UNDERSTANDABLE ARITHMETIC

Pat H. Sterbenz
Brooklyn College
Brooklyn, New York

Summary

Since the floating-point operations form the basic steps in our programs, the programmer has to understand the results that will be produced by these operations. This paper discusses operations which have been or might be implemented in the hardware. The emphasis is on making the results easy for the user to understand.

Introduction

Most programs are written in a high level language, and the programmer has to understand the basic operations that will be performed by the statements he writes. Integer arithmetic is easy to understand, and there are few surprises. But scientific computing is performed with floating-point numbers, so the FORTRAN programmer has to understand what happens when he uses these numbers in his program. Since the floating-point arithmetic will be performed by the hardware, he has to understand the operation of the floating-point instructions on the machine he is using.

Floating-Point Numbers

In order to allow for differences in machines, we shall designate the set of floating-point numbers by $S(r,p)$, where r is the radix and p is the number of base r digits in the mantissa. Then $S(r,p)$ contains zero and numbers of the form $\pm r^e m$, where

$$r^{-1} \leq m < 1, \tag{1}$$

and $r^p m$ is an integer. The inequality (1) implies that all of the floating-point numbers are normalized, which is a valid assumption for programs compiled by many FORTRAN compilers. The number of bits used to hold the exponent e places a restriction on its size, so it must satisfy a constraint of the form

$$-e_* \leq e \leq e^*. \tag{2}$$

Therefore, there is a largest floating-point number $\Omega = r^{e^*}(1-r^{-p})$ and a smallest floating point number $\omega = r^{(e_*+1)}$.

We are continually faced with the problem of approximating a real number x by a floating-point number. Let x_L be the largest number in $S(r,p)$ which is $\leq x$, and let x_R be the smallest number in $S(r,p)$ which is $\geq x$. Then x_L and x_R are called the *left* and *right neighbors* of x, respectively, and either $x_L < x < x_R$ or else $x_L = x = x_R$. We say that $\tilde{x}$ is the approximation for x obtained by *chopping* x to $S(r,p)$ if $\tilde{x}$ is the neighbor of x with the smaller absolute value. Similarly, $\tilde{x}$ is obtained by *rounding* x to $S(r,p)$ if $\tilde{x}$ is the nieghbor which is closer to x. (If x_L and x_R are equally close to x, a rule must be given to specify which neighbor is to be chosen. The commonest rule is that rounding selects the neighbor with the larger absolute value when the neighbors are equally close to x.)

Floating-Point Arithmetic

The hardware usually provides instructions for floating-point addition, subtraction, multiplication, and division. The sum, difference, product, or quotient of two floating-point numbers may require more than p digits, so the floating-point arithmetic operations produce results that are approximations for the results that would be produced in the real number system. Since these floating-point operations are the basic operations that the programmer uses in his programs, it is important for him to understand what they do. He should be able to think about what happens when these instructions are executed without having to read the microprogram. Therefore, we want to be able to describe the results produced by these instructions in terms that are meaningful to the user.

Suppose that the operands are a and b, and let x be the result that would be produced if the operation were performed in the real number system. Then the floating-point arithmetic will produce an approximation $\tilde{x}$ for x, and our objective is to describe the result $\tilde{x}$ to the user. Some possible statements are:

(a) The result is x rounded to $S(r,p)$.
(b) The result is x chopped to $S(r,p)$.
(c) The result is one of the neighbors of x.
(d) The result does not differ from x by more than k units in the last place. Alternatively, the relative error in the approximation $\tilde{x} \approx x$ is less than, say, 10^{-7}.

Each of these statements is understandable, but statements (a) and (b) are much easier for the user to comprehend and use.

Much of the discussion of floating-point arithmetic is concerned with the accuracy of the results it produces. Here, our emphasis is different. While we want the arithmetic to be accurate, we also want it to be understandable. There are several reasons for this. First, consider the person who is developing the library programs for the computer. Some of these programs, such as the routine that computes sin x, are among the primitive elements that are used by the applications programmer, so they should produce results that are almost as accurate as the results that are produced by the floating-point arithmetic. In order to avoid needless loss of accuracy, the developer of these programs must understand the floating-point arithmetic. Otherwise, he simply can't produce high quality mathematical software. To cite a simple example, he wants to know whether multiplication by a power of the

radix is exact, so that he can choose scale factors that don't introduce errors.

