To Appear in IEEE Trans. on Comps.

Multi-Step Gradual Rounding

Corinna Lee

Computer Science Division
University of California
Berkeley, CA 94720

Abstract: A value V is to be rounded to an arbitrary precision resulting in the value V". Conventional rounding technique uses one step to accomplish this. Alternatively, *multi-step rounding* uses several steps to round the value V to successively shorter precisions with the final rounding step producing the desired value V". This alternate rounding method is one way to implement, with a minimum of hardware, the denormalization process that the IEEE floating-point standard 754 requires when underflow occurs. There are certain cases for which multi-step rounding produces a different result than single-step rounding. To prevent such a *step error*, I introduce a new rounding procedure called *gradual rounding* that is very similar to conventional rounding with the addition of two tag bits associated with each floating-point register.

Index Terms: denormalized numbers, floating-point arithmetic, IEEE Standard 754, multi-step gradual rounding, tagged data for rounding, variable precision arithmetic

1. Introduction

In numeric computations, a precise value V must be represented in a computing machine by a value V" of finite precision, where precision refers to the number of significant bits in the fraction. As shown in Figure 1a, the function used to map V to V" is known as the rounding rule. There are situations, for reasons of economy or ease of design, in which it is desirable to round in multiple steps. As shown in Figure 1b, however, rounding first to an intermediate value V' does not always produce the same result as if rounding directly to V". I call this discrepancy a *step error*. In order to produce the same result as if rounding from V to V" in one step, I introduce a new rounding function called the *gradual round*. The motivation for rounding in multiple steps and how it can be done to simulate rounding in a single step are the purposes of this paper.

1.1. Motivation for Multi-Step Rounding.

Multi-step rounding is useful when it is undesirable to perform a round in a single step. For example, as specified by the IEEE standard 754 for binary floating-point arithmetic [1], the handling of underflow exceptions requires rounding to an arbitrary precision. Underflows are processed in one of two ways: either a denormalized result with diminished precision is delivered as the default action, or a trap occurs and a user-defined trap-handler is given a value rounded to the destination's precision with an adjusted exponent.

For simplicity, the implementor may choose to have the hardware always trap in response to an underflow exception and provide the trap-handling software with the underflowed result rounded to the destination's precision. For the default action, the trap-handling software could re-execute the operation and force the formation of a denormalized number by rounding the ALU result to a precision that varies with the amount of the underflow. This requires the following hardware support:

(1) a mechanism to save the operands in order to re-execute the operation

(2) the ability to round (in a single step) the ALU result to an arbitrary precision

Alternatively, the trap-handling software can round the hardware-delivered value to the shorter precision effecting a two-step round. The first hardware support requirement is no longer necessary and as will be shown, the hardware cost of the second requirement can be minimized.

Multi-step rounding is also appropriate when it is desirable to perform arithmetic operations in a precision not directly supported by the underlying hardware.[1] Such arithmetic simulations occur when the

[1] A precision is said to be directly supported in hardware if the result of an arithmetic operation can be directly rounded to that precision.

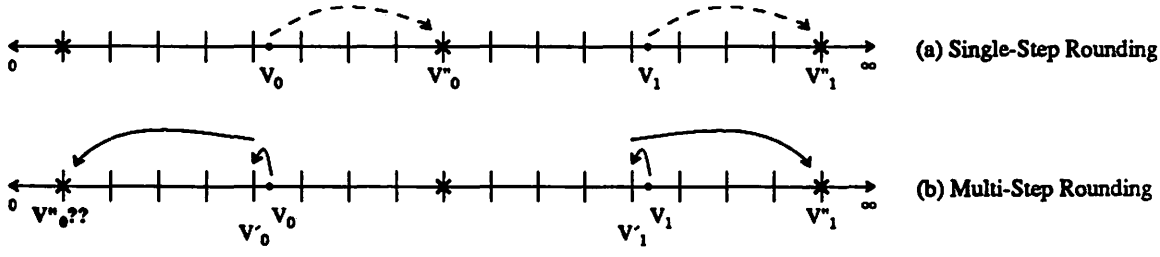

Figure 1 Single-Step Rounding vs. Multi-Step Rounding

For single-step rounding, using the "round-to-nearest-even" rule, the real value V_i is rounded to V''_i, the nearest representable value with precision p" (indicated by the crosses). This is the conventional manner by which rounding is carried out.

For multi-step rounding, using the same rule, the real value V_i is rounded first to V'_i, the nearest representable value with precision p' (indicated by the vertical bars) and then V'_i is rounded to V''_i.

