Floating-point representation

Actually, let's start off with the more general question of representing fractional numbers. Representing unsigned integers was quite straightforward: you just convert into base two and you're done. Representing signed numbers is almost as straightforward. But how to represent fractional numbers?

Generally speaking, the problem in representing *any* information in a digital computer is that you have to reduce the information to an unambiguous string of zeroes and ones. When we are encoding numbers, we impose an additional restriction in the name of convenience and efficiency: that is that the information has to fit in a field of constant width.

So the problem is that we have only two symbols with which we can store information, 0 and 1, but there are now three symbols which we have to encode: 0, 1, and the *radix point*. (In base ten, the radix point is what you know as the *decimal point*, which term is applicable only to base ten numbers. ''Radix point'' is the more generic term.)

Fixed point

The easiest way would be to assume that the radix point always falls at the same point in the number. To take a specific example, say we designate the point between the eighth and ninth most significant bits as the radix point position. Like this:

| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | . | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

How do you interpret this? Well, the process is analogous to interpreting whole numbers. In an *n*-bit binary whole number, the righmost bit got multiplied by 2 to the zeroth power, the next-to-rightmost by 2 to the first, the next by 2 to the second, and so on until the leftmost digit was multiplied by 2 to the (n-1)th. It's basically the same deal here, except that digits to the right of the radix point are going to get multiplied by 2 to negative exponents. So the first digit to the right represents 2 to the negative one (one half), the next digit represents 2 to the negative 2 (one fourth), and so on. So, to sum up the digits from the above number:

```
0 * 2^7  (128)        =  0.0
1 * 2^6  (64)         = 64.0
0 * 2^5  (32)         =  0.0
1 * 2^4  (16)         = 16.0
1 * 2^3  (8)          =  8.0
0 * 2^2  (4)          =  0.0
1 * 2^1  (2)          =  2.0
0 * 2^0  (1)          =  0.0
.
1 * 2^-1 (.5)         =  0.5
0 * 2^-2 (.25)        =  0.0
0 * 2^-3 (.125)       =  0.0
1 * 2^-4 (.0625)      =  0.0625
1 * 2^-5 (.03125)     =  0.03125
1 * 2^-6 (.015625)    =  0.015625
0 * 2^-7 (.0078125)   =  0.0
```

```
     0 * 2^-8 (.00390625) =  0.0
                            ---------
                            90.609375
```

The earliest computers that handled fractional numbers represented them this way.

The main advantage of this representation method is its simplicity. All arithmetic operations are relatively straightforward. The major drawback is its inefficiency. We have sixteen bits here; let's try to represent the number 256.5. The whole part of the number is represented as 100000000b, the fractional part as .1b --- a total of ten digits. But not representable using the system outlined above, because the whole part cannot take more than eight digits.

What we need is a way to represent the numbers so that the radix point can float to different places in the number depending on the magnitude of the number --- a *floating-point* representation.

Floating-point

Consider scientific notation, which represents numbers in the form *a times ten to the b*. Floating-point representation is similar, but with base two. So we break the number into two parts: the part that holds *a* is called the *mantissa*, the *significand*, or the *fractional part*, depending upon who you ask. I learned *mantissa*. *b* is called the *exponent*, or the *characteristic*, also depending on who you ask. I learned *exponent*. Let's take the same 16-bit field and divide it into a 4-bit exponent and a 12-bit mantissa. 256.5 decimal is 100000000.1b or in binary scientific notation, 1.000000001*10^1000. We could represent this as

1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0

exponent *mantissa*

With the fixed-point representation the smallest number (other than zero) that could be represented as was 00000000.00000001b or 0.00390625 decimal; the largest was 11111111.11111111b or 255.99609375 decimal. With the floating-point representation, the smallest number (other than zero) we can represent is 0.000000000001*10^0b or 0.000244140625 decimal and the largest is 0.111111111111*10^1111b or 134184960 decimal. If we use a signed exponent, then we can represent much smaller numbers, though we won't be able to represent as large numbers. If we also use a signed mantissa, then we can represent negative as well as positive numbers. This is the style of representation used in modern computers.

