Principles, Preferences and Ideals for Computer Arithmetic

Thomas E. Hull

Christian H. Reinsch

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
80-339
ABSTRACT

This paper presents principles and preferences for the implementation of computer arithmetic and ideals for the arithmetic facilities in future programming languages. The implementation principles and preferences are for the current approaches to the design of arithmetic units. The ideals are for the long term development of programming languages, with the hope that arithmetic units will be built to support the requirements of programming languages.

NOTE: This is a draft and has not yet been approved by IFIP WG 2.5 as stated in the text.
PRINCIPLES, PREFERENCES AND IDEALS FOR COMPUTER ARITHMETIC

T.E. Hull, C.H. Reinsch and J.R. Rice

1. INTRODUCTION

This paper presents principles and preferences for the implementation of computer arithmetic and ideals for the arithmetic facilities in future programming languages. This paper has been discussed by IFIP Working Group 2.5 on Numerical Software and has been approved by the Working Group, but it does not constitute an official IFIP document.

The implementation principles and preferences are taken from [Reinsch, 1979] with very minor changes. Thus we restrict ourselves to a most concise statement of them and refer the reader to this source for lengthy motivations and analyses. These implementation guidelines are a minimal set for the current approach to arithmetic hardware. They are based on the experiences of the numerical computation community with the many arithmetic units in widespread use that have very poor numerical properties. It is hoped that the considerable difficulties due to these poor designs can be avoided in future systems by adhering to these implementation guidelines. They do not imply significant additional expenses or complexity and, in fact, arithmetic units such as the IEEE 1980 standard [Coonen et al, 1979] based microprocessors meet all these implementation requirements and have many other desirable properties in addition [Coonan, 1980].

The ideals for the long term are based on the paper [Hull, 1979], but this paper focuses more sharply on the arithmetic issues. We give some of the primary motivations for the ideals stated; the reader may consult this reference for further discussion. There ideals do imply considerable
reorganization of the hardware design plus some extra complexity and cost. A cost analysis [Hamacher, 1980] of specific designs to support these ideals suggests a factor of about two increase in the cost of the arithmetic units and a much smaller increase in execution times. We believe that these ideals may lead to a decrease in overall computing costs.

2. **THE SYSTEM OF FLOATING POINT NUMBERS.** The system of floating-point numbers exactly representable is main memory is denoted by $F$; similar systems of double, triple or higher precision are denoted by $F_2$, $F_3$, ..., if implemented. The range of $F$ is defined by $\lambda$, $\sigma$, $\sigma$ and $\lambda$ which are, respectively, the smallest negative, largest negative, smallest positive and largest positive numbers; see Figure 1.

![Figure 1. The extremal points of $F$ which define its range.](image)

Computations which produce numbers outside the range of $F$ cause over/underflow, O/U. Numbers in $F$ are usually represented as $x = m \times b^{e-t}$ where $-b^t < m < b^t$ and $e_1 \leq e \leq e_2$. The parameters $m$ and $b$, and usually $t$, are integers; it is common to assume $|m| \geq b^{t-1}$ to give normalized numbers.

For $F$ we have:

**Principle 1.1:** $F$ is to be symmetric, i.e., zero is in $F$, $-x$ and therefore $|x|$ is included for each $x$.

**Principle 1.2:** $F$ is to be balanced in the sense that $\sigma \times \lambda = 1$.

**Preference 1.3:** Several levels of precision should be provided with

$F \subseteq F_2 \subseteq F_3 \subseteq \ldots$

**Preference 1.4:** The special floating point operands $\omega$(UNDEFINED) and $\pm$ OVERFLOW should be provided.

**Preference 1.5:** A positional system with small radix $b$ should be used.
Implementations following these principles and preferences provide an environment which is natural for the occasional programmer and which allows the expert to anticipate and handle exceptional cases with relative ease. The use of a small radix minimizes the inherent growth of round-off. [Reinsch, 1979] contains further details on the desirable properties of the radix b.

3. FLOATING POINT ARITHMETIC. The symbol * denotes any of the four symbols +, -, *, /, and * denotes the computed result of *. O/U is excluded in this section.

Principle 2.1: The hardware reference manual must provide a precise and complete algorithmic description of the implemented floating point operations.

