(The following remarks are extracted from Sun's Numerical Computation Guide Appendix D, an expansion upon David Goldberg's "What Every Computer Scientist Should Know About Floating-Point Arithmetic". Page numbers refer to that document.)

Contents:

Differences Among IEEE 754 Implementations

Note: This section is not part of the published paper. It has been added to clarify certain points and correct possible misconceptions about the IEEE standard that the reader might infer from the paper. This material was not written by David Goldberg, but it appears here with his permission.

The preceding paper has shown that floating-point arithmetic must be implemented carefully, since programmers may depend on its properties for the correctness and accuracy of their programs. In particular, the IEEE standard requires a careful implementation, and it is possible to write useful programs that work correctly and deliver accurate results only on systems that conform to the standard. The reader might be tempted to conclude that such programs should be portable to all IEEE systems. Indeed, portable software would be easier to write if the remark on page 195, "When a program is moved between two machines and both support IEEE arithmetic, then if any intermediate result differs, it must be because of software bugs, not from differences in arithmetic," were true.

Unfortunately, the IEEE standard does not guarantee that the same program will deliver identical results on all conforming systems. Most programs will actually produce different results on different systems for a variety of reasons. For one, most programs involve the conversion of numbers between decimal and binary formats, and the IEEE standard doesn't completely specify the accuracy with which such conversions must be performed. For another, many programs use elementary functions supplied by a system library, and the standard doesn't specify these functions at all. Of course, most programmers know that these features lie beyond the scope of the IEEE standard.

Many programmers may not realize that even a program that uses only the numeric formats and operations prescribed by the IEEE standard can compute different results on different systems. In fact, the authors of the standard intended to allow different

implementations to obtain different results. Their intent is evident in the definition of the term *destination* in the IEEE 754 standard: "A destination may be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values." (IEEE 754-1985, p. 7) In other words, the IEEE standard requires that each result be rounded correctly to the precision of the destination into which it will be placed, but the standard does not require that the precision of that destination be determined by a user's program. Thus, different systems may deliver their results to destinations with different precisions, causing the same program to produce different results (sometimes dramatically so), even though those systems all conform to the standard.

Several of the examples in the preceding paper depend on some knowledge of the way floating-point arithmetic is rounded. In order to rely on examples such as these, a programmer must be able to predict how a program will be interpreted, and in particular, on an IEEE system, what the precision of the destination of each arithmetic operation may be. Alas, the loophole in the IEEE standard's definition of *destination* undermines the programmer's ability to know how a program will be interpreted. Consequently, several of the examples given above, when implemented as apparently portable programs in a high-level language, may not work correctly on IEEE systems that normally deliver results to destinations with a different precision than the programmer expects. Other examples may work, but proving that they work may lie beyond the average programmer's ability.

In this section, we classify existing implementations of IEEE 754 arithmetic based on the precisions of the destination formats they normally use. We then review some examples from the paper to show that delivering results in a wider precision than a program expects can cause it to compute wrong results even though it is provably correct when the expected precision is used. We also revisit one of the proofs in the paper to illustrate the intellectual effort required to cope with unexpected precision even when it doesn't invalidate our programs. These examples show that despite all that the IEEE standard prescribes, the differences it allows among different implementations can prevent us from writing portable, efficient numerical software whose behavior we can accurately predict. To develop such software, then, we must first create programming languages and environments that limit the variability the IEEE standard permits and allow programmers to express the floating-point semantics upon which their programs depend.

Current IEEE 754 Implementations

Current implementations of IEEE 754 arithmetic can be divided into two groups distinguished by the degree to which they support different floating-point formats in

hardware. *Extended-based* systems, exemplified by the Intel x86 family of processors, provide full support for an extended double precision format but only partial support for single and double precision: they provide instructions to load or store data in single and double precision, converting it on-the-fly to or from the extended double format, and they provide special modes (not the default) in which the results of arithmetic operations are rounded to single or double precision even though they are kept in registers in extended double format. (Motorola 68000 series processors round results to both the precision and range of the single or double formats in these modes. Intel x86 and compatible processors round results to the precision of the single or double formats but retain the same range as the extended double format.) *Single/double* systems, including most RISC processors, provide full support for single and double precision formats but no support for an IEEE-compliant extended double precision format. (The IBM POWER architecture provides only partial support for single precision, but for the purpose of this section, we classify it as a single/double system.)