It should be noted that, for the simple reason

Normalization

One consequence of the mantissa/exponent method of representing floating-point numbers is that there can be multiple representations of the same number. For example, the number 1.5 could be represented as 0.11*10^1b or as 0.011*10^10b or 0.0011*10^100b, etc. For this reason, as well as to maximize precision, floating-point numbers are stored in a *normalized* form, that is such that the most significant bit of the mantissa is always 1. Ordinarily, you assume that the radix point is located immediately after this digit.

To take an example, let’s say we want to subtract 2.0 from 3.0. The floating-point representations of these numbers are 1.0*10^1b and 1.1*10^1b, respectively. Since both have the same exponent, we can subtract the mantissas directly, giving 0.1*10^1b. To normalize this number, we take advantage of the fact that you can multiply and divide floating-point numbers by powers of two in two different ways. You can either shift the mantissa (as with the `shl` and `shr` instructions) or you can add to or subtract from the exponent. Shifting the mantissa to the right one place is equivalent to subtracting one from the exponent; shifting the mantissa to the left one place is equivalent to adding one to the exponent. So, to normalize a floating point number, we determine how many places to the left we need to shift the mantissa, shift it that number of places, and then subtract the number from the exponent. So to show our example being normalized:

$$0.5*2^1 = 1.0*2^0 = 1.0$$

If we want to add or subtract numbers with different exponents, we apply the same sort of transformation to the one with the lesser exponent (except shifting to the right and adding to the exponent) so that the exponents match. Then the mantissas can be added or subtracted directly.

If we assume that floating-point numbers are always going to be normalized, then we can gain an extra binary digit of precision by having the most significant bit be implicit. So if you have the number 1.1*10^1b, the part that is actually stored is .1*10^1b, the digit in the one’s place being assumed.

ANSI/IEEE 754-1985

Floating-point representations were standardized in 1985 by the Institute for Electrical and Electronic Engineers (IEEE) and the American National Standards Institute (ANSI) on 26 July 1985. The formats defined in the document have become nearly universal and are implemented in the 80387, 80486, and Pentium. The 8087 and 80287 were both designed before certain details of the standard were decided and therefore there are some incompatibilities.

ANSI/IEEE 754 specifies four floating-poing formats in two groups: basic and extended. Within each of these groups, there is a "single-precision" format and a "double-precision" format. The basic formats are specified exactly in terms of field widths and semantics; the extended formats are specified only in terms of minimum requirements for precision. The basic single-precision number is 32 bits wide, has an 8-bit exponent and a 24-bit mantissa. The basic double-precision number is 64 bits wide, has an 11-bit exponent and a 53-bit mantissa.

Both mantissa and exponent are signed numbers, but neither is represented in two’s-complement format. The mantissa is stored in sign magnitude format and the exponent in something called *biased form* (also called *excess-n form*). In biased form, there is a positive integer (called the *bias*) which is subtracted from the stored number to get the actual number. For example, in an eight-bit exponent, the bias is usually 127. To represent 0, you store the number 127. To represent -100, you store 27. And so forth. The most significant bit of the exponent can be read as a sign bit, but its semantics are reversed from the usual. Zero represents a negative number and one a positive number. The reason for this representation is one of efficiency. Comparing exponents is much faster because you can simply scan the bits from most significant to least and the first one that differs tells you which one is greater. Since there is not usually a need to compare the exponent of a floating-point number with anything other than another exponent, this difference of representation is not a problem. Another thing to note about the exponent is that an exponent of all zeroes and an exponent of all ones are both reserved for special cases (which I'll cover in a few minutes), so in an eight-bit field you get possible exponents of -126 to +127. In the basic floating-point formats, the mantissas are assumed to be normalized so that the most significant bit is one and it is stored in the one's digit. Only the fractional part is stored.

The IEEE 754 floating point formats are broken into three fields. The most significant bit stores the sign of the mantissa, the next group of bits stores the exponent in biased form, and the rest of the bits store the magnitude of the fractional portion of the mantissa. So the number 2.25 would be stored in a basic single-precision number as:

```
 0    01111110 00100000000000000000000
sign  exponent         mantissa
```

or 1.125*2^1, the one being implicit.

