Specifications for a Variable-Precision Arithmetic Coprocessor

T.E. Hull, M.S. Cohen* and C.B. Hall**

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

The authors have been developing a programming system which is intended to be especially convenient for scientific computing. Its main features are variable precision (decimal) floating-point arithmetic and convenient exception handling. The software implementation of the system has evolved over a number of years, and a partial hardware implementation of the arithmetic itself was constructed and used during the early stages of the project. Based on this experience, the authors have developed a set of specifications for an arithmetic coprocessor to support such a system. The main purpose of this paper is to describe these specifications. An outline of the language features and how they can be used is also provided, to help justify our particular choice of coprocessor specifications.

Introduction

The general purpose of our project is to provide better programming language facilities for scientific computing, especially in terms of precision control and exception handling. Early efforts were concerned primarily with precision control and led to the development of preprocessors, both for Algol and Fortran, which made it easy to change precision dynamically between single, double, triple, etc., precisions. Then an attached processor, called CADAC, which carried out variable precision decimal arithmetic was built and attached to a Vax [3,4]. Finally, the present software system, which implements the programming language Numerical Turing (NT) and which provides for variable-precision decimal arithmetic and exception handling, was developed [7,8,9]; special arithmetic capabilities are also included, such as directed roundings and exponent manipulations. It runs under Unix on either a Vax or a Sun 3.

Based on experience with these developments, the language specifications have of course evolved. But they are now at a stage where we have some confidence in their appropriateness, and believe that it would be useful to design and build the necessary hardware support. The main purpose of this paper is to describe the corresponding specifications, for the arithmetic unit, and also to indicate what other hardware features would be most helpful to the systems programmer, especially for implementation of the exception handling.

Precision control

We first describe the floating-point arithmetic (the real arithmetic) and related functions. In this section we do so in terms of the programming language, that is, from the user's point of view. Implications for the hardware will be considered in the next section.

Real values are p-digit, decimal, normalized floating-point numbers with exponents in the range $[-10p, 10p]$, where $1 \leq p \leq$ *maxprecision*. (In NT the parameter *maxprecision* is only 200, but it should be much larger, at least 1000.)

A programmer specifies what precision is to prevail by means of a precision statement. In NT the precision statement is of the form

precision *intexp*

where *intexp* is any integer expression. The specified precision is to prevail throughout the scope of the precision statement. For example, in Figure 1, the value of p determines the precision in effect from immediately after the precision statement to the end of the loop.

All declarations and floating-point operations (arithmetic, elementary functions, procedures, etc.) within this scope are carried out in precision p. In this example

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

* Present address: 141 North Meadow Crescent, Thornhill, Ontario, Canada L4J 3C4.

** Present address: IBM Canada Limited, 844 Don Mills Road, North York, Ontario, Canada M3C 1V7.

```
----
----
var x : real
var p := currentprecision
loop
   precision p
   var y : real
   ----
   ----
   solve( ----, y)
   if y ---- then
      x := y
      exit
   end if
   p := p+10
end loop
----
```

Figure 1. All declarations and floating-point operations (arithmetic, elementary functions, procedures, etc.) within the scope of the precision statement are carried out in precision *p*.

p is first set equal to the value of the current precision (in NT *currentprecision* is a function which returns this value). The value of *p* is then increased by 10 each time the loop is executed until the condition in the *if* statement is *true*. Then *y* is assigned to *x*, and, if the precision of *y* is higher than that of *x*, the value of *y* will have to be rounded to the precision of *x* before being assigned. There must be a default precision in case no precision statement is yet in effect (in NT it is 16).

Any value in any expression that appears within the scope of a precision statement must be rounded or, in effect, extended to the prevailing precision before it can be used, if its precision differs from that of the prevailing precision.

This example illustrates one way in which precision control can be used. The procedure *solve* is executed in higher and higher precision until some criterion, presumably an accuracy criterion, is satisfied.

Basically this same idea can be implemented in Aberth's system [1], where the calculations inside the loop would be done in interval arithmetic and the decision to exit is then based on whether the interval associated with the final result is small enough. This is possible also in NT with the help of the directed roundings. (In a future version of NT, it is intended to have interval arithmetic "built-in", but in either case the implications for the coprocessor are the same: directed roundings must be supported.)

