Formal Methods Applied to a Floating-Point Number System

GEOFF BARRETT

Abstract—This paper presents a formalization of the IEEE standard for binary floating-point arithmetic in the set-theoretic specification language Z. The formal specification is refined into four sequential components, which unpack the operands, perform the arithmetic, and pack and round the result. This refinement follows proven rules and so demonstrates a mathematically rigorous method of program development. In the course of the proofs, useful internal representations of floating-point numbers are specified. The procedures which are presented here form the basis for the floating-point unit of the Inmos IMS T800 transputer.

Index Terms—Floating-point arithmetic, formal specification, program derivation, refinement, transputer, Z.

I. INTRODUCTION

THE main aim of a standard is that “conforming” implementations should behave in the manner specified; it is, therefore, desirable that they should be proved to do so. A floating-point package has several features which make it particularly suitable for a formal approach. First, its basic necessity to many programs makes it correctness essential. Second, exhaustive testing is so lengthy as to be impractical. Last, most existing implementations which used an informal methodology are incorrect. The wish to produce a correct floating-point unit for the Inmos transputer and minimize design costs prompted the initiation of the formal verification of the algorithms which were to be used alongside and some time after the informal development. Some testing had already been performed on existing algorithms, and mistakes had been found. However, it is never certain that all bugs have been found by this method. The attempt to make a formal proof of equivalence between the two sets of algorithms quickly revealed the mistakes already discovered, plus one further error.

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. The specification presented in this paper illustrates the usefulness of the structuring and abstraction mechanisms available in a nonalgorithmic specification language. In the implementation, it is not possible to separate concerns between the value returned and the exceptions raised, whereas the structuring devices of the specification language allow this to be done. Furthermore, rounding can be described with sufficient abstraction for both floating-point and integer results: the actual algorithms employed for the two tasks may be quite different and quite different again from an algorithm which is “obviously” correct with respect to a natural language description. Both these features, then, permit a specification which corresponds more closely to the natural language description of [11].

Formality has been applied to floating-point systems before, e.g., [2]. However, this analysis had the aim of eliciting more general axioms in order to describe what the numerical analyst needs in order to ensure certain program properties. The analysis of [10] employs a similar model in a proposal to bound the rounding errors in arithmetic for standard Pascal. Although the object of this paper and both [10] and [2] is to enable formal reasoning about algorithms which use floating-point arithmetic, the specification given here is concrete and is more useful as a definition from which to prove more abstract theorems.

There do exist packages to “validate,” or test, implementations of floating-point (for examples, see [3],[7]). These rely on two things: first, that errors in the implementation will be shown up by test input of particular forms, and second, that the reference implementation which they must provide is correct. Obviously, this can never be a complete testing method.

The notation used in this paper is Z (see [4],[9]). The meaning of each new piece of Z 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 other people’s code, and the structure of a program is designed to be read by machines 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, a natural language description, or even of a program. But, above all, they can also be nonalgorithmic. This means that one can formalize 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 derived from the decomposed specifications. And last, 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 develop formally a complete floating-point system. This work has been taken further by Shepherd to transform the resulting routines into a software model of the inmos IMS T800 processor and so specify its functions. Some of this work can be seen in [8]. Thus, the development process has been carried through from formal specification to silicon implementation.

II. Specification

A. Format and Value

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>( \triangle ) Format</th>
<th>expwidth = 11 &amp; wordlength = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleExtended</td>
<td>expwidth ( \geq ) 11 &amp; wordlength ( \geq ) 43</td>
<td></td>
</tr>
<tr>
<td>DoubleExtended</td>
<td>expwidth ( \geq ) 15 &amp; wordlength ( \geq ) 79</td>
<td></td>
</tr>
</tbody>
</table>

Four formats are specified—the exponent width and wordlength are constrained to have particular values:

<table>
<thead>
<tr>
<th>Single</th>
<th>expwidth = 8 &amp; wordlength = 32</th>
</tr>
</thead>
</table>