For some V_i, rounding to precision p" in two steps does not produce the same V''_i as rounding in one step (eg. V''_0 above). The reason for this step error, how it can be corrected and the motivation for rounding in >1 steps are discussed in this paper.

user wishes to experiment with non-standard precisions. For example, at the University of Toronto, Hull et al. have developed a Pascal-like language called Numerical Turing that features variable-precision real arithmetic [2]. Arithmetic simulations are also necessary when design simplicity has higher priority than speed of execution: hardware that rounds to one preset precision is easier to design than hardware whose precision can be altered at run-time.

A need arises for a multi-step round with more than two roundings when variable-precision arithmetic is combined with the specifications of the IEEE standard 754 for handling underflow exceptions. For example, an underflow exception for a precision not supported directly in hardware requires a three-step round to produce the correct denormalized value:

1st round	round result from ALU to hardware-supported precision
2nd round	round value to unsupported precision and detect underflow exception
3rd round	round value to precision that varies with the amount of underflow to form denormalized result

Interest in such an application is demonstrated by the article by Cody et al. entitled "A Proposed Radix- and *Word-length-independent* Standard for Floating-point Arithmetic" [3].

1.2. Implementation Assumptions. Providing the gradual rounding capability does not preempt the need for conventional rounding logic. Although there is a necessity for the gradual round, it is not expected to be used nearly as often as the conventional round. In order to minimize the additional hardware, I assume the rounding mode does not change during the execution of a multi-step round. Otherwise, the rounding mode for one step could be round-to-+∞ that causes an increment to occur (with the appropriate sign), and then the rounding mode for the subsequent step could be round-to-0 which performs a truncation. In such a situation, the increment would have to be canceled by a full-length decrement. When the multi-step round is used for the formation of denormalized numbers or the simulation of variable precision arithmetic, the invariance of the rounding mode is not a restrictive assumption.

I also assume that the values to be rounded are always positive. This excludes designs that produce a 2's complement value as an intermediate result. For these designs, it is feasible to combine part of the conversion back to sign-magnitude form with the rounding function. To simplify the discussion, my explanation of the gradual round excludes negative inputs. The results of this paper are easily extended to cope with simultaneous complementation and gradual rounding.

1.3. Notation. Conventional rounding uses six bits of information:

l	The *least* significant bit of the precision to which the value is being rounded.
r	The *round* bit is just to the right of *l*.
s	The *sticky* bit is the logical-or of all the bits (collectively known as *S*) to the right of *r*.
σ	The sign of the result where σ=1 represents a negative sign.
RM	Two bits for the Rounding Mode. Only the rounding modes defined by the IEEE standard 754 will be considered. Of these, round-to-0, round-to-+∞ and round-to-−∞ are categorized as *directed* rounding modes since the direction of rounding is fixed even though the rounded value is not always the closest representable value to the ideal result. On the other hand, round-to-nearest-even is classified as *unbiased* rounding since the cumulative average error due to rounding is expected to be zero.

From the implementation assumptions, σ and RM are constant for all the rounding steps in a particular multi-step round. However, the rounding bits *lrs* change from step to step since their value depends upon the precision being rounded to (see Figure 2). These six inputs determine the rounding *action* (truncate or increment) to be taken whenever a value is inexact (i.e. *rs*≠00).

Although only the rounding modes used by the IEEE standard 754 are discussed, the results of this paper are easily extended to other rounding modes such as round-halfway-up, round-to-nearest-odd, etc.

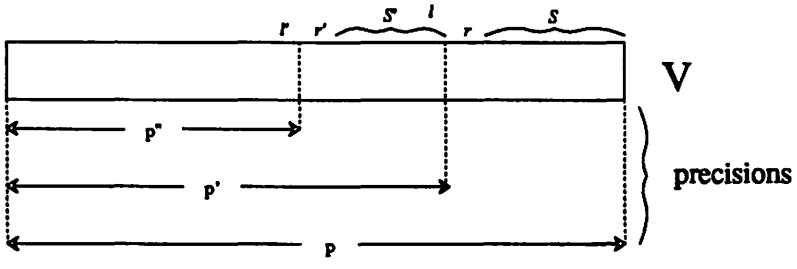

Figure 2 Rounding Bits *lrs* for a Two-Step Round

The rounding bits *lrs* vary for each of the steps in a particular multi-step round. *lrS* are the rounding bits used for the first rounding step while *l'r'S'* are used for the second rounding step. The sticky bit *s* is the logical-or of the bits *S* while *s'* is the logical-or of the bits *S'*.