To see how a computation might behave differently on an extended-based system than on a single/double system, consider a C version of the example from page 211:

```
int main() {
    double  q;

    q = 3.0/7.0;
    if (q == 3.0/7.0) printf("Equal\n");
    else printf("Not Equal\n");
    return 0;
}
```

Here the constants 3.0 and 7.0 are interpreted as double precision floating-point numbers, and the expression 3.0/7.0 inherits the `double` data type. On a single/double system, the expression will be evaluated in double precision since that is the most efficient format to use. Thus, `q` will be assigned the value 3.0/7.0 rounded correctly to double precision. In the next line, the expression 3.0/7.0 will again be evaluated in double precision, and of course the result will be equal to the value just assigned to `q`, so the program will print "Equal" as expected.

On an extended-based system, even though the expression 3.0/7.0 has type `double`, the quotient will be computed in a register in extended double format, and thus in the default mode, it will be rounded to extended double precision. When the resulting value is assigned to the variable `q`, however, it may then be stored in memory, and since `q` is declared `double`, the value will be rounded to double precision. In the next line, the expression 3.0/7.0 may again be evaluated in extended precision yielding a result that differs from the double precision value stored in `q`, causing the program to print "Not Equal". Of course, other outcomes are possible, too: the compiler could decide to store and thus round the value of the expression 3.0/7.0 in the second line before comparing it

with `q`, or it could keep `q` in a register in extended precision without storing it. An optimizing compiler might evaluate the expression 3.0/7.0 at compile time, perhaps in double precision or perhaps in extended double precision. (With one x86 compiler, the program prints "Equal" when compiled with optimization and "Not Equal" when compiled for debugging.) Finally, some compilers for extended-based systems automatically change the rounding precision mode to cause operations producing results in registers to round those results to single or double precision, albeit possibly with a wider range. Thus, on these systems, we can't predict the behavior of the program simply by reading its source code and applying a basic understanding of IEEE 754 arithmetic. Neither can we accuse the hardware or the compiler of failing to provide an IEEE 754 compliant environment; the hardware has delivered a correctly rounded result to each destination, as it is required to do, and the compiler has assigned some intermediate results to destinations that are beyond the user's control, as it is allowed to do.

Pitfalls in Computations on Extended-Based Systems

Conventional wisdom maintains that extended-based systems must produce results that are at least as accurate, if not more accurate than those delivered on single/double systems, since the former always provide at least as much precision and often more than the latter. Trivial examples such as the C program above as well as more subtle programs based on the examples discussed below show that this wisdom is naive at best: some apparently portable programs, which are indeed portable across single/double systems, deliver incorrect results on extended-based systems precisely because the compiler and hardware conspire to occasionally provide more precision than the program expects.

Current programming languages make it difficult for a program to specify the precision it expects. As the section "Languages and Compilers" on page 214 mentions, many programming languages don't specify that each occurrence of an expression like `10.0*x` in the same context should evaluate to the same value. Some languages, such as Ada, were influenced in this respect by variations among different arithmetics prior to the IEEE standard. More recently, languages like ANSI C have been influenced by standard-conforming extended-based systems. In fact, the ANSI C standard explicitly allows a compiler to evaluate a floating-point expression to a precision wider than that normally associated with its type. As a result, the value of the expression `10.0*x` may vary in ways that depend on a variety of factors: whether the expression is immediately assigned to a variable or appears as a subexpression in a larger expression; whether the expression participates in a comparison; whether the expression is passed as an argument to a function, and if so, whether the argument is passed by value or by reference; the current precision mode; the level of optimization at which the program was compiled; the precision mode and expression evaluation method used by the compiler when the program was compiled; and so on.