The applications programmer also has to understand the floating-point arithmetic of the machine he is using. Even though he writes his programs in FORTRAN, the basic steps in his program are the floating-point operations, and they will be the operations provided by the hardware. Therefore, he must be aware of the anomalies of the floating-point arithmetic, such as the failure of the cancellation law. For example, since multiplication by 2 is exact only on a binary machine, the FORTRAN statements

```
A=2*X
B=2*Y                    (3)
```

can produce the same value for A and B even though $X \neq Y$.

The primary reason that the details of the floating-point arithmetic operations are important to the applications programmer is that a change of 1 unit in the last place can change the branches that are taken, and any change that affects the flow of the program can have a dramatic effect on the answer. For example, suppose that the statements in (3) are followed by the statements

```
IF(X.LE.Y) GO TO 500     (4)
```

and

```
C=2./(A-B)               (5)
```

If statement (5) is executed, we know that $X \neq Y$, so we might expect A and B to be unequal. But it is possible to have A=B, and this will cause difficulty when we try to divide by A-B in statement (5). Similarly, if we use the statements (3) and (4) and then print A and B, the fact that the same value is printed for A and B does not tell us which branch was taken at statement (4). This can cause us to search for a bug in the wrong part of the program.

Exponent Spill

Another aspect of the floating-point arithmetic is the way the hardware behaves when the answer cannot be represented as a floating-point number without violating the constraint (2). This is called exponent spill. Suppose that x is the number that would be produced if we did not have the constraint (2). The spill is called exponent overflow if $|x| > \Omega$, and it is called exponent underflow if $0 < |x| < \omega$. Now typical values for Ω are on the order of 10^{38}, 10^{75}, or 10^{300}, and ω is on the order of $1/\Omega$. Since this allows us to handle such a large range of numbers, it might appear that we will never encounter exponent spill unless there is a bug in the program and it is running wild. But there are calculations in which we develop very large (or very small) intermediate results, even though the final answer is neither abnormally large nor abnormally small. For example, in the evaluation of the expression $n!e^{-x}$, we can encounter spill in both $n!$ and e^{-x}, even though the final answer may be on the order of 1.

It is important for the hardware to provide a reasonable treatment for exponent spill. Many computers produce an interrupt when exponent spill occurs, and then either the compiler or the operating system must supply an interrupt handler which provides the appropriate treatment for spills. With a versatile interrupt handler, this approach can allow the user to control the way spills are handled. He can ask the interrupt handler to print an error message for each spill, and he can specify whether he wants it to terminate the calculation or to provide an appropriate fixup and allow the calculation to proceed. Ways to use various fixups have been discussed in the literature.[3,5]

Since exponent spill is an exceptional case, we don't want the tests that determine whether spill has occurred to degrade the performance in the normal case in which there is no spill. When spill does occur, we want the system to provide a satisfactory treatment for it, but we are not worried about the speed with which it recovers from the spill.

Other Operations

As the cost of logic drops, increased attention is being directed toward including other operations in the hardware. The usual criterion is cost/performance, and the designer considers whether the improvement in speed justifies the additional cost. But we should also consider the understandability of the operations we add. If the operation is poorly implemented, either it will be wasted because the compilers don't use it, or else the user will be stuck with it.

There are some simple operations, such as reverse divide, that present no problems. The user will be unaware of whether the compiler uses it, and the only question is whether the operation is worth including. Multiply-add is a little bit more subtle. Is the multiply-add command equivalent to a multiply followed by an add? If it is not, a new operation has been introduced, so the user has to know when the compiler will use it. Such an operation may make the results unpredictable, because the programmer doesn't know which operations will be produced by the statements he writes.

It is also reasonable to consider commands for the conversion of data types, such as FLOAT to FIXED. These operations are easy for the user to understand, but we still have to be careful about the details. What happens if the floating-point number is too large to be represented in the fixed-point format? How is the floating-point number to be shortened if it is not an integer? If these details are not handled carefully, the operation will not be useful.

