Subject: Re: How Computers Represent Floats
Posted by William Clodius on Thu, 30 Nov 2000 08:00:00 GMT
View Forum Message <> Reply to Message

Almost every computer newsgroup on programming languages or numerics has this discussion at one time or another. The vast majority of programming languages rely on the hardware representation of floating point numbers so that this issue is essentially programming language independent. The vast majority of computer systems (where computer systems excludes hand held calculators which often implement binary coded decimal) now implement the core of the IEEE 754 standard, so my remarks will be confined to such systems. However the minority of computer systems that do not implement the IEEE 754 standard share many of the same quirks.

Details on these quirks, with an emphasis on the IEEE 754 standard, are available from reports by David Goldberg, "What Every Computer Scientist Should Know about Floating Point Arithmetic," and the subsequent supplement to that report by Doug Priest, "Appendix D". Both reports were written for Sun Computer systems and are available from docs.sun.com in pdf and postscript formats. Links to the documents are available at www.validgh.com.

In almost any binary system including IEEE 754, most of the time floating point numbers can be thought of as divided into three parts, a sign, a mantisa, and an exponent stored in a fixed size "word". In IEEE 754 the mantisa can be thought of as an integer with values from 2^n_mant to 2*2^n_mant-1, where n_mant is the number of bits available for the mantisa. Note that the mantisa is non-zero. In IEEE 754 the exponent can be thought of as a scale factor multiplying the integer, where the scale factor is a simple poser of two whose relative range is determined by the number of bits available for the exponent. IEEE 754 requires that the computer make available to the user two such representations: what we normally think of as 32 bit single and 64 bit double precision. IEEE 754 requires that all intermediate calculations be performed a higher precision so

The fact that the data is stored in a fixed size word results in the first surprise to very inexperienced users: this representation is correct for only a finite number of values. This representation cannot deal with all elements of the countable set of all rationals, let alone the uncountable set of all irrationals. Inaccuracies and errors are almost unavoidable in any attempt to use this data type, except for knowledgeable users in limited domains.

This binary representation does not let IEEE 754 exactly represent numbers that are not simple exact multiples of powers of two. This results in the second surprise to many users: most simple decimal floating point numbers (e.g., 0.3 or 0.1) cannot be exactly represented

and any attempt to store such numbers results in errors. E.g., 0.1 might become 0.1000000005 when stored.

Most manipulations of IEEE 754 numbers result in intermediate values that cannot be represented exactly by single and double precision numbers. This results in the third surprise, most manipulation result in a cumulative loss of accuracy. To minimize this loss of accuracy it requires that intermediate calculations be performed at a higher precision with well defined rounding rules to obtain the final representation of the intermediate results. Most systems appear to use double precision to represent intermediate results for single precision calculations. All systems must use a precision higher than double for intermediate results of double precision calculations. This higher precision representation need not be available to users, however the Intel extended precision is essentially this higher precision intermediate type (it differs slightly in ways that irritate numericists). When this higher precision type is available it need not have the well defined rounding and error propagation propers of single and double precision.

While most of the time IEEE 754 has this behavior there are exceptions represented by special values. Obviously there has to be a zero value. IEEE 754 also has special values such as + or - infinity to represent such things as dividing a finite number by zero, NaN (Not a Number) for representing such things as zero divided by zero, and a signed zero. This allows calcuations to procede at high speed code and the post-facto recognition and (if neecessary) correction of problems with the algorithm or its "inaccurate" implementation in IEEE 754. It also generates floating point exceptions that can be detected automatically withot examining all the output numbers. Many languages were developed before IEEE 754 and do not map naturally to this model.

The existence of infinities and NaN leads to a fourth surprise to many users: obviously bad results can be generated and the computer does not stop the instance it detects such "bad" values. As there are prblem domains where such values are expected and not an indication of problems, it is up to the user to check for such values if they can be generated and when generated indicate a problem.