The next points refer to the rounding scheme used; the preferred scheme is perfect rounding (rounding to the nearest number in F, towards even numbers in the case of ties). We would accept less desirable schemes that satisfy Assumptions A or B below. Note that perfect rounding automatically meets the requirements of Principles 2.2 - 2.8. Assumptions A and B are developed from the following criteria:

(i) the sizes of the maximum and average rounding error should be small.
(ii) the arithmetic should be suitable for rigorous rounding error analysis.
(iii) most laws of exact arithmetic should remain valid.

Assumption A: There is a number \( \varepsilon_{\text{mach}} \) called machine precision so that for all \( x, y \in F \):

\[
x \ast y = x \cdot (1 + \xi) \ast y \cdot (1 + \eta)
\]

with \( \xi, \eta \) depending on \( x, y, \ast \) but

\[
|\xi|, |\eta| \leq \varepsilon_{\text{mach}} < 1
\]
Assumption B: For all $x, y \in F$:

\[ x \ast y = (x*y) \cdot (1+\alpha) \]

with $\alpha$ depending on $x, y, \ast$ and $|\alpha| \leq \varepsilon_{\text{mach}}$.

Principle 2.2: The rounding error in the floating point operations must be locally unbiased or, at least, must have a negligible bias compared with the rms error.

Principle 2.3: The floating point operations must satisfy Assumption A or, preferably, Assumption B.

Principle 2.4: The hardware reference manual must include the actual value of each $\varepsilon_{\text{mach}}$ for Assumption A or B.

Certain relational properties are not valid for finite precision arithmetic and it is not practical to preserve all the mathematical properties of arithmetic and functions. However, the following properties can and should be preserved.

Principle 2.5: The floating point operations must preserve strong monotonicity; that is for all $w, x, y$ and $z$ in $F$.

\[ x \ast y \leq w \ast z \text{ implies } \hat{x} \ast \hat{y} \leq \hat{w} \ast \hat{z} \]

This simple principle has the following corollaries:

(a) $x \ast y = w \ast z$ implies $\hat{x} \ast \hat{y} = \hat{w} \ast \hat{z}$

(b) the result of a floating point operation does not change if different encodings of the operands are used (e.g., $+0$ and $-0$)

(c) commutativity is preserved:

\[ \hat{x} + \hat{y} = \hat{y} + \hat{x}, \quad \hat{x} \ast \hat{y} = \hat{y} \ast \hat{x} \]

(d) sign-invariance is preserved:

\[ (\hat{-x}) \ast (\hat{-y}) = \hat{x} \ast \hat{y}, \quad (\hat{-x})/(\hat{-y}) = \hat{x}/\hat{y} \]

Principle 2.6: The floating point operations must by symmetric with respect to the signs of operands:

\[ (-x) + (-y) = -(x + y) \]
\[ (-x) \ast y = -(x \ast y) \]
\[ (-x) / y = -(x/y) \]
Principle 2.7: Subtraction must be related to addition as follows:

\[ x - y = x + (-y) \]

Principle 2.8: The floating point operations must produce the exact result if it is representable in \( F \), that is

\[ x \cdot y \text{ in } F \text{ implies } x \cdot y = x \cdot y \]

Note that Principles 2.5 and 2.7 imply that the rounding error is at most one unit in the last place.

The complexity of reliable software would be reduced greatly if the following preference were implemented.

Preference 2.9: A standard scheme for all computers should be adopted for the rounding of floating point operations.

The final preference is of somewhat narrower interest, but its application is frequent enough to merit its implementation.

Preference 2.10: Floating point multiplication should be available producing the exact double length product of two single precision floating point numbers. The result should be a feasible operand for double precision addition and subtraction if they are implemented.

4. OVERFLOW AND UNDERFLOW OF EXPONENT.

The preceding section involves the normal cases of arithmetic; some results, however, cannot be in range and they create exceptional cases. Other exceptional cases (e.g., division by zero and square roots of negative numbers) are better prevented by the software designer than handled by the hardware designer.

Principles 3.1 and 3.2: Each occurrence of overflow or underflow must be reported. More generally, a floating point operation must never deliver an incorrect or undefined result without an error indication.

The alternatives for action subsequent to an exceptional case are discussed in [Reinsch, 1979]. The error indicators must, of course, be made
known to the programmer and he should have the ability to choose the subsequent action.