Language standards are not entirely to blame for the vagaries of expression evaluation. Extended-based systems run most efficiently when expressions are evaluated in extended precision registers whenever possible, yet values that must be stored are stored in the narrowest precision required. Constraining a language to require that `10.0*x` evaluate to the same value everywhere would impose a performance penalty on those systems. Unfortunately, allowing those systems to evaluate `10.0*x` differently in syntactically equivalent contexts imposes a penalty of its own on programmers of accurate numerical software by preventing them from relying on the syntax of their programs to express their intended semantics.

Do real programs depend on the assumption that a given expression always evaluates to the same value? Recall the algorithm presented in Theorem 4 for computing $\ln(1 + x)$, written here in Fortran:

```
real function log1p(x)
real x
if (1.0 + x .eq. 1.0) then
   log1p = x
else
   log1p = log(1.0 + x) * x / ((1.0 + x) - 1.0)
endif
return
```

On an extended-based system, a compiler may evaluate the expression `1.0 + x` in the third line in extended precision and compare the result with `1.0`. When the same expression is passed to the log function in the sixth line, however, the compiler may store its value in memory, rounding it to single precision. Thus, if `x` is not so small that `1.0 + x` rounds to `1.0` in extended precision but small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` will be zero instead of `x`, and the relative error will be one---rather larger than five rounding errors. Similarly, suppose the rest of the expression in the sixth line, including the reoccurrence of the subexpression `1.0 + x`, is evaluated in extended precision. In that case, if `x` is small but not quite small enough that `1.0 + x` rounds to `1.0` in single precision, then the value returned by `log1p(x)` can exceed the correct value by nearly as much as `x`, and again the relative error can approach one. For a concrete example, take `x` to be $2^{-24} + 2^{-47}$, so `x` is the smallest single precision number such that `1.0 + x` rounds up to the next larger number, $1 + 2^{-23}$. Then `log(1.0 + x)` is approximately 2^{-23}. Because the denominator in the expression in the sixth line is evaluated in extended precision, it is computed exactly and delivers `x`, so `log1p(x)` returns approximately 2^{-23}, which is nearly twice as large as the exact value. (This actually happens with at least one compiler. When the preceding code is compiled by the Sun WorkShop Compilers 4.2.1 Fortran 77 compiler for x86 systems using the `-O` optimization flag, the generated code computes `1.0 + x` exactly as

described. As a result, the code delivers zero for `log1p(1.0e-10)` and `1.19209E-07` for `log1p(5.97e-8)`.)

For the algorithm of Theorem 4 to work correctly, the expression `1.0 + x` must be evaluated the same way each time it appears; the algorithm can fail on extended-based systems only when `1.0 + x` is evaluated to extended double precision in one instance and to single or double precision in another. Of course, since `log` is a generic intrinsic function in Fortran, a compiler could evaluate the expression `1.0 + x` in extended precision throughout, computing its logarithm in the same precision, but evidently we cannot assume that the compiler will do so. (One can also imagine a similar example involving a user-defined function. In that case, a compiler could still keep the argument in extended precision even though the function returns a single precision result, but few if any existing Fortran compilers do this, either.) We might therefore attempt to ensure that `1.0 + x` is evaluated consistently by assigning it to a variable. Unfortunately, if we declare that variable `real`, we may still be foiled by a compiler that substitutes a value kept in a register in extended precision for one appearance of the variable and a value stored in memory in single precision for another. Instead, we would need to declare the variable with a type that corresponds to the extended precision format. Standard FORTRAN 77 does not provide a way to do this, and while Fortran 90 offers the `SELECTED_REAL_KIND` mechanism for describing various formats, it does not explicitly require implementations that evaluate expressions in extended precision to allow variables to be declared with that precision. In short, there is no portable way to write this program in standard Fortran that is guaranteed to prevent the expression `1.0 + x` from being evaluated in a way that invalidates our proof.

There are other examples that can malfunction on extended-based systems even when each subexpression is stored and thus rounded to the same precision. The cause is *double-rounding*. In the default precision mode, an extended-based system will initially round each result to extended double precision. If that result is then stored to double precision, it is rounded again. The combination of these two roundings can yield a value that is different than what would have been obtained by rounding the first result correctly to double precision. This can happen when the result as rounded to extended double precision is a "halfway case", i.e., it lies exactly halfway between two double precision numbers, so the second rounding is determined by the round-ties-to-even rule. If this second rounding rounds in the same direction as the first, the net rounding error will exceed half a unit in the last place. (Note, though, that double-rounding only affects double precision computations. One can prove that the sum, difference, product, or quotient of two p-bit numbers, or the square root of a p-bit number, rounded first to q bits and then to p bits gives the same value as if the result were rounded just once to p bits provided $q >= 2p + 2$. Thus, extended double precision is wide enough that single precision computations don't suffer double-rounding.)