Radix conversion is another candidate for inclusion in the hardware. Although it is easy to understand the conversion of small integers, the conversion of floating-point numbers is more difficult. Should the answer be rounded or chopped? Can we guarentee that it will be accurate to a unit in the last place? How many digits can be specified in the decimal representation of the number? Radix conversion is particularly important because most of the constants in a FORTRAN program are written in the decimal representation, and the programmer can't understand what happens during the execution of his

program unless he understands how these constants will be converted. If the radix conversion is performed by the hardware, it becomes the responsibility of the hardware designer to provide clean conversions.

Functions

When we consider including additional features in the hardware, it is natural to consider the elementary functions, such as sin x or e^x, which are extensively used by the applications programmer. But the implementation of these functions presents some subtle problems.

We shall begin by discussing exponentiation, as called for by the ** operation in FORTRAN. The fact that exponentiation is written as if it were an arithmetic operation has led many users to expect it to be as accurate and as reliable as the floating-point arithmetic is. But it can be very treacherous. Is 3.0**2 equal to 9.0? What about 3.0**2.0 and (-3.0)**2.0? It is natural to use logarithms for the real to real case, and many of the subroutines for the real to real case don't bother to check whether the exponent is the floating-point representation of an integer. Then X**Y is equivalent to the expression EXP(Y*ALOG(X)), so 3.0**2.0 is computed as EXP(2.0*ALOG(3.0)), which can produce a result that is not exactly 9.0. Even more annoying, this approach will not work at all for (-3.0)**2.0, because it would require us to compute the logarithm of a negative number. Consequently, many of the subroutines for exponentiation give an error message for (-3.0)**2.0. Regardless of whether exponentiation is performed by the hardware or the software, great care must be taken with its implementation in order to produce reliable results.[1,2]

We encounter similar problems with other functions, such as e^x or sin x. The domain of the function is the set of values for which the function is defined, so we have an error condition if the argument is outside of the domain. Practical considerations require us to restrict the domain to values of the argument for which the answer can be represented as a floating-point number. For example, $e^x > \Omega$ for rather modest sized values of x, so the domain of the function EXP is the set of arguments for which e^x lies between ω and Ω. Also, we may want to exclude from the domain arguments for which the problem is extremely ill-conditioned. For example, suppose that we are using a decimal machine on which the floating-point numbers have 8 digit mantissas. If the argument for sin x is $x=10^9 x.12345678$, then a change of 1 unit in the last place of x is a change of 10 radians, which is more than a revolution. Therefore, even if we knew that the error in x was less than 1 unit in the last place, we would still have no idea what sin x was. Many of the library programs for the trigonometric functions treat this case as an error, because the answer is so sensitive to noise in the argument that the value is almost never worth computing. Thus, the domain of the function is not as obvious as it appears to be.

Since the elementary functions are among the basic elements we use in constructing our programs, it is important that the values produced for them be very accurate. Ideally, we would like these values to be good to the last bit. But this often requires us to use a few guard bits in computing them. One good reason for implementing these functions in hardware is the possibility of retaining guard bits in their calculation. This would make it possible to produce highly reliable values for these functions.

Conclusion

In order to write reliable programs, the programmer must understand the operations performed by the hardware. Therefore, great care must be taken in the design of these operations to ensure that they can be described to the user in simple terms.

References

1. Clark, N. A., W. J. Cody, and H. Kuki, "Self-Contained Power Routines," in "Mathematical Software" (ed. J. R. Rice), Academic Press, New York, 1971, pp.399-415.
2. Cody, W. J., "Software for Elementary Functions," in "Mathematical Software" (ed. J. R. Rice), Academic Press, New York, 1971, pp.171-186.
3. Kahan, W. "7094 II System Support for Numerical Analysis," SHARE Secretary Distribution, SSD 159, C4537, pp. 1-54.
4. Matula, D. W., "In and Out Conversions," Comm. Assoc. Comput. Mach., 11, 1968, pp.47-50.
5. Sterbenz, P. H., "Floating-Point Computation," Prentice-Hall, 1974.