For instance, in Single format, the values of the constants are:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>0..1</td>
</tr>
<tr>
<td>exp, frac, nat</td>
<td>0</td>
</tr>
</tbody>
</table>

The hexadecimal number 3FB504F316 has

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>0</td>
</tr>
<tr>
<td>exp</td>
<td>7F16 = 127</td>
</tr>
<tr>
<td>frac</td>
<td>3504F316 = 3474675</td>
</tr>
</tbody>
</table>

Once the format is known, the sign, exponent, and fraction can be encoded as a single integer \( \text{nat} \):  

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

\[
\text{exp} < 2^{\text{expwidth}} 
\]

\[
\text{frac} < 2^{\text{fracwidth}} 
\]

Elements of Fields whose exp is EMax and frac is non-zero are considered to be exception codes, or nonnumbers. These will be denoted by NaNF:

\[
\text{NaNF} \triangleq \text{Fields} | \text{frac} \neq 0 \& \text{exp} = \text{EMax} 
\]

For example, in Single format, the number whose word representation is 7F800001 16 is a NaNF.

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 nonnumber: ("infinite" numbers are given a value to facilitate the definition of rounding):

\[
\begin{align*}
\text{FP} : & \\
\text{Fields, value} : & \\
\text{(Single} \lor \text{Double}) \land \neg \text{NaNF} \Rightarrow \\
\text{exp} = \text{EMin} \land \text{value} = (-1)^{\text{sign}} \times 2^{\text{exp} - \text{Bias}} \times 2 \times \text{frac}_0 \\
\lor \\
\text{exp} \neq \text{EMin} \land \text{value} = (-1)^{\text{sign}} \times 2^{\text{exp} - \text{Bias}} \times (1 + \text{frac}_0) \\
\text{where} \text{frac}_0 = 2^{-\text{fracwidth}} \times \text{frac} 
\end{align*}
\]
To facilitate further descriptions, \( FP \) is partitioned into five classes, depending on how its value is calculated from its fields (nonnumbers; infinite, normal, and denormal numbers; and zero):

- NaN \( \triangleq FP \) | frac \( \neq 0 \) \& exp = EMax
- Inf \( \triangleq FP \) | frac = 0 \& exp = EMax
- Norm \( \triangleq FP \) | EMin < exp < EMax
- Denorm \( \triangleq FP \) | frac \( \neq 0 \) \& exp = EMin
- Zero \( \triangleq FP \) | frac = 0 \& exp = EMin

\( \text{Finite} \triangleq \text{Norm} \vee \text{Denorm} \vee \text{Zero}. \)³

The following constants can be shown to have the correct values:

Single \( \vdash \) Inf = \( (\text{nat} = 7F800000o_{16} \vee \text{nat} = FF800000o_{16}) \)

FP value = 1 \( \Rightarrow \) nat = 3F800000o_{16}

FP value = -1 \( \Rightarrow \) nat = BF800000o_{16}

FP value = 0 \( \Rightarrow \) (nat = 00000000o_{16} \vee \text{nat} = 80000000o_{16}).

### B. Rounding

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

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, and
- 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, \( \text{MaxValue} \) and \( \text{MinValue} \), are introduced. These values delimit the range of the approximating values and are treated by the arithmetic like \( +\infty \) and \( -\infty \). The set of \( \text{Preferred} \) 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 well defined, 0 must be in the approximating values.⁴

#### Modes

\[
\text{Modes} ::= \text{ToNearest} \mid \text{ToZero} \mid \text{ToNegInf} \mid \text{ToPosInf}
\]