Some algorithms that depend on correct rounding can fail with double-rounding. In fact, even some algorithms that don't require correct rounding and work correctly on a variety of machines that don't conform to IEEE 754 can fail with double-rounding. The most useful of these are the portable algorithms for performing simulated multiple precision arithmetic mentioned on page 186. For example, the procedure described in Theorem 6 for splitting a floating-point number into high and low parts doesn't work correctly in double-rounding arithmetic: try to split the double precision number $2^{52} + 3 \times 2^{26} - 1$ into two parts each with at most 26 bits. When each operation is rounded correctly to double precision, the high order part is $2^{52} + 2^{27}$ and the low order part is $2^{26} - 1$, but when each operation is rounded first to extended double precision and then to double precision, the procedure produces a high order part of $2^{52} + 2^{28}$ and a low order part of $-2^{26} - 1$. The latter number occupies 27 bits, so its square can't be computed exactly in double precision. Of course, it would still be possible to compute the square of this number in extended double precision, but the resulting algorithm would no longer be portable to single/double systems. Also, later steps in the multiple precision multiplication algorithm assume that all partial products have been computed in double precision. Handling a mixture of double and extended double precision variables correctly would make the implementation significantly more expensive.

Likewise, portable algorithms for adding multiple precision numbers represented as arrays of double precision numbers can fail in double-rounding arithmetic. These algorithms typically rely on a technique similar to Kahan's summation formula. As the informal explanation of the summation formula given on page 239 suggests, if `s` and `y` are floating-point variables with |`s`| >= |`y`| and we compute:

```
t = s + y;
e = (s - t) + y;
```

then in most arithmetics, `e` recovers exactly the roundoff error that occurred in computing `t`. This technique doesn't work in double-rounded arithmetic, however: if `s` = $2^{52} + 1$ and `y` = $1/2 - 2^{-54}$, then `s + y` rounds first to $2^{52} + 3/2$ in extended double precision, and this value rounds to $2^{52} + 2$ in double precision by the round-ties-to-even rule; thus, the net rounding error in computing `t` is $1/2 + 2^{-54}$, which isn't representable exactly in double precision and so can't be computed exactly by the expression shown above. Here again, it would be possible to recover the roundoff error by computing the sum in extended double precision, but then a program would have to do extra work to reduce the final outputs back to double precision, and double-rounding could afflict this process, too. For this reason, although portable programs for simulating multiple precision arithmetic by these methods work correctly and efficiently on a wide variety of machines, they don't work as advertised on extended-based systems.

Finally, some algorithms that at first sight appear to depend on correct rounding may in fact work correctly with double-rounding. In these cases, the cost of coping with double-rounding lies not in the implementation but in the verification that the algorithm works as advertised. To illustrate, we prove the following variant of Theorem 7:

Theorem 7'

If m and n are integers representable in IEEE 754 double precision with $|m| < 2^{52}$ and n has the special form $n = 2^i + 2^j$, then $\mathrm{fl}(\mathrm{fl}(m / n) \times n) = m$, provided both floating-point operations are either rounded correctly to double precision or rounded first to extended double precision and then to double precision.

Proof

Assume without loss that $m > 0$. Let $q = \mathrm{fl}(m / n)$. Scaling by powers of two, we can consider an equivalent setting in which $2^{52} <= m < 2^{53}$ and likewise for q, so that both m and q are integers whose least significant bits occupy the units place (i.e., $\mathrm{ulp}(m) = \mathrm{ulp}(q) = 1$). Before scaling, we assumed $m < 2^{52}$, so after scaling, m is an even integer. Also, because the scaled values of m and q satisfy $m/2 < q < 2m$, the corresponding value of n must have one of two forms depending on which of m or q is larger: if $q < m$, then evidently $1 < n < 2$, and since n is a sum of two powers of two, $n = 1 + 2^{-k}$ for some k; similarly, if $q > m$, then $1/2 < n < 1$, so $n = 1/2 + 2^{-(k+1)}$. (As n is the sum of two powers of two, the closest possible value of n to one is $n = 1 + 2^{-52}$. Because $m/(1 + 2^{-52})$ is no larger than the next smaller double precision number less than m, we can't have $q = m$.)

