A Formal Approach to Rounding
Geoff Barrett
Oxford University Programming Research Group

Abstract
This paper presents a formal description of rounding, as specified in the IEEE Standard, and an algorithm to perform the task along with its proof of correctness.

Introduction
The main aim of a standard is that "conforming" implementations should behave in the manner specified — it is, therefore, necessary that they should be proved to do so. It has long been argued that natural language specifications can be ambiguous or misleading and, furthermore, that there is no formal link between specification and program. This paper sets out to formalise the definition of rounding given in [IEEE] and to present an algorithm, with proof of correctness, which performs this task.

The notations used in this paper are Z (see [Abrall, Hayes, Z]) and occam (see [lumos]). The meaning of each new piece of notation is explained in a footnote before an example of its use.

Using a formal specification language bridges the gap between natural language specification and implementation. Natural language specifications have two disadvantages: they can be ambiguous; and it is difficult to show their consistency. The first problem is considered to be an important source of software and hardware errors and is eliminated completely by a formal specification. Further, it is important to show that a specification is consistent (i.e. has an implementation) for obvious reasons.

Of course, it could be argued that an implementation of a solution provides a precise specification of a problem. While this is true, no one likes to read others' code and the structure of a program is designed to be read by machine and not by humans. Moreover, any flexibility in the approach to the problem is hampered by the need to make concrete design decisions. Specification languages are structured in such a way that they can reflect the structure of a problem or a natural language description or even of a program. But, above all, they can also be non-algorithmic. This means that one can formalise what one has to do without detailing how it is to be done.

A formal development divides the task of implementing a specification into four well-defined steps. The first is to write a formal specification using mathematics. In the second, this specification is decomposed into smaller specifications which can be recombined in such a way that it can be shown formally that the decomposition is valid. Third, programs are written to satisfy the decomposed specifications. And, lastly, program transformations can be applied to make the program more efficient or, possibly, to adapt it for implementation on particular hardware configurations.

The example presented here is part of a large body of work which has been undertaken to formally develop a complete floating-point system. A formalisation of most of the IEEE Standard and proofs of the non-exceptional arithmetic routines can be found in [Barrett]. This work has been taken further by David Shepherd to transform the resulting routines into a software model of the lumos T800 processor, and so specify its functions. Thus, the development process has been carried through from formal specification to silicon implementation.

The structure of this paper is as follows. Section 1 presents a formal definition of floating-point numbers and how they are used to approximate real numbers. Section 2 states some theorems and decomposes the definition to give a specification of a rounding module. Section 3 presents the algorithm with its proof and a brief description of the techniques used in the proof.

1 Specification.
This section presents a formal description of floating-point numbers and how they are used to approximate real numbers. The description serves as a specification for a rounding procedure.

First, floating-point numbers and their representation are described. Each number has a format. This consists of the exponent and fraction widths and other useful constants associated with these — the minimum and maximum exponent and the bias:

\[
\begin{align*}
\text{wordlength, expwidth, fracwidth} & : N \\
E_{\text{Min}}, E_{\text{Max}}, \text{Bias} & : N \\
\text{wordlength} &= \text{expwidth} + \text{fracwidth} + 1 \\
E_{\text{Min}} &= 0 \\
E_{\text{Max}} &= 2^{\text{expwidth}} - 1 \\
\text{Bias} &= 2^{\text{expwidth}} - 1 - 1
\end{align*}
\]

The variable names which are used are declared in a signature (the upper part of the box) and any constraints on these are described by the predicates in the lower part.
Four formats are specified—the exponent width and wordlength are constrained to have particular values:

- **Single** - format: \( \text{expwidth} = 8 \) \( \land \) \( \text{wordlength} = 32 \)
- **Double** - format: \( \text{expwidth} = 11 \) \( \land \) \( \text{wordlength} = 64 \)
- **SingleExtended** - format: \( \text{expwidth} \geq 11 \) \( \land \) \( \text{wordlength} \geq 43 \)
- **DoubleExtended** - format: \( \text{expwidth} \geq 15 \) \( \land \) \( \text{wordlength} \geq 79 \)