2. Why Step Error Occurs

As indicated in the introduction, there are values for which multi-step rounding does not produce the same result as single-step rounding when the round-to-nearest-even mode is used. If rounding is viewed as a "1-bit summary of many bits", clearly information is lost after a rounding step. I will show in this section how this lost information will cause a step error for certain cases when using the unbiased rounding mode and never when using the directed rounding modes.

To simplify the discussion, a rounding action function $\alpha(rs)$ is defined as follows:

PROPERTIES OF $\alpha(rs)$:
(1) setting truncate=0 and increment=1, $\alpha(rs)$ is monotonically increasing
(2) $\alpha(00)=0$

POSSIBLE FUNCTION DEFINITIONS FOR α

case	*rs*	a(*rs*)	b(*rs*)	c(*rs*)	d(*rs*)
EXACT	00	0	0	0	0
<1/2	01	0	0	0	1
1/2-way	10	0	0	1	1
>1/2	11	0	1	1	1

The properties of α are the minimum any rounding rule should satisfy. Using the rest of the inputs RM, σ, *l*, the four rounding modes of the IEEE standard 754 can be defined in terms of α as follows:

RM	σ	*l*	α
round-to-0	x	x	a
round-to-+∞	0	x	d
	1	x	a
round-to-−∞	0	x	a
	1	x	d
round-to-nearest-even	x	0	b
	x	1	c

I will show the main result of this section by proving that:

(1) step error can only occur for certain cases when α=b,c and never for α=a,d

(2) no new conditions for step error are introduced when α changes during a particualr multi-step round (eg. when RM=round-to-nearest-even)

The notation of Figure 2 is used extensively in the ensuing discusssion.

DEFINITION

A *step error* at precision p" occurs when the resultant *l'* from a two-step round is not equal to the resultant *l'* from a single-step round. This occurs when either

(1) the single-step round generates a truncation and the multi-step round generates an increment at *l'*

or (2) the single-step round generates an increment and the multi-step round does *not* generate an increment at *l'*

THEOREM

A step error as defined above will occur if and only if either of the following conditions is satisfied:

(1) α=b, $r'S'=100\cdots00$ and a previous step trunctated an inexact value
(2) α=c, $r'S'=011\cdots11$ and a previous step incremented

PROOF

Since, by fixing α=a or d, $\alpha(rs)$ is constant for $r|s=1$ (i.e. when the value is inexact), the rounding action for a single-step round is the same as those performed on each step of a

corresponding multi-step round (except for an exact value). Hence, as shown in Figure 3a, successive rounding steps produce values that lie between the original value V and the desired value V" inclusive. Intermediate values will never "pass" V" since an exact value alsyas generates a "truncation" (eg. V'_1 and V'_2 in Figure 3a). As a result, no step error can occur.

For α=b or c, the rounding action depends upon the actual value of the round bits and therefore varies for a single-step round or any of the steps in the corresponding multi-step round. For the latter, a previous round may cause the current round bits to choose a rounding action that will result in a step error (see Figure 3b). The exact conditions for producing a step error are determined using a two-step round. These conditions are easily extended to a >2-step round.

Suppose step error occurs when α=b (= increment when >1/2). Then either

(1) the single-step round generates a truncation and the two-step round generates an increment at l'

or (2) vice-versa

In the first case since $r'S'rS \leq 1000\cdots00$, a step error can occur only if the first rounding step generates an increment - i.e. $r'S'rS < 1000\cdots00$. Since such an increment cannot possibly propagate a carry into l', the second round step must also generate an increment - i.e. $r'S' > 10\cdots00$. This is impossible and no step error can occur.

For the second case, $r'S'rS > 1000\cdots00$ and the multi-step round cannot generate an increment at l'. This is possible only if the first and second rounding steps truncate - i.e. $r'S' = 10\cdots00$ and $00\cdots00 < rS \leq 10\cdots00$. Note that these values do indeed cause step error to occur.

The argument for α=c (= increment when $\geq$1/2) is similar. •

For round-to-0, α=a always; for round-to-$\pm\infty$, α is determined by the sign of the result σ which is fixed for the duration of a multi-step round. Since step error cannot occur if α=a or α=d, a step error can never occur when using any of the directed rounding modes. For round-to-nearest-even, α may change for a particular multi-step round. However, the argument in the proof for α=b shows that no step error can occur if the first step uses α=c and the second step uses α=b. Similarly for the reverse situation.

