in any base is the same as a right-shift of one binary bit. For whole numbers you continuously divide by two and record remainders (which can be only 0 or 1) until there is nothing left of the original number; the list of remainders contains the bits of the binary number in right-to-left order.

For fractions you continuously multiply by two and record any whole part (which can be only 0 or 1) until there is nothing left of the original fraction or until you have "enough" bits. The list of bits that results is in natural left-to-right order.

For a number such as 23.375 the process is to convert the whole part (23) separately from the fractional part (0.375). First we convert the whole part: when dividing by two, we record the remainder (0 or 1), and then *drop that remainder* from the result leaving a smaller number for the next step. Next, we convert the fractional part: when multiplying by two, we record the whole part (0 or 1), and then *drop that whole part* from the result leaving just the fraction for the next step. This process is shown below:



These techniques work well for numbers where the fraction is composed of a small, finite list of inverse powers of two. Unfortunately, that is a very small portion of the set of all possible real numbers. Transcendental numbers such as π (approximately 3.14151926535…) have an infinite number of non-repeating digits in decimal, and the situation isn't improved when the numbers are converted to binary. Rational numbers such as ⅓, which have an infinite number of repeating digits in decimal (0.33333…), also have an infinite number of fraction bits in binary.

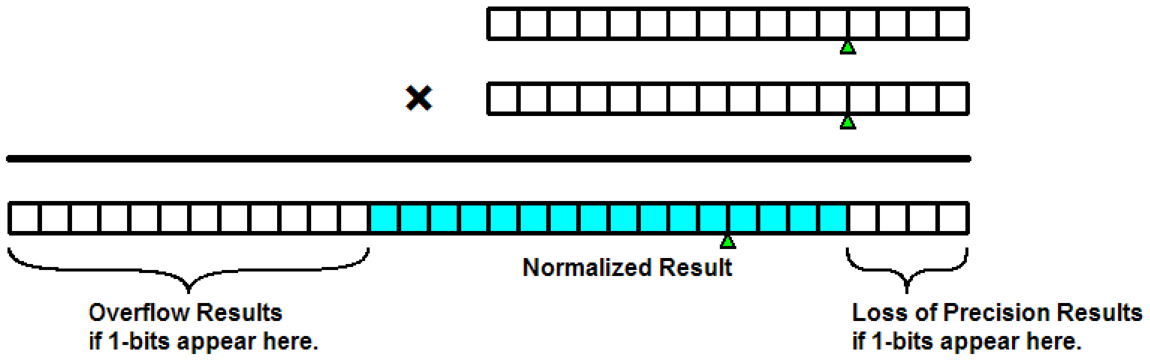
What is surprising, however, is that some perfectly well behaved rational numbers in decimal also have infinite binary fractions. The classic example of this is 1/10, which has 0.1 as its decimal value. In binary the equivalent value is 0.000110011001100…, which repeats the pattern "0011" forever.

Variables in memory have finite sizes. Precision is lost any time you attempt to store an infinite number of bits into a fixed-size slot! This is the great "dirty secret" of computer science; nearly all real numbers lose fraction bits when stored with a fixed number of bits.

Simple Fixed-Point Storage Techniques

The simplest approach to storing numbers with fractions into memory is to arbitrarily assign some chunk of bits to contain the whole part and the rest to hold the fraction. This representation is called ***fixed-point***. For example, in a 16-bit memory word you might assign 12 bits to the whole part and 4 bits to the fraction, 5 bits to the whole part and 11 to the fraction, or split the difference and assign 8 and 8. As far as addition and subtraction are concerned, one choice is just as good as another. The only requirement is that fixed-point representations must not be mixed; if two numbers are added or subtracted the binary points must "line up" or the result will be meaningless.