Once the format is known, the sign, exponent and fraction can be extracted from the integer in which they are stored. This process can be used to identify FIs, FiD, or non-numbers. These will be denoted by NaNF:

\[ \text{NaNF} \equiv \text{Fields} \mid \text{frac} \neq 0 \land \text{exp} = \text{EMax} \]

Now, there are enough definitions to give a definition of the value. This is only specified in single or double formats when the number is not a non-number: ("infinite" numbers are given a value to facilitate the definition of rounding). (See FP, figure 2)

To facilitate further descriptions, FP is partitioned into five classes depending on how its value is calculated from its fields: (non-numbers; infinite, normal, denormal numbers; and zero)

\[ \text{NaN} \equiv \text{FP} \mid \text{frac} \neq 0 \land \text{exp} = \text{EMax} \]
\[ \text{Inf} \equiv \text{FP} \mid \text{frac} = 0 \land \text{exp} = \text{EMax} \]
\[ \text{Norm} \equiv \text{FP} \mid \text{EMin} < \text{exp} < \text{EMax} \]
\[ \text{Denorm} \equiv \text{FP} \mid \text{frac} \neq 0 \land \text{exp} = \text{EMin} \]
\[ \text{Zero} \equiv \text{FP} \mid \text{frac} = 0 \land \text{exp} = \text{EMin} \]

\[ \text{Finite} \equiv \text{Norm} \lor \text{Denorm} \lor \text{Zero} \]

The essential ingredients of rounding are as follows:

- the number to be approximated;
- a set of values in which the approximation must be;
- a rounding mode;
- a set of preferred values in case two approximations are equally good.

Because the number to be approximated may be outside the range of the approximating values, two values, MaxValue and MinValue, are introduced which are analogous to \( +\infty \) and \( -\infty \). The set of Preferred values is restricted to ensure that when two approximations are equally good, at least one of them is preferred. To ensure that rounding to zero is consistent, 0 must be in the approximating values.

\[ \text{Mode} \equiv \text{ToNearest} \lor \text{ToZero} \lor \text{ToNegInf} \lor \text{ToPosInf} \]

(See RoundSignature, figure 3)

The following schemas describe the closest approximations from above and below. If, e.g., the number is smaller than MinValue, then the approximation from below is MinValue. (See Above, figure 4) (See Below, figure 5)

Finally, we are in the position to define rounding in its various different modes. Rounding toward zero gives the approximation with the least modulus:

**RoundToZero**

\[ \text{RoundSignature} \]

| mode = ToZero |
| (r ≥ 0 \land Below) ∨ (r ≤ 0 \land Above) |

Rounding to positive or negative infinity returns the approximation which is respectively greater or less than the given number:

**RoundToPosInf**

\[ \text{RoundSignature} \]

| mode = ToPosInf |
| Above |

**RoundToNegInf**

\[ \text{RoundSignature} \]

| mode = ToNegInf |
| Below |

When rounding to nearest, the closest approximation is returned, but if both are equally good, a member of the set Preferred is returned. (See RoundToNearest, figure 6) These specifications can be disjoined to give the full specification as follows.

\[ \text{Round} \equiv \text{RoundToNearest} \lor \text{RoundToZero} \lor \text{RoundToPosInf} \lor \text{RoundToNegInf} \]

4Decorating the name of a schema with, e.g., \( \text{ToNearest} \) has the effect of dedicating the names of the variables in the signature of that schema throughout.

---

2 This form is equivalent to declaring the variables of Format in the signature and conjointing the constraints with the new constraint.