3. How to Prevent Step Error

From the discussion in the previous section, it can be seen that the occurrence of a step error should be relatively infrequent when using multi-step conventional rounding. Although infrequent, the potential for misunderstanding the "correctness" of an implementation's arithmetic is high, particularly if exact reproducibility amongst several implementations is desired. Because of the infrequency of occurrence, an architect might be tempted to simply detect and trap on such cases rather than include additional logic that would solve the problem. However, there is no way to correct such cases (even in software) since

(1) each rounding is executed as distinct steps with no indication that the roundings are in any way connected.

(2) important information from previous rounds has been lost

Herein lies one solution. State information is maintained between rounding steps and is used to determine the rounding action for round-to-nearest-even since step error occurs only for this rounding mode. Since from the results of the theorem, step error occurs if the 1/2-way case is indicated *after* the previous round steps have completed, a new rounding procedure, called *gradual rounding*, takes into account the action of the previous round to prevent step error as follows:

increment if (>1/2 case) or
(1/2 even case and previous round truncated an inexact value) or (1)
(1/2 odd case and previous round did not increment)

where "1/2 even case" refers to using α=b when l=0.

To implement this, the necessary state information takes the form of two tag bits, i and x, that are associated with each floating-point register and provide information about the most recent round action performed on the stored floating-point value. Since these bits are stored in the register file, they are part of the program state and must be saved on a context switch. With i=1 to indicate an increment has been generated and x=1 to indicate the value was inexact, these bits, collectively, identify the following three actions:

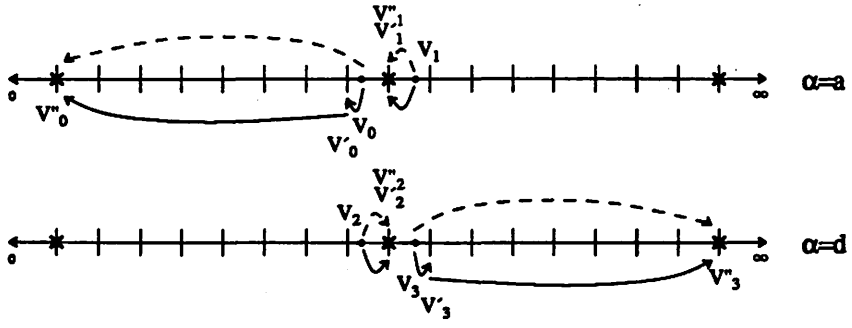

Figure 3a Multi-Step Rounding for α=a, d

For α=a, d, the same action will occur for each step whenever the value is inexact. Hence, successive truncations produce values that are non-increasing in magnitude, while successive increments produce values that are non-decreasing in magnitude. The solid lines depict the multi-step rounds while the dashed lines show the single-step rounds.

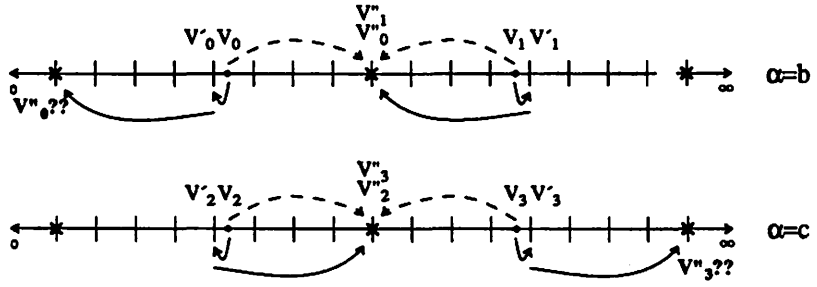

Figure 3b Multi-Step Rounding for α=b, c

For α=b, c, the round action depends upon the actual value of the round bits. Hence, a previous round may cause the current round bits to choose a rounding action that will result in a step error as shown here for $V"_0$ and $V"_3$.

ix	Action of Previous Round
00	nothing (value was exact)
01	truncate
10	— (cannot occur)
11	increment

Thus (1) becomes:

increment if (>1/2 case) or
(1/2 even case and previous round truncated an inexact value) or
(1/2 odd case and previous round did not increment)
$= (rs) / (lr\bar{s}\,\mathrm{ix}) / (lrs\,\mathrm{i})$
$= r \wedge (s / \mathrm{i} \wedge (l / \mathrm{x}))$

Notice the similarity to the implementation of the corresponding conventional round:

increment if (>1/2 case) or (1/2 odd case)
$= (rs) / (lr\bar{s})$
$= r \wedge (s / l)$