#### Round Signature

\[
\begin{align*}
\text{Round Signature} & \\
& \quad \text{r:} \mathbb{R}; \text{mode: Modes} \\
& \quad \text{ApproxValues, Preferred:} \subseteq \mathbb{R} \\
& \quad \text{MinValue, MaxValue:} \subseteq \mathbb{R} \\
& \quad \text{value':} \subseteq \mathbb{R} \\
& \quad \text{Preferred} \subseteq \text{ApproxValues} \cup \{\text{MinValue, MaxValue}\} \\
& \quad 0 \in \text{ApproxValues} \\
& \quad \forall \text{value}, \text{value}': \text{ApproxValues} \cup \{\text{MinValue, MaxValue}\} \mid \text{value} > \text{value}' \cdot \\
& \quad 3p: \text{Preferred} \cdot \text{value} \geq p \geq \text{value} \\
& \quad \forall \text{value}: \text{ApproxValues} \cup \{\text{MinValue}\} \mid \text{value} \leq \max \text{value}' \cdot \\
\end{align*}
\]

The following schemas describe the closest approximations from above and below. If, e.g., the number is greater than \( \text{MaxValue} \), then the approximation from below is \( \text{MaxValue} \):

#### Above

\[
\begin{align*}
\text{Above} & \\
\text{Round Signature} & \\
& \quad \text{r > MaxValue} \Rightarrow \text{value}' = \text{MaxValue} \\
& \quad \text{r} \leq \text{MaxValue} \Rightarrow \text{value}' \geq \text{r} \\
& \quad \forall \text{value}: \text{ApproxValues} \cup \{\text{MaxValue}\} \mid \text{value} \geq \text{r} \cdot \\
& \quad \text{value} \geq \text{value}' \\
\end{align*}
\]

#### Below

\[
\begin{align*}
\text{Below} & \\
\text{Round Signature} & \\
& \quad \text{r < MinValue} \Rightarrow \text{value}' = \text{MinValue} \\
& \quad \text{r} \geq \text{MinValue} \Rightarrow \text{value}' \leq \text{r} \\
& \quad \forall \text{value}: \text{ApproxValues} \cup \{\text{MinValue}\} \mid \text{value} \leq \text{r} \cdot \\
& \quad \text{value} \leq \text{value}' \\
\end{align*}
\]

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

⁴The expression \( X \) denotes the power set of \( X \). The symbol \( \cdot \) is simply a separator.
Notice that if \( MinValue \notin \text{ApproxValues} \), then the value returned by Above cannot be \( MinValue \):

\[
\text{Above} \mid MinValue \notin \text{ApproxValues} \Rightarrow \text{value}' \neq \text{MinValue}.
\]

The two numbers \( 3F8504F3_{16} \) and \( 3F8504F4_{16} \) with respective values \( 1.414213579150390625 \) and \( 1.4142135816986083984375 \) are the closest approximations of the square root of 2 in Single format.

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

\[
\begin{align*}
\text{RoundToZero} & \quad \text{Round} \_\text{Signature} \\
\text{mode} & = \text{ToZero} \\
(r \geq 0 \land \text{Below}) & \lor \\
(r \leq 0 \land \text{Above}) & \\
\end{align*}
\]

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

\[
\begin{align*}
\text{RoundToPosInf} & \quad \text{Round} \_\text{Signature} \\
\text{mode} & = \text{ToPosInf} \\
\text{Above} & \\
\end{align*}
\]

\[
\begin{align*}
\text{RoundToNegInf} & \quad \text{Round} \_\text{Signature} \\
\text{mode} & = \text{ToNegInf} \\
\text{Below} & \\
\end{align*}
\]

When rounding to the nearest, the closest approximation is returned, but if both are equally good and unequal, a member of the set \( \text{Preferred} \) is returned (in the latter case, the restriction which \( \text{Round-Signature} \) imposes on \( \text{Preferred} \) ensures that one of the approximations is preferred):

\[
\begin{align*}
\text{RoundToNearest} & \quad \text{Round} \_\text{Signature} \\
\text{mode} & = \text{ToNearest} \\
\exists a, b : & (\text{Above}[a/\text{value}'] \land \text{Below}[b/\text{value}'] \land \\
& a - r < r - b \land \text{Above} \lor \\
& a - r > r - b \land \text{Below} \lor \\
& a - r = r - b \land \\
& (a = b \land \text{Above} \land \text{Below} \land \\
& a \neq b \land \text{value}' \in \text{Preferred} \land (\text{Above} \lor \text{Below})) \\
\end{align*}
\]