3 Logical operators between schemas have the effect of merging the signatures and performing the logical operation between the predicates.
### Fields

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign   : 0..1</td>
</tr>
<tr>
<td>exp, frac, nat : (\mathbb{N})</td>
</tr>
</tbody>
</table>

\[
\text{nat} = \text{sign} \times 2^{\text{wordlength}-1} + \text{exp} \times 2^{\text{fracwidth}} + \text{frac} \\
\text{exp} < \text{maxwidth} \\
\text{frac} < 2^{\text{fracwidth}}
\]

**Figure 1: Fields**

### FP

<table>
<thead>
<tr>
<th>Fields; value : (\mathbb{R})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Single (\lor) Double) (\land) (\neg) NaNF (\Rightarrow)</td>
</tr>
<tr>
<td>[\text{exp} = \text{EMin} \land \text{value} = (-1)^{\text{sign}} \times 2^{\text{exp}} \times \text{bias} \times 2 \times \text{frac}_0 ] (</td>
</tr>
</tbody>
</table>
| \[\text{exp} \neq \text{EMin} \land \text{value} = (-1)^{\text{sign}} \times 2^{\text{exp}} \times 2^{\text{bias}} \times (1 + \text{frac}_0)\] where \(\text{frac}_0 = 2^{-\text{fracwidth}} \times \text{frac}\)

**Figure 2: FP**

### Round Signature

<table>
<thead>
<tr>
<th>(r : \mathbb{R};) mode : Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApproxValues, Preferred : (\mathbb{R})</td>
</tr>
<tr>
<td>MinValue, MaxValue : (\mathbb{R})</td>
</tr>
<tr>
<td>value' : (\mathbb{R})</td>
</tr>
</tbody>
</table>

\(\text{Preferred} \cup \{\text{value}'\} \subseteq \text{ApproxValues} \cup \{\text{MinValue}, \text{MaxValue}\}\)

\(0 \in \text{ApproxValues}\)

\(\forall \text{value}_1, \text{value}_2 : \text{ApproxValues} \cup \{\text{MinValue}, \text{MaxValue}\} \mid \text{value}_1 > \text{value}_2\cdot 3p : \text{Preferred} \bullet \text{value}_1 \geq p \geq \text{value}_2\)

\(\forall \text{value} : \text{ApproxValues} \bullet \text{MinValue} \leq \text{value} \leq \text{MaxValue}\)

**Figure 3: Round Signature**

### Above

<table>
<thead>
<tr>
<th>Round Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r &gt; \text{MaxValue} = \text{value}' = \text{MaxValue})</td>
</tr>
<tr>
<td>(r \leq \text{MaxValue} = \text{value}' \geq r)</td>
</tr>
<tr>
<td>(\forall \text{value} : \text{ApproxValues} \cup {\text{MaxValue}} \mid \text{value} \geq r \bullet)</td>
</tr>
</tbody>
</table>
| \(\text{value} \geq \text{value}'\)

**Figure 4: Above**
Below

\[\text{Round Signature}\]

\[r < \text{MinValue} \Rightarrow \text{value}' = \text{MinValue}\]
\[r \geq \text{MinValue} \Rightarrow \text{value}' \leq r\]

\[\forall \text{value} : \text{ApproxValues} \cup \{\text{MinValue}\} \mid \text{value} \leq r \Rightarrow \text{value}' \leq \text{value}'\]

Figure 5: Below

RoundToNearest

\[\text{Round Signature}\]

\[\text{mode} = \text{ToNearest}\]
\[\exists \text{Above}_1; \text{Below}_2 \mid r_1 = r = r_2 \Rightarrow\]
\[\text{value}_1 - r < r - \text{value}_2 \land \text{Above} \lor \]
\[\text{value}_1 - r > r - \text{value}_2 \land \text{Below} \lor \]
\[\text{value}_1 - r = r - \text{value}_2 \land\]
\[\{\text{value}_1 = \text{value}_2 \land \text{Above} \land \text{Below}\} \lor\]
\[\text{value}_1 \neq \text{value}_2 \land \text{value}' \in \text{Preferred} \land (\text{Above} \lor \text{Below})\]

Figure 6: RoundToNearest

FP_Round1

\[\text{Round} \land FP'\]

\[\text{ApproxValues} = \{\text{Finite} \mid \text{Format} = \text{Format}' \land \text{value}\}\]
\[\text{Preferred} = \{\text{Finite} \mid \text{Format} = \text{Format}' \land \text{frac MOD} = 0 \land \text{value}\}\]
\[\text{MinValue} = \{\text{Inf} \mid \text{Format} = \text{Format}' \land \text{sign} = 1 \land \text{value}\}\]
\[\text{MaxValue} = \{\text{Inf} \mid \text{Format} = \text{Format}' \land \text{sign} = 0 \land \text{value}\}\]