A second example is shown in Figure 2. In this example the increase in precision is used primarily to make sure that no intermediate overflow or underflow can

```
---- % n and the array a must
---- % be known at this point
var x : real
begin
   precision 2*currentprecision + getexp(n) + 1
   var sum := 0.0
   for i : 1..n
      sum := sum + a(i)*a(i)
   end for
   x := sqrt(sum)
end
----
```

Figure 2. This program fragment computes the Euclidean norm of the array *a*. The precision is increased just enough so that no intermediate overflow or undrflow can occur. Overflow may however occur in the final assignment to *x*.

take place. (The exponent range increases with the precision.) In fact, the precision is increased by the minimum amount that is needed to guarantee that no intermediate overflow or underflow can occur. (The function *getexp* returns the exponent of its argument, which, for the purpose of this example, means that $getexp(n) + 1 = ceil(\log_{10} n)$.) Of course, overflow might occur in the final assignment, because the calculated value to be assigned must, at that point, be rounded to the precision of *x* before being assigned to *x*.

In a third example one might use precision control to run a calculation in two different precisions and subtract the final results to measure the cumulative effect of rounding errors. (Of course, care must be taken in the use of any such technique; otherwise the results could be misleading.)

In another example, the "exact dot product", which plays an important role in ACRITH [12], can be implemented in NT. The maximum precision must of course be higher than it is in the current implementation of NT, and this should be kept in mind in developing specifications for the coprocessor.

The available floating-point operations in all these and other examples must of course include addition, subtraction, multiplication and division. The rounding in these operations is important and it should be unbiased. (In NT it is "round to nearest, or nearest even in case of a tie", which is what is done in IEEE arithmetic [10,11], but we are now considering a simpler rule, as described in the next section.) The same rounding rule should hold for the implicit rounding that occurs when a higher precision value must be rounded down (coerced) to some prevailing current precision, or when an assignment to some lower precision variable is about to be made.

As already indicated, directed roundings (round up and round down) should also be available with the four basic arithmetic operations. The functions *floor*, *ceil*, and *round* (the latter being to nearest integer in case of a tie) should also be available.

Other functions are required to determine quotients and remainders, and also to determine and set exponents, and to convert from integer to real. Of course comparison operators are also needed.

The operations and functions just described are sufficient for the construction of other functions which might be required, especially the elementary functions. In fact, the elementary functions can be programmed using these operations and functions in a quite straightforward and easily understood manner, especially when it is possible to increase the precision at appropriate stages in the calculations [5,6].

Other helpful functions, such as the *nextafter* function recommended in the IEEE standard, can also be programmed quite easily with those described above, but it may be that it is convenient to build some of them directly in the hardware.

Besides the functionality of the language, the user's primary concern is efficiency. Most of the computation will be in the default precision, which of course must therefore be as efficient as possible. But the other most frequently used precisions are only at most a few digits more than the default precision. For example, when dot products are accumulated in higher precision they will usually need only about two extra digits of precision [6]. The requirements of the elementary functions are more variable, but most of the time only two or three extra digits are required to deliver results to within an error of less than one unit in the last place [5,6 for example]. It would therefore be desirable that precisions slightly more than the default precision be the second consideration in terms of efficiency.

The next most frequently used precisions are double precision, and occasionally slightly more than double. An example was shown in Figure 2. These precisions should therefore be given third priority.

To support the ACRITH "exact dot product" facility, some consideration should also be given to making efficient the accumulation of dot products in a very long "accumulator".

It should be acknowledged again that the higher precisions will not be used very much of the time and the overall efficiency of a program will therefore usually not be seriously affected if efficiency in these precisions is not very great.

In our experience, high precision calculations are quite often used only as test programs. They are used to find the "true" solutions of some problems (matrix calculations, or elementary function evaluations, for example) in order to test the accuracy of programs running in some standard precision. In such cases, the high precision program needs to be run only once for each test problem, whereas there may be many test programs or variations on a test program to be run on the same problem.

From another point of view, we have also found that the availability of higher precision has often made our programming more efficient. In such cases using higher precision has enabled us to handle a problem in a direct and relatively simple way, rather than in a roundabout and relatively convoluted manner.