Let e denote the rounding error in computing q, so that $q = m/n + e$, and the computed value $\mathrm{fl}(q \times n)$ will be the (once or twice) rounded value of $m + ne$. Consider first the case in which each floating-point operation is rounded correctly to double precision. In this case, $|e| < 1/2$. If n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of $2^{-(k+1)}$ and $|ne| < 1/4 + 2^{-(k+2)}$. This implies that $|ne| <= 1/4$. Recall that the difference between m and the next larger representable number is 1 and the difference between m and the next smaller representable number is either 1 if $m > 2^{52}$ or 1/2 if $m = 2^{52}$. Thus, as $ne <= 1/4$, $m + ne$ will round to m. (Even if $m = 2^{52}$ and $ne = -1/4$, the product will round to m by the round-ties-to-even rule.) Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of 2^{-k} and $|ne| < 1/2 + 2^{-(k+1)}$; this implies $|ne| <= 1/2$. We can't have $m = 2^{52}$ in this case because m is strictly greater than q, so m differs from its nearest representable neighbors by $+/-1$. Thus, as $|ne| <= 1/2$, again $m + ne$ will round to m. (Even if $|ne| = 1/2$, the product will round to m by the round-ties-to-even rule because m is even.) This completes the proof for correctly rounded arithmetic.

In double-rounding arithmetic, it may still happen that q is the correctly rounded quotient (even though it was actually rounded twice), so $|e| < 1/2$ as above. In this case, we can appeal to the arguments of the previous paragraph provided we consider the fact that $\mathrm{fl}(q \times n)$ will be rounded twice. To account for this, note that the IEEE standard requires that an extended double format carry at least 64 significant bits, so that the numbers $m \pm 1/2$ and $m \pm 1/4$ are exactly representable in extended double precision. Thus, if n has the form $1/2 + 2^{-(k+1)}$, so that $|ne| \leq 1/4$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/4, and as noted above, this value will round to m in double precision. Similarly, if n has the form $1 + 2^{-k}$, so that $|ne| \leq 1/2$, then rounding $m + ne$ to extended double precision must produce a result that differs from m by at most 1/2, and this value will round to m in double precision. (Recall that $m > 2^{52}$ in this case.)

Finally, we are left to consider cases in which q is not the correctly rounded quotient due to double-rounding. In these cases, we have $|e| < 1/2 + 2^{-(d+1)}$ in the worst case, where d is the number of extra bits in the extended double format. (All existing extended-based systems support an extended double format with exactly 64 significant bits; for this format, $d = 64 - 53 = 11$.) Because double-rounding only produces an incorrectly rounded result when the second rounding is determined by the round-ties-to-even rule, q must be an even integer. Thus, if n has the form $1/2 + 2^{-(k+1)}$, then $ne = nq - m$ is an integer multiple of 2^{-k}, and $|ne| < (1/2 + 2^{-(k+1)})(1/2 + 2^{-(d+1)}) = 1/4 + 2^{-(k+2)} + 2^{-(d+2)} + 2^{-(k+d+2)}$. If $k \leq d$, this implies $|ne| \leq 1/4$. If $k > d$, we have $|ne| \leq 1/4 + 2^{-(d+2)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/4, and by previous arguments, the second rounding will round to m. Similarly, if n has the form $1 + 2^{-k}$, then ne is an integer multiple of $2^{-(k-1)}$, and $|ne| < 1/2 + 2^{-(k+1)} + 2^{-(d+1)} + 2^{-(k+d+1)}$. If $k \leq d$, this implies $|ne| \leq 1/2$. If $k > d$, we have $|ne| \leq 1/2 + 2^{-(d+1)}$. In either case, the first rounding of the product will deliver a result that differs from m by at most 1/2, and again by previous arguments, the second rounding will round to m. #

The preceding proof shows that the product can incur double-rounding only if the quotient does, and even then, it rounds to the correct result. The proof also shows that extending our reasoning to include the possibility of double-rounding can be challenging even for a program with only two floating-point operations. For a more complicated program, it may be impossible to systematically account for the effects of double-rounding, not to mention more general combinations of double and extended double precision computations.