Figure 7: FP_Round1

250
So far, the specification is suitable for describing rounding into any format - be it integer or floating-point. To adapt Round specifically for floating-point format, all that is necessary is to fill in the definitions of ApproxValues, Preferred, MinValue, and MaxValue. This inevitably involves the format of the destination, so FP" must be conjoined with Round. Once the definitions are filled in, they are no longer needed outside the specification and can be hidden by existential quantification. It is not difficult to show that the definition of Preferred is consistent with the constraint in Round_Signature, but this will be left until section 2 where a result is proved which makes it even simpler. It is also simple to verify that 0 is an element of ApproxValues and that MinValue and MaxValue satisfy the constraint of Round_Signature. (See FP.Round1, figure 7)

$$\text{FP.Round2} \equiv \text{FP.Round1}\{\text{ApproxValues, Preferred, MinValue, MaxValue}\}$$

The resulting error-conditions have not yet been specified. The conditions resulting in overflow and underflow exceptions are specifically related to a floating-point format and can be described as follows:

$$\text{Errors} \quad ::= \quad \text{inexact} \mid \text{overflow} \mid \text{underflow}$$

$$\text{Error_Signature} \quad ::= \quad r : \mathbb{R}; errors' : \mathbb{P}\text{Errors}; FP'$$

(See Error.Spec, figure 8) (The two alternative conditions under which underflow is included in the set errors' mean that there is a choice about which condition to implement.)

Finally, the whole specification is:

$$\text{FP.Round} \equiv \text{FP.Round2} \land \text{Error.Spec}$$

2 Towards an Algorithm.

The aim of this section is to specify the relation between R and its representations in the program.

First, notice the simple result that the order on the absolute value of a number is the same as the usual order on the less significant bits of its representation as a word:

$$\text{FP; FP}' \mid \text{Format} = \text{Format}' \land \neg(NaN \lor NaN') \vdash \text{abs value} \leq \text{abs value}'$$

$$\Leftrightarrow \text{nats MOD 2^{\text{wordlength}-1} \leq \text{nats}' \text{MOD 2^{\text{wordlength}-1}}$$

This can be used to see that the number of least modulus with modulus greater than a given finite number is obtained by incrementing its representation as a word: (See Succ, figure 9)

$$\text{FP; FP}' \mid \text{Finite } \land nats = nats + 1 \vdash \text{Succ}$$

From this result, the consistency of Preferred in section 1 can be deduced.

In turn, this means that if the approximation of less modulus is known only enough extra information to determine the five predicates in RoundToNearest is needed to return the correct value. This is, of course, the familiar guard and sticky bits defined below:

$$\begin{align*}
\text{Bounds} \vdash \\
\text{Succ; guard, sticky : 0..1; r : } \mathbb{R} \\
\text{r > 0} & \Rightarrow \text{sign = 0 } \land \text{Below}[value/value'] \\
\text{r = 0} & \Rightarrow \text{Zero} \\
\text{r < 0} & \Rightarrow \text{sign = 1 } \land \text{Above}[value/value'] \\
\text{guard = 0} & \Rightarrow \text{r - value } < \text{value}_0 - \text{r} \\
\text{sticky = 0} & \Rightarrow \text{r - value } < \text{value}_0 - \text{r} \lor \text{r } = \text{value}
\end{align*}$$

$$\begin{align*}
\text{Bounds} \vdash \\
\exists \text{Above}; \text{Below} \mid r_1 = r = r_2 \Rightarrow \\
\text{value}_1 - r < r - \text{value}_2 \Rightarrow \text{guard = 0 } \land \text{sticky = 1} \\
\text{value}_1 - r < r - \text{value}_2 \Rightarrow \text{guard = 1 } \land \text{sticky = 1} \\
\text{value}_1 - r < r - \text{value}_2 \Rightarrow \text{sticky = 0} \\
\text{value}_1 = \text{value} \Rightarrow \text{guard = 0 } \land \text{sticky = 0}
\end{align*}$$