For multiplication the rules are a little more complicated. In any base, the count of digits in the product will be the sum of the digits counts of the two numbers, and the count of digits in the fraction will be the sum of the digit counts of the fractional parts of the two numbers. For example, when multiplying two 16-bit numbers together, where each number has 12 whole bits and 4 fraction bits, the product will be a 32-bit number with 24 bits of whole part and 8 bits of fraction. The last step, called ***normalization***, is to extract the 16 bits containing the binary point aligned to the correct position. Loss of precision results if the discarded fraction bits are non-zero, and overflow results if the discarded whole bits are non-zero. This is shown below:



This normalization process is often simplified if the number of bits in the whole part and the number of bits in the fraction correspond to byte boundaries. For 16-bit numbers this usually means 8 whole bits and 8 fraction bits; for 32-bit numbers this means 16 whole bits and 16 fraction bits.

As an unsigned integer, 16 bits is enough to hold any value between 0 and 65535, but the largest number possible is only 255.996 when split into half whole bits and half fraction bits. Similarly, 32 bits can hold a maximum unsigned value of 4294967295, but the largest possible number is only 65535.99998 when split into half whole bits and half fraction bits. By allocating bits to the fraction, you lose out big time on the maximum integer value.

While fixed-point arithmetic is actually pretty easy to implement in assembly language, the representation has some serious pitfalls. Very large numbers and very small numbers can't be constructed; even if whole bits or fraction bits are not required for some particular number they cannot be reallocated to the section were they are most needed. These pitfalls are addressed in the next section.

Floating-Point

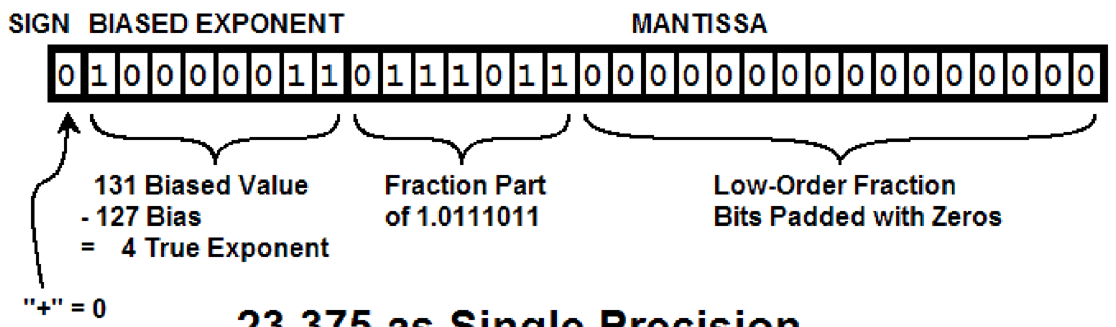
In ***floating-point*** representations, very large and very small numbers are equally constructible. Most similar to scientific notation, floating-point representations require both a normalized fraction and a scale factor to properly position the binary point. In our earlier example we saw that the decimal number 23.375 is 10111.011 in binary. The normalized scientific notation version of 23.375 is 2.3375×10^1, and the normalized binary version of 10111.011 is 1.0111011×2^4.

With the exception of zero, *every* normalized binary number is of the form $1.\mathrm{xxxx}\times2^{\mathrm{YYYY}}$, where the bit to the left of the binary point is always a 1. If it is always known to be a 1, we don't need to store it. This releases one more bit of precision to be used for the fractional part of the number.

The power of 2 can be either positive or negative, where positive values indicate that the binary point should be shifted some number of bits to the right and negative values indicate that the binary point should be shifted some number of bits to the left. Rather than store this exponent as a signed integer, most floating-point representations add a ***bias*** to that value to insure that the resulting ***biased exponent*** is always positive.

In ***single-precision*** floating-point, the 32 bits of a word are divided up into three regions: 1 bit for the sign (0 for "+" and 1 for "-"), 8 bits for the biased exponent, and 23 bits for the fraction, called the ***mantissa*** or ***significand***. The bias is +127.

In the example, 1.0111011×2^4, the biased exponent value is 4 + 127 = 131, which is 10000011 in binary. The mantissa consists of everything to the right of the binary point, discarding the leading 1. Unused low-order mantissa bits are filled with 0. Since the number is positive the sign bit will be 0. The final single-precision binary version of 23.375 stored in memory is:



23.375 as Single Precision