These specifications can be disjoined to give the full specification as follows:

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

So far, the specification is suitable for describing rounding into any format, be it integer or floating-point. To adapt \( \text{Round} \) specifically for a floating-point format, all that is necessary is to fill in the definitions of \( \text{ApproxValues, Preferred, MinValue, and MaxValue} \). This inevitably involves the format of the destination, so \( FP^5 \) must be conjoined with \( \text{Round} \). 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 \( \text{Preferred} \) is consistent with the constraint in \( \text{Round-Signature} \), but this will be left until Section III-B, where a result is proved which makes it even simpler. It is also simple to verify that \( 0 \) is an element of \( \text{ApproxValues} \) and that \( \text{MinValue} \) and \( \text{MaxValue} \) satisfy the constraint of \( \text{Round-Signature} \).

\[
\begin{align*}
\text{FP} \_\text{Round1} & \quad \text{FP} \land \text{FP}' \\
\text{ApproxValues} & = \{\text{Finite} \mid \text{Format} = \text{Format'} \land \text{value}\} \\
\text{Preferred} & = \{\text{Finite} \mid \text{Format} = \text{Format'} \land \text{frac} \text{MOD} 2 = 0 \land \text{value}\} \\
\text{MinValue} & \in \{\text{Inf} \mid \text{Format} = \text{Format'} \land \text{sign} = 1 \land \text{value}\} \\
\text{MaxValue} & \in \{\text{Inf} \mid \text{Format} = \text{Format'} \land \text{sign} = 0 \land \text{value}\} \\
\end{align*}
\]

\[
\text{FP} \_\text{Round2} \triangleq \text{FP} \_\text{Round1} \setminus \{\text{ApproxValues, Preferred, MinValue, MaxValue}\}. \\
\]

\(^*\)Decorating the name of a schema with, e.g., a subscript 1 or a prime has the effect of decorating the names of the variables in the signature of that schema throughout.
The resulting exception 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:

\[
\begin{align*}
\text{Exceptions} &::= \text{inexact} \mid \text{overflow} \mid \text{underflow} \\
\text{Exception \_ Signatures} &::= r \mid \text{exceptions}' \mid \text{Exceptions} \cup \text{FP}'.
\end{align*}
\]

**Exception \_ Spec**

- \text{inexact} \in \text{exceptions}' \Rightarrow r \neq \text{value}'
- \text{overflow} \in \text{exceptions}' \Rightarrow \text{Inf}' \lor \exists \text{Inf} \bullet \text{abs} r \geq \text{abs value}
- \text{underflow} \in \text{exceptions}' \Rightarrow 0 \neq \text{abs} r < 2^{EMin'Bias'}
- \text{underflow} \in \text{exceptions}' \Rightarrow \text{Denorm}'

(The two alternative conditions under which underflow is included in the set exceptions' mean that there is a choice about which condition to implement.)

Finally, the whole specification is

\[
\text{FP \_ Round} \triangleq \text{FP \_ Round}2 \land \text{Exception \_ Spec}.
\]

**C. Addition, Subtraction, Multiplication, and Division**

In order to discuss these operators, they must be introduced into the mathematics:

\[
\text{Ops} ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \text{div}.
\]