This is, however, not quite enough information to return the correct overall condition. If r ≥ 2^{EMax' - Bias'}, this information is lost. Conversely, it is not possible to determine the overall condition before rounding as the condition Int' cannot be tested until the final result is calculated. Thus, it is necessary to divide Error.Spec into two parts. The inexact and underflow conditions can be determined before or after rounding. The design decision is made that so many error conditions as possible will be determined after rounding in order that the precondition of the module is simpler. Thus, the following decomposition is valid (the validity is demonstrated by the theorem):

$$\begin{align*}
\text{Error.Before} & \quad \text{Error.Signature} \\
\text{overflow } \in \text{errors}' \Rightarrow \text{abs r } \geq 2^{EMax' - Bias'}
\end{align*}$$

$$\begin{align*}
\text{Error.After} & \quad \text{errors : P Errors} \\
\text{overflow } \in \text{errors} \Rightarrow \text{abs r } \geq 2^{EMax' - Bias'} \\
\text{inexact } \in \text{errors}' \Rightarrow r \neq value' \\
\text{overflow } \in \text{errors}' \Rightarrow \text{overflow } \in \text{errors } \lor \text{Int'} \\
\text{underflow } \in \text{errors}' \Rightarrow \text{Denorm'}
\end{align*}$$

If a schema is thought of as a function from its unprimed to its primed components, the sequential composition (\(\circ\)) is analogous to the right composition of the two functions. The symbol \(\equiv\) is used to indicate that a design decision has been made.
Because real numbers cannot be represented in a machine, the specification which has been produced so far is not implementable. But, given the conditions of `Bounds`, it is possible to write an implementation. The specification of the module that will be presented here is:

\[
\text{Round}_\text{Proc} \equiv \exists r : R; FP_0 \land \FP\text{Round}_2 \land \text{Bounds} \land \text{Error}_\text{After} \land \text{sign} = \text{sign}'
\]

(Since the sign of 0.0 often depends on factors not available to the rounding procedure, the sign of the number is maintained.)

3 The Algorithm.

There are two things to notice about the specification:

- the specification of errors is conjoined in such a way that the unprimed variable, `errors`, upon which it depends is not restricted by the the other conjuncts; thus the specification decomposes into a sequential composition of a specification on `FP` and a specification on `errors`.

- `Round`, and hence `FP\text{Round}_2`, is a disjunction of specifications and thus may be implemented by a conditional.

The first observation can be formalised as:

\[
\vdash \text{Round}_\text{Proc} = (\exists r : R; FP_0 \land FP\text{Round}_2 \land \text{Bounds}) \land \text{Error}_\text{After}
\]

And the second observation can be formalised as:

\[
\vdash \FP\text{Round}_2 \land \text{Bounds} = \\
\FP\text{Round}_2 \land \text{mode} = \text{ToNearest} \land \text{Bounds} \lor \\
\FP\text{Round}_2 \land \text{mode} = \text{ToPosInf} \land \text{Bounds} \lor \\
\FP\text{Round}_2 \land \text{mode} = \text{ToNegInf} \land \text{Bounds} \lor \\
\FP\text{Round}_2 \land \text{mode} = \text{ToZero} \land \text{Bounds}
\]

The first observation has the obvious implication that the module can be implemented as the sequence of two smaller programs, the first of which sets the correct approximation and the second of which returns the correct error conditions.

The second observation leads to a decomposition because each of the disjuncts is disjoint (i.e. the conjunction of any two is not satisfiable). Thus, a conditional can be formed in which the guards discriminate according to the rounding mode.