5. THE ARITHMETIC RELATIONS.

The symbol \( \sim \) denotes any of the relational operators \(<, \leq, =; \neq, \geq\) and the caret \( \wedge \) again denotes the computed value. It is not possible to preserve all these relations between variables throughout a computation, but one must be able to evaluate them correctly at any point in the computation.

**Principle 4.1:** The hardware must always allow correct decisions for the relational operators, i.e.

\[ x \sim y \text{ if and only if } x \wedge y \]

Most hardware implementations require \( x \sim y \) to be replaced by \((x-y) \sim 0\) and hence this principle may place constraints on how well subtraction is implemented. The use of subtraction raises the nuisance possibility of O/U. This principle together with Principle 3.1 makes it safe to use a statement like:

\[ \text{If } (x \neq y) \text{ then } z = 1/(x-y) \]

6. INTEGER ARITHMETIC

**Principle 5.1:** Integer arithmetic must provide for \(+, -, \times, \div\), the modulo function and the relational operators \(<, \leq, =, \neq, \geq, \rangle\).

The following preference is important for computers without floating point hardware or where a multi-precision software package is likely to be needed.

**Preference 5.2:** The hardware should provide efficient facilities for

(a) addition with carry-in

(b) subtraction with borrow

(c) a wrapped around result in case of O/U

(d) access to the double length product

(e) The modulo function and \( \div\) with double length numerator
Principle 5.3: Integer arithmetic must produce either the exact result or an error condition.

Principle 5.4: The exact result must be delivered for all the operations of Principle 5.1 whenever the result is representable.

7. CONVERSION OF TYPE, FORMAT AND RADIX

The previous points concern the hardware design while this final section involves operations currently done by software. However, they are basic to all arithmetic and are likely to be implemented in the hardware of future computers. An example of conversion of type is going from single precision to double precision or vice versa. An example of conversion of format is going from integer to floating point or from character string to integer.

Principle 6.1: Conversions of type or format must be done exactly when possible and must always have an error in the result of less than one unit in the last place.

Input/output usually involves a change of radix as well as a conversion of format and it is not practical to achieve such high accuracy in this case.

Principle 6.2: Conversion of radix must be accurate within one unit in the last place for a specified inner exponent range (e.g. 10^{-6} to 10^{12}); in particular, integer conversion must be exact in this range as well as the conversion of common fractions with exact representations in both radices (e.g. 0.5).

Preference 6.3: Each computer system should have a single input/output conversion routine for all users.

8. PRECISION IN PROGRAMMING LANGUAGES

The principles and preferences of the preceding sections are directed toward the current situation. Arithmetic units which meet these minimum
requirements provide a base for reasonable numerical computation. We now turn to the question: what is desirable for the long run, what would be the ideal situation? Numerical computation is done via a relatively high level language and thus the desirable properties of arithmetic must be expressed at this level and not by specifying detailed properties of arithmetic units. The implication is that hardware designers will eventually produce units to support the programming language level requirements.

The ideals given below are not very practical with the current binary word oriented computer structure. We feel that the ideals can be met with a modest extra cost in hardware and that the benefits derived are likely to worth this cost. We are not asking for completely blue sky capabilities such as exact arithmetic. Note that it would be no surprise if the basic computer architecture would change once or twice in this century; engineers are even now exploring architecture based on character string processing where the ideals expressed below would be implemented naturally.

A programming language should provide a means of expressing algorithms which is as natural as possible and which minimizes the numbers of surprises and irrelevant rules. The numerical language for numerical computations is decimal and scientific notation, i.e., numbers like 1234.2201 or 1.2343301*10³. The underlying arithmetic should behave as we would expect from reading the program. This gives us

**Ideal 7.1:** The radix of the arithmetic is 10.

The fractional and exponent parts of scientific notation are represented by sign and magnitude; the behavior of the actual representation should not conflict with this. One should be allowed to specify the numerical enviornment to a certain extent.

**Ideal 7.2:** The declaration of data type allows separate specification of the range of the exponent and the amount of precision.
A recurring source of difficulty in current programming languages is that the precision of operators is determined from the precision of the operands. PL/1 and ADA allow specific declarations of the precision of variables in an attempt to allow the programmer to explicitly state the desired level of precision in the computation. This is a step in the right direction, but the connection between precision in operators and operands should be cleanly broken.