Programming Language Support for Extended Precision

The preceding examples should not be taken to suggest that extended precision *per se* is harmful. Many programs can benefit from extended precision when the programmer is

able to use it selectively. Unfortunately, current programming languages do not provide sufficient means for a programmer to specify when and how extended precision should be used. To indicate what support is needed, we consider the ways in which we might want to manage the use of extended precision.

In a portable program that uses double precision as its nominal working precision, there are five ways we might want to control the use of a wider precision:

1. Compile to produce the fastest code, using extended precision where possible on extended-based systems. Clearly most numerical software does not require more of the arithmetic than that the relative error in each operation is bounded by the "machine epsilon". When data in memory are stored in double precision, the machine epsilon is usually taken to be the largest relative roundoff error in that precision, since the input data are (rightly or wrongly) assumed to have been rounded when they were entered and the results will likewise be rounded when they are stored. Thus, while computing some of the intermediate results in extended precision may yield a more accurate result, extended precision is not essential. In this case, we might prefer that the compiler use extended precision only when it will not apprciably slow the program and use double precision otherwise.

2. Use a format wider than double if it is reasonably fast and wide enough, otherwise resort to something else. Some computations can be performed more easily when extended precision is available, but they can also be carried out in double precision with only somewhat greater effort. Consider computing the Euclidean norm of a vector of double precision numbers. By computing the squares of the elements and accumulating their sum in an IEEE 754 extended double format, with its wider exponent range, we can trivially avoid premature underflow or overflow for vectors of practical lengths. On extended-based systems, this is the fastest way to compute the norm. On single/double systems, an extended double format would have to be emulated in software (if one were supported at all), and such emulation would be much slower than simply using double precision, testing the exception flags to determine whether underflow or overflow occurred, and if so, repeating the computation with explicit scaling. Note that to support this use of extended precision, a language must provide both an indication of the widest available format that is reasonably fast, so that a program can choose which method to use, and environmental parameters that indicate the precision and range of each format, so that the program can verify that the widest fast format is wide enough (e.g., that it has wider range than double).

3. Use a format wider than double even if it has to be emulated in software. For more complicated programs than the Euclidean norm example, the programmer may simply wish to avoid the need to write two versions of the program and instead rely on extended precision even if it is slow. Again, the language must provide

environmental parameters so that the program can determine the range and precision of the widest available format.

4. Don't use a wider precision; round results correctly to the precision of the double format, albeit possibly with extended range. For programs that are most easily written to depend on correctly rounded double precision arithmetic, including some of the examples mentioned above, a language must provide a way for the programmer to indicate that extended precision must not be used, even though intermediate results may be computed in registers with a wider exponent range than double. (Intermediate results computed in this way can still incur double-rounding if they underflow when stored to memory: if the result of an arithmetic operation is rounded first to 53 significant bits, then rounded again when it must be denormalized, the final result may differ from what would have been obtained by rounding just once to a denormalized number. Of course, this form of double-rounding is highly unlikely to affect any practical program adversely.)

5. Round results correctly to both the precision and range of the double format. This strict enforcement of double precision would be most useful for programs that test either numerical software or the arithmetic itself near the limits of both the range and precision of the double format. Such careful test programs tend to be difficult to write in a portable way; they become even more difficult (and error prone) when they must employ dummy subroutines and other tricks to force results to be rounded to a particular format. Thus, a programmer using an extended-based system to develop robust software that must be portable to all IEEE 754 implementations would quickly come to appreciate being able to emulate the arithmetic of single/double systems without extraordinary effort.

No current language supports all five of these options. In fact, few languages have attempted to give the programmer the ability to control the use of extended precision at all. One notable exception is C9X, the latest revision to the C language, which is now in the final stages of standardization.