The following annotated program constitutes a proof of correctness. The predicates in braces, e.g. \{\phi\}P\{\psi\}, mean that if P is executed in a state satisfying \(\phi\), then it is guaranteed to terminate in a state satisfying \(\psi\). Some of the conjuncts of the assertions are omitted for the sake of clarity. The first assertion is called the precondition of the program — if this does not hold on entry to the program, neither is it guaranteed to terminate nor, if it does, to terminate in any sensible state. The rules relating the program to the assertions are described in [Gries], [Dijkstra] and
[Hoare]. A brief description follows:

Rule 1 The program SKIP does nothing but terminate:

\[ \vdash \{\phi\} \text{SKIP}\{\phi\} \]

Rule 2 If the expression \( a \) can be evaluated correctly (i.e. there is no division by zero etc.), then if the state is required to satisfy \( \phi \) after termination, it must satisfy \( \phi \) with \( a \) substituted for \( x \) before:

\[ \vdash \{D \wedge \phi[a/x]\} x := a(\phi) \]

Rule 3 If \( P \) starts in state \( \phi \) and terminates in state \( \psi \) and \( Q \) starts in state \( \psi \) and terminates in state \( \chi \), then \( P \) followed by \( Q \) starts in state \( \phi \) and terminates in state \( \chi \):

\[ \{\phi\} P\{\psi\} \wedge \{\psi\} Q\{\chi\} \vdash \{\phi\} P\{\chi\} \]

Rule 4 The rule for conditionals is that if \( P \) starts in a state satisfying \( \phi \) and its guard and terminates in state \( \psi \) and similarly for \( Q \) then the conditional composition can start in a state which satisfies one or other of the guards and \( \phi \) and terminate in a state satisfying \( \psi \):

\[ \{b \wedge \phi\} P\{\psi\} \wedge \{b \wedge \phi\} Q\{\psi\} \]

IF

\[ \vdash \{b_{p} \wedge \phi\} P\{\psi\} \]

\[ \vdash \{b_{q} \wedge \phi\} Q\{\psi\} \]

Rule 5 The precondition of a program may be strengthened:

\[ (x \rightarrow \phi) \wedge \{\phi\} P\{\psi\} \vdash \{\phi\} P\{\psi\} \]

Rule 6 The postcondition of a program may be weakened:

\[ (\chi \leftarrow \psi) \wedge \{\phi\} P\{\psi\} \vdash \{\phi\} P\{\chi\} \]

The most obscure line is the following: \( \text{nat} := \text{nat} + \text{(guard \wedge (sticky \lor \text{nat}))} \). This is derived from: \( \text{nat} := \text{nat} + ((\text{guard} \wedge \text{sticky}) \lor (\text{guard} \wedge (\text{nat} \wedge 1))) \). Using \( \text{guard} = \text{guard} \wedge 1 \) and the commutativity and associativity of \( \wedge \), the last part of the expression reduces to \( \text{guard} \wedge \text{nat} \). Now, \( \wedge \) distributes through \( \lor \) to give the optimised expression.

The original expression can be seen to be correct by studying the inequalities used to define \( \text{RoundToNearest} \).

\( \{\text{overflow \in errors} \implies r \geq 2^{\text{EMax-Bias}}\} \)

\( \{r \geq 0 \implies \text{Date} \in [FP/FP']\} \)

\( \{r \leq 0 \implies \text{Absont} \in [FP/FP']\} \)

SEQ

IF

\[ \text{mode} = \text{Inzero} \]

\[ \text{SKIP} \]

\[ \{FP\_Round2[FP/FP']\} \]

\[ \text{mode} = \text{T.RegInf} \]

IF

\[ \text{sign} = 0 \]

\[ \text{SKIP} \]

\[ \text{sign} \neq 0 \]

\[ \text{nat} := \text{nat} + 1 \]

\[ \{FP\_Round2[FP/FP']\} \]

\[ \text{mode} = \text{T.PosInf} \]

IF

\[ \text{sign} = 0 \]

\[ \text{nat} := \text{nat} + 1 \]

\[ \text{sign} \neq 0 \]

\[ \text{SKIP} \]

\[ \{FP\_Round2[FP/FP']\} \]

\[ \text{mode} = \text{T.Nearest} \]

\[ \text{nat} := \text{nat} + \]

\[ \text{guard} \wedge (\text{sticky} \lor \text{nat}) \]

\[ \{FP\_Round2[FP/FP']\} \]

\[ \{\text{overflow \in errors} \implies r \geq 2^{\text{EMax-Bias}}\} \]

\[ \text{errors} := \text{errors} \cap \{\text{overflow}\} \]

\[ \{\text{underflow}, \text{inexact} \in \text{errors}\} \]

\[ \{\text{overflow \in errors} \implies r \geq 2^{\text{EMax-Bias}}\} \]