Arithmetic specifications

The functionalities described in the previous section, provide all the requirements to be met by a hardware floating-point unit, apart from what happens when exceptions occur. Some further details about the arithmetic, especially with regard to rounding, will be discussed in this section, and exception handling will be considered in the next section.

It might be helpful to consider a possible representation of floating-point numbers. We are not committed to the following representation, but we did use it recently in some preliminary design investigations. We used 4 bytes to store the following:

sign	1 bit	
exponent	≥ 15 bits	(for exponents at least in [-10000,10000])
precision	≥ 10 bits	(for precisions at least in [1,1000])
extended	1 bit	(to indicate whether or not the format is extended)
uninitialized	1 bit	(to indicate whether or not the value has not yet been assigned)

and this was followed by a sufficient number of 4-byte words to represent the normalized significand (4 bits for each decimal digit).

This representation is more compact than what is used by the software implementation in NT, but is otherwise equivalent. (The "extended" bit is there to allow for the possibility of having an alternative format for even higher precisions, which could be handled entirely in software, but we have not yet tried to take advantage of this idea, either in the original CADAC design or in the present software implementation.)

Precision specifications for the arithmetic operations, directed roundings, quotient and remainder, etc., have already been spelled out in detail elsewhere [6], so they will not be repeated here. The only new possibility has to do with rounding, which up until now has always followed the IEEE rule: "round to nearest, or nearest even in case of a tie." We are now considering another rule, like one suggested by von Neumann [2, pp 57-58], which is easier to implement and is, at the same time, also "unbiased". The rule is:

> "if there is something non-zero following the least significant digit in the true result, set the last bit to 1 (so that the last digit will be odd) and throw away the "something", otherwise just throw away the zeros following the least significant digit".

With this rule, there is no possibility of a carry operation, or any subsequent renormalization. The only disadvantage is that the maximum error can be almost 1 in the last place, compared to a maximum of 1/2 in the last place with the earlier rule. But this disadvantage seems to us to be one we need not worry about in a variable precision environment where it is such a simple matter to increase precision whenever one wishes. In fact, the elementary functions can also be made accurate to within an error of less than 1 unit in the last place (just as they are in the current implementation of NT), so that the elementary functions *and* the arithmetic operations would then both meet the same accuracy requirement.

Note that the requirement is *less than* 1 unit, not *less than or equal to* 1 unit. This is important because we can then deduce some convenient results, such as, for example, that the square root of a perfect square will be exact, or that sines and cosines cannot exceed 1 in value.

Whatever the rounding rule, the arithmetic unit should be designed so that the rounding operation is easily implemented for the implicit rounding that can arise with coercion or assignment, as mentioned earlier, as well as with the usual arithmetic operations. Note that, because of the coercion rule, any two numbers involved in an arithmetic operation will always be, at least in effect, in the same precision, the precision prevailing at the time the operation is carried out.

Exception handling

The exception handling facilities are described in detail elsewhere [8]. For the purposes of this paper, the key feature is that handlers are attached to operators (arithmetic operators, functions, procedures – including assignments because a precision change can take place with assignment, and this could result in overflow or underflow). An example in NT is shown in Figure 3.

```
function norm (a : array 1..* of real): real
   var sum := 0.0
   for i:1.. upper(a)
      sum := sum + a(i)**2
   end for
   result sqrt(sum)
end norm
---- % n and the array b are determined in
---- % these statements, where n is
---- % the number of elements in b
handler h
   on failure:
      precision 2*currentprecision + getexp(n) + 1
      x := norm(b)
      nextstatement
end h
var x := norm(b)@h
```

Figure 3. The function computes the Euclidean norm. It is first invoked in the current precision. Any intermediate overflow or underflow that might occur is not handled, and this causes the function to raise the *failure* exception. This in turn invokes the handler called *h* which causes the calculation to be repeated in high enough precision so that no intermediate overflow or underflow can occur. An overflow might still occur on assignment to *x*. For this to be handled, another handler would have to be attached to the assignment operator inside the handler *h*.

The advantage in this example, compared to Figure 2, is that the norm is almost always done in the more efficient lower precision, and the extra work in higher precision is done only when necessary, and presumably only rarely. This illustrates an exception handling technique that can be applied quite generally.