Using the following encodings for RM:

RM	Rounding Mode
00	round-to-0
01	round-to-+∞
10	round-to—∞
11	round-to-nearest-even

the complete implementation of the gradual rounding procedure for all four rounding modes is:

I = increment if [([RM=round-to-+∞ and positive sign] or
[RM=round-to—∞ and negative sign]
) and
inexact] or
[RM=round-to-nearest-even and
(>1/2 case or
1/2 even case and previous round truncated an inexact value or
1/2 odd case and previous round did not increment
)]

$= [\,(\bar{R}M\bar{\sigma} \mid R\bar{M}\sigma) \wedge (r \mid s)\,] \mid [\,RM \wedge r \wedge (s \mid \bar{\mathrm{i}} \wedge (l \mid \mathrm{x}))\,]$ (2)

To maintain the state information, two more equations are required to produce new tag bits for storing in the destination register along with the rounded fraction. Since a multi-step round can have more than two steps and the action of *any* previous step can cause a step error, the information in the tag bits must be "passed on" when a particular step indicates an exact value (eg. consider rounding $V=x\cdots x01000000001$ to $x\cdots x1$ in three steps). The following boolean equations will correctly generate the new tag bits, i' and x':

i' = (current increment signal) or (exact value and a previous round action was increment)
$= \mathrm{I} \mid (\bar{r} \wedge \bar{s} \wedge \mathrm{i})$

x' = (inexact value) or (exact value and previous round was not exact)
$= (r \mid s) \mid (\bar{r} \wedge \bar{s} \wedge \mathrm{x})$
$= r \mid s \mid \mathrm{x}$ (3)

3.1. Hardware Implementation of Gradual Rounding. Implementing gradual rounding in hardware is very straightforward — the rounding logic in Figure 4 is merely the realization of the Boolean equations (2) and (3) developed in the previous section. To perform a conventional round, i and x are

cleared before being input, in effect forcing the gradual rounding logic to match that for conventional rounding.

To perform *multi-step* gradual rounding, a variable convert instruction will produce a value rounded to an explicitly given precision. Rounding to any precision shorter than the widest, not merely those precisions at which the rounding hardware is placed, is done by shifting the fraction right before rounding and shifting left after rounding to normalize the fraction. Such barrel shifters can also be used to implement a floating-point add or subtract instruction for the pre-aligning right shift and the normalizing left shift.

3.2. Software Implementation of Gradual Rounding. If it is undesirable to implement the variable convert instruction, subsequent rounding steps can be performed in software; the first rounding step must still be performed by the hardware. As long as the rounding tag bits ix are set properly by the hardware, gradual rounding can be implemented in software.

As shown in Figure 5a, a specially created number yYYY $\cdots$ YYY is added to the bits that will form *rs*. When incrementing is the correct rounding action, the value of yYYY $\cdots$ YYY must cause a carry to propagate to *l*. Conversely, when truncating, yYYY $\cdots$ YYY must prevent any carry from propagating to *l*. After discarding the bits to the right of the *l* bit, the resultant sum is the desired rounded outcome. In addition, the rounding tags must be generated in software. Calculating x' is straightforward, while i' is set by comparing the old *l* with the new *l*.

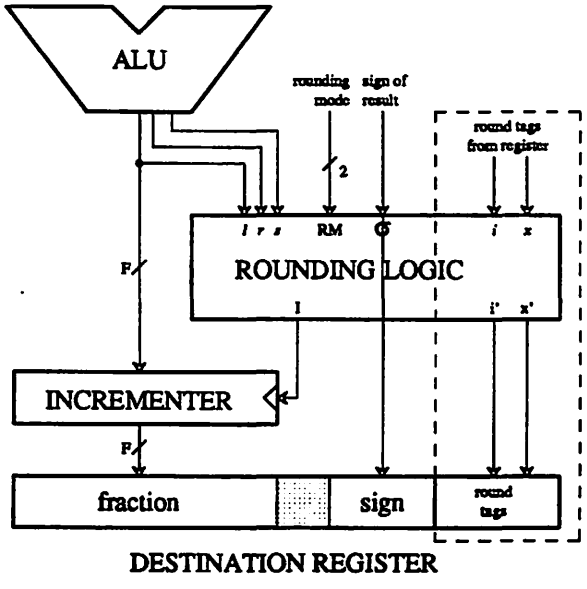

Figure 4 Rounding Logic

This figure shows where the rounding logic is placed in floating-point hardware. The portion boxed in by the dotted line contains the additional logic needed for the gradual round. Signals are one bit unless otherwise noted. In particular, F is the precision of the fraction to be stored in the destination register.