IF

\[ \text{Inf} \]

\[ \text{errors} := \text{errors} \cup \{\text{overflow}\} \]

\[ \neg \text{Inf} \]

\[ \text{SKIP} \]

\[ \{\text{overflow \in errors} \implies \text{Inf} \lor r \geq 2^{\text{EMax-Bias}}\} \]

\[ \{\text{underflow \in errors}\} \]

IF

\[ \text{Denorm} \]

\[ \text{errors} := \text{errors} \cup \{\text{underflow}\} \]

\[ \neg \text{Denorm} \]

\[ \text{SKIP} \]

\[ \{\text{underflow \in errors} \implies \text{Denorm}\} \]

\[ \{\text{inexact} \in \text{errors}\} \]

IF

\[ (\text{sticky} \lor \text{guard}) \neq 0 \]

\[ \text{errors} := \text{errors} \cup \{\text{inexact}\} \]

\[ (\text{sticky} \lor \text{guard}) = 0 \]

\[ \text{SKIP} \]

\[ (\text{inexact} \in \text{errors} \implies r \neq \text{value}) \]

\[ \{FP\_Round[FP/FP']\} \]

Conclusions

It is often heard that formal methods can only be applied to practically insignificant problems, that development costs in large products are too high, and that the desired reliability is still not achieved. Al-
though the problem presented here may seem insignificant, it is only a small part of a large body of work which has been undertaken to implement a provably correct floating-point system. This work develops the system from a Z specification to silicon implementation — an achievement which cannot be considered insignificant. The formal development was started some time after the commencement of an informal development and has since overtaken the informal approach. The reason for this was mainly because of the large amount of testing involved in the intermediate stages of an informal development — a process which becomes less necessary with a formal development.

As for reliability, that remains to be seen. However, the existence of a proof of correctness means that mistakes are less likely and can be corrected with less danger of introducing further mistakes. Errors can arise in two ways: first, a simple mistype in the program; or a genuine error in the proof. Because of the steps in the development, the effect of this can be limited. Either, a fragment of program is wrong and can be corrected without affecting any larger scale properties of the program; or, the initial decomposition was at fault, in which case most of the development may have to be reworked. If the last scenario seems a little dire, remember that decomposition is a prerequisite of any structured programming methodology but errors at this stage are more likely to be discovered in a formal development. Furthermore, there are now two ways to discover bugs and a way to show that they are not present. The possibility of automatic proof-checkers gives some hope that programmers will be able to guarantee the quality of a program more reliably than an architect can guarantee the robustness of a house.

This example, however, does demonstrate some of the advantages which can be gained from a formal specification. Specifications often become modified — either the customer changes his mind or the original description of the problem is found to be at fault. Trying to modify a badly documented system is disastrous. Trying to modify a well documented system is, at best, error prone. Using a formal specification, it is possible to determine which parts of the system to change and, moreover, how to change them without affecting unmodified parts. For instance, if the specification of error conditions were to change, it would be possible to prove that only the second part of the rounding module need, perhaps, its precondition need be changed. The modification can take place without having to resort to various sets of code. Likewise, in the development stage, the specification allows to reason about how proposed modules will fit together. Moreover, modules may be reused with greater confidence because there is a precise description of what each one does.

The advantages of a non-algorithmic formalism speak for themselves. The language used here bears a formal relation to its implementation and can be transformed to emulate the structure of a program. On the other hand, the high-level specification can be written to bear a close relationship to a natural language description — there are many mathematical idioms which already exist to formalise seemingly intractable descriptions. This paper has assumed some familiarity with the IEEE Standard, but it is desirable to use the formalism as a supplement to a natural language specification to which reference can be made in case of ambiguity.

References