Special numbers

- Zero: zero is represented with an exponent of all zeroes and a mantissa of zero. Note that because the mantissa is stored as a sign magnitude number, there are two zeroes, a positive one and a negative one.

- *Denormal numbers*: To store very small numbers, it is necessary to be able to store them without normalizing them. To do this, you give the exponent as zero, which is interpreted as zero minus the bias, and then for the mantissa, you a fractional part, but the one's digit is assumed to be zero. This lets you represent numbers ordinarily too small to be represented, but there is a cost in terms of precision. If the most significant digit is stored in the 12th digit of the mantissa, then you only have half the significant digits that you ordinarily would.

- Infinity: originally in the 8087 and 80287, there were two different models for representing infinity: *affine closure* and *projective closure*. Projective closure was the default model. Under projective closure there is a single representation for infinity which stood for any number whose magnitude was too large to be represented. Under affine closure, there were two infinities, a positive infinity which represented any positive number whose magnitude was too large to be represented and a negative infinity which represented any negative number whose magnitude was too large to represent. IEEE 754 specified affine closure as the only model, and when the 80387 was designed, projective closure was dropped.

  You represent infinity with an exponent of all ones and a magnitude of all zeroes. You differentiate between negative and positive infinity by use of the sign bit.

- Not a Number (NaN): This is a catch-all representation for any absurd, unrepresentable, or undefined results, such as dividing by zero, multiplying infinity by zero, taing the square root of a negative number (complex numbers not supported), etc. NaN is represented by an exponent of all ones and a magnitude greater than zero. This allows for encoding of many many different conditions which could generate a NaN.

  NaNs can be *signaling* or *quiet*, the difference being that signaling numbers are not meant to be used as numeric operands and quiet NaNs can. In the Intel implementations, the most significant bit of the magnitude is one for quiet NaNs and zero for signaling.

Exceptions

According to IEEE 754, there are five conditions which can generate an exception. An exception can be handled by either setting a flag or calling a user-defined exception handler. The five conditions are (1) invalid operation, (2) divide by zero, (3) overflow, (4) underflow, and (5) inexact result.

Invalid operation

This exception is generated by (1) an attempt to use a signaling NaN as a numeric operand, (2) addition or subtraction where both operands are infinities, (3) multiplying infinity by zero, (4) dividing zero by zero or infinity by infinity, (5) a remainder operation where the first operand is zero or the second operand is infinity, (6) taking the square root of a negative number, (7) conversions to integers which cannot be faithfully represented, (8) comparing a NaN to a number or to another NaN.

Divide by zero

Dividing by zero (except dividing zero by zero) produces a result of infinity and invokes the divide by zero exception.

Overflow

Overflow is handled, depending upon how you have selected to perform rounding, either by setting the result as positive or negative inifinity or by setting the result with the greatest possible magnitude. This is called *abrupt* or *sudden* overflow, as you immediately lose the exact result.

Underflow

Underflow, on the other hand, is handled *gradually*. The result is stored as a denormal. Abrupt underflow (which is not implemented) would entail setting the result to zero, which can be a problem if you're subtracting two different numbers and then get zero. Suddenly, adding the second number back to the result gives you a number other than the first number.

Inexact result

This is quite common and results from any operation whose result cannot be represented exactly. For example dividing 1 by 3.

80x87 data types

The 80x87 has single- and double-precision numbers per the IEEE 754 specs, and it also implements an "extended-precision" format, 80 bits long, which is used internally to represent all numbers. It is highly recommended that you do not use this extended-precision number except to store intermediate results of calculations, because it provides a safety mechanism in terms of loss of precision.

The extended precision is a little bit different from the others in that the most significant bit is not implicit. All digits must be explicitly represented; the most significant digit in the magnitude field is the one's digit rather than the one-half's digit as in the smaller formats. The reason for this is that the extended precision is what the 80x87 uses internally and it has to be able to use denormalized numbers to perform many of its operations.

The 80x87 also provides conversions from integer formats. It understands three sizes of signed integer, *word* (16 bits), *short* (32 bits), and *long* (64 bits). It also understands Binary Coded Decimal numbers, which we have not gone over yet and which can safely be ignored for most purposes.