	$XXX\cdots XXXl$	$rsss\cdots sss$
+	$000\cdots 0000$	$yYYY\cdots YYY$
ROUNDED RESULT		discard

Figure 5a Rounding in Software

switch on rounding mode
 case round-to-0:
  $yYYY\cdots YYY = 000\cdots 000$
  $i' = 0$
 case round-to-$+\infty$:
  $yYYY\cdots YYY = \overline{\sigma\sigma\sigma}\cdots\overline{\sigma\sigma\sigma}$
  $i' = 0$
 case round-to-$-\infty$:
  $yYYY\cdots YYY = \sigma\sigma\sigma\cdots\sigma\sigma\sigma$
  $i' = 0$
 case round-to-nearest-even:
  $m = \bar{i}\wedge(l|x)$
  $yYYY\cdots YYY = m\overline{mmm}\cdots\overline{mmm}$
  if (value is inexact)
   then $i' = 0$
   else $i' = i$
end switch
$x' = r \mid s \mid x$
value = value + $000\cdots 000yYYY\cdots YYY$
$i' = i' \mid (\text{old } l \neq \text{new } l)$

Figure 5b Algorithm for Gradual Rounding in Software

For the directed rounding modes, the value of the special number is easily derived from σ and the rounding mode. For unbiased *conventional* rounding, setting $y=l$ and $Y=\bar{l}$ will appropriately generate an increment when lrs=11X or lrs=011. Since l is replaced with the expression $\bar{i}\wedge(l|x)$ in the equation for gradual rounding, setting $y=\bar{i}\wedge(l|x)$ and $Y=\bar{y}$ will generate the correct rounding action for unbiased gradual rounding.

The complete method is given in Figure 5b.

4. Summary

Multi-step gradual rounding emulates rounding a value V to a precision p" by rounding to intermediate values with successively shorter precisions, the last rounding step producing the value V" with the desired precision p". Rounding in multiple steps requires a special type of rounding called gradual rounding in order to reproduce the same result as the single-step round from V to V" and avoid a step error.

The multi-step gradual round is one way to implement the denormalization process specified by the IEEE floating-point standard 754 when underflow occurs. Multi-step rounding is also appropriate when it is desirable to perform arithmetic operations in a precision not directly supported by the underlying hardware. Alternatives to implementing these capabilities require more hardware.

Given the assumptions that the same rounding mode is used for all rounding steps and that the value to be rounded is always positive, a step error at precision p" can only occur when round-to-nearest-even is the rounding mode in effect and a rounding step causes the 1/2-way case to occur for the subsequent step that rounds to precision p". As a result, the gradual round is very similar to the conventional round when using the rounding modes specified by the IEEE floating-point standard 754. The major deviation is the introduction of state information in the form of two tag bits used to preserve the action taken on previous rounding steps. These tag bits are the *minimum* hardware support needed to correct step error.

Either a hardware or software implementation of the multi-step gradual round is possible. The hardware implementation of the gradual round is shown in Figure 4 where the rounding logic is merely the realization of the Boolean equations (2) and (3) derived in Section 3. The conventional round can be executed by setting the rounding tags to zero before being input to the rounding logic. The multi-stepping capability is provided by a variable convert instruction that rounds a value to an explicitly given precision. If it is preferable to minimize the additional hardware cost, subsequent rounding steps after the first one can be done in software by adding a special number determined by the rounding mode.

Acknowledgements

This paper is the result of discussions with W. M. Kahan. Without his encouragement, many of the ideas presented here would have remained undeveloped. I am grateful to Dave Patterson and Bradd Hart for their guidance in the latter stages of writing this paper. Thanks also to my anonymous referees whose comments revealed the inadequacies of my explanations.

This paper is based upon work supported by DARPA under contract order 482427-25840 and by NSERC of Canada.

References

1. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE 754-1985*, IEEE Inc., New York, August 1985.
2. T. E. Hull, A. Abrham, M. S. Cohen, A. F. X. Curley, C. B. Hall, D. A. Penny, and J. T. M. Sawchuk, "Numerical Turing," *SIGNUM Newsletter*, vol. 20, no. 3, pp. 26-33, July 1985.
3. W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson, "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic," *IEEE Micro*, vol. 4, no. 4, pp. 86-100, August 1984.