**Ideal 7.3:** The programmer can specify the precision of the arithmetic operation apart from and independently of the precision of the operands.

This ideal only solves part of the precision problem: many computations need to be made several times in different precisions or, more generally, the amount of precision needed is not known until the part of the computation is made.

**Ideal 7.4:** The amount of the precision is allowed to vary dynamically.

We visualize Ideal 7.4 being implemented by "precision blocks" with simple versions for a complete program unit or a single statement.

Ideal 7.4 means that the precision is to be as specified, not "at least as high" as specified. Perhaps the main motivation for this is belief in the principle that the programmer should know what is happening at all times. If "use 8 digits precision" is specified, then that should be what happens.

Beyond this basic principle is the fact that there are common cases where the "at least" approach simply does not work; if one asks for 8 digits precision and then 12 digits, it will not do to have 14 digits used each time. The arithmetic unit described in [Hamacher, 1980] achieves dynamic precision with a moderate increase in the cost of the unit. In a word oriented computer, it might be feasible to achieve high efficiency for moderate precisions (say 25 digits or less) through hardware and then switch to software for higher precisions. There is no requirement that precision of 500 digits be inexpensive.
9. **ARITHMETIC AND ROUNDDING**

It is obvious that if we prefer perfect rounding for current computers, then we have this as our ideal for the future. A really good arithmetic unit provides much more than the minimums of the foregoing principles and preferences. The question is: how many of these capabilities should be available at the programming language level? Perfect rounding will be the normal mode of operation; should we also allow round-up, round-down, round-toward-zero (chopping), etc. to be specified at the language level? We believe such capabilities should be accessible at the programming language level, but it is not imperative that accessing them be painless.

The reason for the following ideal is the basic principle that the programmer must always be able to know what is happening.

**Ideal 9.1:** There are rules that resolve all ambiguities in arithmetic expressions.

10. **ARITHMETIC ERRORS AND EXCEPTIONS**

An ideal arithmetic unit will identify various types of arithmetic errors and exceptions; some of this information and part of the responsibility for corrective actions must surface at the programming language level. The number system should have values to represent errors and exceptions:

**Ideal 9.1:** The number system has some special values including ±OVERFLOW, ±UNDERFLOW, UNDEFINED and possibly others.

Examples of other values include INFINITY, ZERO-DIVIDE (which is essentially the same as INFINITY) and OUT-OF-RANGE. Note that 1/0 is not the same as overflow; the former is an error in the program (unless it is due to round-off) and the latter is a shortcoming in the hardware. It is not clear that the programming language should provide special values for every program error. All variable values should be initialized to UNDEFINED.
Ideal 9.2: The default action for an error or exceptions is to stop the computations and provide a message about where and why.

Ideal 9.3: The programmer can specify special actions to be taken when exceptions or errors occur.

Ideal 9.4: Program control returns to the point of interruption unless the programmer specifies otherwise. One can assign a value to the "RESULT" as part of the actions.

We visualize exception handling involving an "ON" condition with an associated scope to define an exception handling block. There should also be a short form of the "ON" condition to handle simple statements such as

\[ \text{NEXTSTEP} = \min(Y/\text{ERROREST}, 2.) \]

and one needs protection from a nearly underflowed - or even zero-value for \text{ERROREST}.

One should consider the possibility of giving the programmer access to the wrap-around value on overflow or underflow. This value has the correct fractional part and the exponent is modulo the range of the exponent. For example with REAL (PRECISION = 10, RANGE = -99,99) \( X \), \( Y \) and \( X = 1.2 \times 10^{55} \), \( Y = 2.0 \times 10^{52} \), the wrap-around value of \( X \times Y \) is \( 2.4 \times 10^{-91} \) where \( -91 = 55 + 52 - (99+99) \). A sophisticated programmer can set up a counter on overflows and compute quantities like the combinations of \( N \) things taken \( K \) at a time \((= N! / (N-K)! K!))\) accurately even though the intermediate results go far outside the intended range of the computer. It certainly is not necessary to provide these values, but one can argue that some people can use them and it will not cause any problems for others.

REFERENCES