There are only 4 built-in exceptions associated with arithmetic operations – *overflow, underflow, domainerror*, and *uninitialized* – plus one other, namely *failure*, which is raised by a function or procedure in which an unhandled exception occurs. (There are only 2 other built-in exceptions, but they are associated with input.) Users can also explicitly raise *user-defined* exceptions. (For example, a user could decide to declare *toonearsingular* to be an exception, and *raise* it in a procedure for solving linear equations.)

The *overflow* exception can be raised by a floating-point add, or an integer multiply, or the exponential function, and so on. The user knows which operation raises the *overflow* exception because the handler is associated with the operation. (This is one of the advantages of attaching the handler to an operator. Another is that anyone reading the program knows which operators have been modified by a handler, and which have not.)

Based on our experience in implementing these exception handling facilities on both Vax and Sun 3 systems, we have been led to the following conclusions:

(1) The hardware should be designed so that it is easy for the system to identify the location and type of an exception. This means it must be possible to provide precise interrupts and to provide for a software mechanism equivalent to UNIX signals.

(2) The system also needs to be able to restart computations following an exception (that is, return to a machine instruction following the location where the exception occurred, as opposed to restarting the instruction which raised the exception). Besides being able to provide precise interrupts, this means that it must also be possible to preserve at least part of the machine state.

(3) Tracing back from an unhandled exception through what might be a sequence of *failure* exceptions, due to a nesting of subprograms, requires that the system be able to 'unroll' stack frames and return to an arbitrary point after an exception. This is mainly a software problem, but the hardware can make it easier – for example, the register mask saved by the VAX 'calls' instruction simplifies restoring the registers when a stack frame is released.

Concluding remarks

On the basis of our experience with a software system that supports variable-precision floating-point arithmetic and exception handling which have been especially designed to facilitate scientific computing, as well as experience with an earlier hardware unit with some aspects of these features, we have described the specifications we believe are most suitable for an arithmetic coprocessor that could support such a system.

Bibliography

[1] Aberth, O. and Schaefer, M., Precise Computation using Range Arithmetic, via C++, private communication.

[2] Burks, A.W., Goldstine, H.H. and von Neumann, J., Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Part I, Vol.1. Report prepared for U.S. Army Ord. Dept. (1946). Reprinted in John von Neumann Collected Works, Vol.V (MacMillan Coy., New York, 1963), 34-79.

[3] Cohen, M.S., Hamacher, V.C. and Hull, T.E., CADAC: An Arithmetic Unit for Clean Decimal Arithmetic and Controlled Precision, Proceedings 5th Symposium on Computer Arithmetic (IEEE Computer Society, Ann Arbor, Michigan, 1981), 106-112.

[4] Cohen, M.S., Hull, T.E. and Hamacher, V.C., CADAC: A Controlled-Precision Decimal Arithmetic Unit, IEEE Transactions on Computers, vol. C-32, 4 (1983), 370-377.

[5] Hull, T.E. and Abrham, A., Properly Rounded Variable Precision Square Root, ACM TOMS 11, 3 (1985), 229-237.

[6] Hull, T.E. and Abrham, A., Variable Precision Exponential Function, ACM TOMS 12, 2 (1986), 79-91.

[7] Hull, T.E. and Cohen, M.S., Toward an Ideal Computer Arithmetic, Proceedings 8th Symposium on Computer Arithmetic (IEEE Computer Society, Como, Italy, 1987), 131-138.

[8] Hull, T.E., Cohen, M.S., Sawchuk, J.T.M. and Wortman, D.B., Exception Handling in Scientific Computing, ACM TOMS, 14, 3 (1988), 201-217.

[9] Hull, T.E. and Hall, C.B., Precision Control and Exception Handling in Scientific Computing, Proc. Symp. Scientific Software (ed. D.Y. Cai, L.D. Fosdick and H.C. Huang), May 31-June 3, 1989 (China University of Science and Technology Press, Beijing, PRC, 1989), 118-131.

[10] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE Computer Society, 345 East 47th Street, New York, NY 10017, USA, 1985).

[11] IEEE Standard for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Standard 854-1987 (IEEE Computer Society, 345 East 47th Street, New York, NY 10017, USA, 1987).

[12] Kulisch, U.W. and Miranker, W.L., The Arithmetic of the Digital Computer: a New Approach, SIAM Review 28, 1 (1986), 1-40.