Like the current C standard, C9X allows an implementation to evaluate expressions in a format wider than that normally associated with their type, but C9X recommends using one of only three expression evaluation methods. The three recommended methods are characterized by the extent to which expressions are "promoted" to wider formats, and the implementation is encouraged to identify which method it uses by defining the preprocessor macro `FLT_EVAL_METHOD`: if `FLT_EVAL_METHOD` is 0, each expression is evaluated in a format that corresponds to its type; if `FLT_EVAL_METHOD` is 1, `float` expressions are promoted to the format that corresponds to `double`; and if `FLT_EVAL_METHOD` is 2, `float` and `double` expressions are promoted to the format that corresponds to `long double`. (An implementation is allowed to set `FLT_EVAL_METHOD` to -1 to indicate that the expression evaluation method is indeterminable.) C9X also requires

that the <math.h> header file define the types float_t and double_t, which are at least as wide as float and double, respectively, and are intended to match the types used to evaluate float and double expressions. For example, if FLT_EVAL_METHOD is 2, both float_t and double_t are long double. Finally, C9X requires that the <float.h> header file define preprocessor macros that specify the range and precision of the formats corresponding to each floating-point type.

The combination of features required or recommended by C9X supports some of the five options listed above but not all. For example, if an implementation maps the long double type to an extended double format and defines FLT_EVAL_METHOD to be 2, the programmer can reasonably assume that extended precision is relatively fast, so programs like the Euclidean norm example can simply use intermediate variables of type long double (or double_t). On the other hand, the same implementation must keep anonymous expressions in extended precision even when they are stored in memory (e.g., when the compiler must spill floating-point registers), and it must store the results of expressions assigned to variables declared double to convert them to double precision even if they could have been kept in registers. Thus, neither the double nor the double_t type can be compiled to produce the fastest code on current extended-based hardware.

Likewise, C9X provides solutions to some of the problem illustrated by the examples in this section but not all. A C9X version of the log1p function is guaranteed to work correctly if the expression 1.0 + x is assigned to a variable (of any type) and that variable used throughout. A portable, efficient C9X program for splitting a double precision number into high and low parts, however, is more difficult: how can we split at the correct position and avoid double-rounding if we cannot guarantee that double expressions are rounded correctly to double precision? One solution is to use the double_t type to perform the splitting in double precision on single/double systems and in extended precision on extended-based systems, so that in either case the arithmetic will be correctly rounded. Theorem 14 says that we can split at any bit position provided we know the precision of the underlying arithmetic, and the FLT_EVAL_METHOD and environmental parameter macros should give us this information. The following fragment shows one possible implementation:

```
#include <math.h>
#include <float.h>

#if (FLT_EVAL_METHOD==2)
#define PWR2  LDBL_MANT_DIG - (DBL_MANT_DIG/2)
#elif ((FLT_EVAL_METHOD==1) || (FLT_EVAL_METHOD==0))
#define PWR2  DBL_MANT_DIG - (DBL_MANT_DIG/2)
#else
#error FLT_EVAL_METHOD unknown!
#endif
```

```
...
    double   x, xh, xl;
    double_t m;

    m = scalbn(1.0, PWR2) + 1.0;  // 2**PWR2 + 1
    xh = (m * x) - ((m * x) - x);
    xl = x - xh;
```

Of course, to find this solution, the programmer must know that `double` expressions may be evaluated in extended precision, that the ensuing double-rounding problem can cause the algorithm to malfunction, and that extended precision may be used instead according to Theorem 14. A more obvious solution is simply to specify that each expression be rounded correctly to double precision. On extended-based systems, this merely requires changing the rounding precision mode, but unfortunately, C9X does not provide a portable way to do this. (Early drafts of the Floating-Point C Edits, the working document that specified the changes to be made to the C standard to support floating-point, recommended that implementations on systems with rounding precision modes provide `fegetprec` and `fesetprec` functions to get and set the rounding precision, analogous to the `fegetround` and `fesetround` functions that get and set the rounding direction. This recommendation was removed before the changes were made to the C9X draft.)

Coincidentally, C9X's approach to supporting portability among systems with different integer arithmetic capabilities suggests a better way to support different floating-point architectures. Each C9X implementation supplies an `<inttypes.h>` header file that defines those integer types the implementation supports, named according to their sizes and efficiency: for example, `int32_t` is an integer type exactly 32 bits wide, `int_fast16_t` is the implementation's fastest integer type at least 16 bits wide, and `intmax_t` is the widest integer type supported. One can imagine a similar scheme for floating-point types: for example, `float53_t` could name a floating-point type with exactly 53 bit precision but possibly wider range, `float_fast24_t` could name the implementation's fastest type with at least 24 bit precision, and `floatmax_t` could name the widest reasonably fast type supported. The fast types could allow compilers on extended-based systems to generate the fastest possible code subject only to the constraint that the values of named variables must not appear to change as a result of register spilling. The exact width types would cause compilers on extended-based systems to set the rounding precision mode to round to the specified precision, allowing wider range subject to the same constraint. Finally, `double_t` could name a type with both the precision and range of the IEEE 754 double format, providing strict double evaluation. Together with environmental parameter macros named accordingly, such a scheme would readily support all five options described above and allow programmers to indicate easily and unambiguously the floating-point semantics their programs require.

Must language support for extended precision be so complicated? On single/double systems, four of the five options listed above coincide, and there is no need to differentiate

fast and exact width types. Extended-based systems, however, pose difficult choices: they support neither pure double precision nor pure extended precision computation as efficiently as a mixture of the two, and different programs call for different mixtures. Moreover, the choice of when to use extended precision should not be left to compiler writers, who are often tempted by benchmarks (and sometimes told outright by numerical analysts) to regard floating-point arithmetic as "inherently inexact" and therefore neither deserving nor capable of the predictability of integer arithmetic. Instead, the choice must be presented to programmers, and they will require languages capable of expressing their selection.

Conclusion

The foregoing remarks are not intended to disparage extended-based systems but to expose several fallacies, the first being that all IEEE 754 systems must deliver identical results for the same program. We have focused on differences between extended-based systems and single/double systems, but there are further differences among systems within each of these families. For example, some single/double systems provide a single instruction to multiply two numbers and add a third with just one final rounding. This operation, called a *fused multiply-add*, can cause the same program to produce different results across different single/double systems, and, like extended precision, it can even cause the same program to produce different results on the same system depending on whether and when it is used. (A fused multiply-add can also foil the splitting process of Theorem 6, although it can also be used in a non-portable way to perform multiple precision multiplication without the need for splitting.) Even though the IEEE standard didn't anticipate such an operation, it nevertheless conforms: the intermediate product is delivered to a "destination" beyond the user's control that is wide enough to hold it exactly, and the final sum is rounded correctly to fit its single or double precision destination.

The idea that IEEE 754 prescribes precisely the result a given program must deliver is nonetheless appealing. Many programmers like to believe that they can understand the behavior of a program and prove that it will work correctly without reference to the compiler that compiles it or the computer that runs it. In many ways, supporting this belief is a worthwhile goal for the designers of computer systems and programming languages. Unfortunately, when it comes to floating-point arithmetic, the goal is virtually impossible to achieve. The authors of the IEEE standards knew that, and they didn't attempt to achieve it. As a result, despite nearly universal conformance to (most of) the IEEE 754 standard throughout the computer industry, programmers of portable software must continue to cope with unpredictable floating-point arithmetic.

If programmers are to exploit the features of IEEE 754, they will need programming languages that make floating-point arithmetic predictable. C9X improves predictability to

some degree at the expense of requiring programmers to write multiple versions of their programs, one for each `FLT_EVAL_METHOD`. Whether future languages will choose instead to allow programmers to write a single program with syntax that unambiguously expresses the extent to which it depends on IEEE 754 semantics remains to be seen. Existing extended-based systems threaten that prospect by tempting us to assume that the compiler and hardware can know better than the programmer how a computation should be performed on a given system. That assumption is the second fallacy: the accuracy required in a computed result depends not on the machine that produces it but only on the conclusions that will be drawn from it, and of the programmer, the compiler, and the hardware, at best only the programmer can know what those conclusions may be.