The essential ingredients of an arithmetic operation are two numbers, \text{FP}_x \text{ and } \text{FP}_y, and an operation \text{op} : \text{Ops}; the number \text{FP}' is the result, and its format must be at least as wide as each of the operands:

\[
\text{Arit \_ Signature} \\
\text{FP}_x, \text{FP}_y, \text{op} : \text{Ops} \\
\text{FP}'
\]

wordlength' \geq \text{wordlength}_x \\
wordlength' \geq \text{wordlength}_y

When both \text{FP}_x \text{ and } \text{FP}_y \text{ are finite numbers, and division is not by zero, the specification is straightforward. A real number is specified, which can be rounded to give the correct result:}

\[
\text{Value \_ Spec} \\
\text{Arit \_ Signature} \land \text{Finite}_x \land \text{Finite}_y \\
r ::=
\]

\[
\begin{align*}
\text{op} = \text{add} \land r &= \text{value}_x + \text{value}_y \\
\text{op} = \text{sub} \land r &= \text{value}_x - \text{value}_y \\
\text{op} = \text{mul} \land r &= \text{value}_x \times \text{value}_y \\
\text{op} = \text{div} \land \neg \text{Zero}_x \land r &= \text{value}_x + \text{value}_y
\end{align*}
\]

If the result after rounding will be zero, some additional specification is necessary to define the sign completely [11, p. 14, sec. 6.3]. This is how it is described in [11]:

\[
\ldots \text{the sign of a product or quotient is the exclusive or of the operands' signs; the sign of a sum, or of a difference } x - y \text{ regarded as a sum } x + (-y), \text{ differs from at most one of the addends' signs.} \ldots
\]

These rules shall apply even when operands or results are zero or infinite.

When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be + in all rounding modes except round toward −∞, in which mode that sign shall be −. However, \(x + x = x - (-x)\) retains the same sign as \(x\) even when \(x\) is zero.

The mathematics is correspondingly elliptic:

\[
\text{Sign \_ Bit} \\
\text{Arit \_ Signature} \\
\begin{align*}
(-1)^{\text{sign}_x} \times \text{abs} r &= r \\
(\text{op} = \text{mul} \lor \text{op} = \text{div}) &\Rightarrow (-1)^{\text{sign}_y} = (-1)^{\text{sign}_x + \text{sign}_y} \\
(\text{op} = \text{add} \lor \text{op} = \text{sub}) \land r = 0 &\Rightarrow \\
\text{Zero}_x \land \text{Zero}_y, (\text{sign}_x = \text{sign}_y \Rightarrow \text{op} = \text{add}) &= \text{sign}' = \text{sign}_x \\
\neg (\text{Zero}_x \land \text{Zero}_y, (\text{sign}_x = \text{sign}_y \Rightarrow \text{op} = \text{add})) &\Rightarrow (\text{sign}' = 1 \Rightarrow \text{mode} = \text{ToNegInf})
\end{align*}
\]
For instance, when the operation is addition, the sign of the result is determined as follows:

\[
\begin{align*}
\text{Zero} \land \text{Zero} & \mid op = \text{add} \\
\leftarrow \text{sign}_r = 0 = \text{sign}_r = \text{sign}' = 0 \\
\text{sign}_r = 1 = \text{sign}_r = \text{sign}' = 1 \\
\text{sign}_r \Rightarrow \text{sign}_r = (\text{mode} \neq \text{ToNegInf} \Rightarrow \text{sign}' = 0) \\
\text{mode} = \text{ToNegInf} \Rightarrow \text{sign}' = 1
\end{align*}
\]

If the operation is subtraction, replace \( \text{sign}_r \) by \( 1 - \text{sign}_r \).

If one of the operands is not a number, then the result is not a number (the standard demands that the result be equal to the offending operand, but that is not possible if the destination format differs from that of the \( \text{NaN} \), [11, p. 13, sec. 6.2]):

\[
\begin{align*}
\text{NaN}_\text{Arit} & \\
\text{Arit}_\text{Signature} & \\
\text{NaN}_x & \lor \text{NaN}_x \\
\text{NaN}' &
\end{align*}
\]

Now, arithmetic with infinity is considered. This is defined to be the limit of finite arithmetic. However, certain cases do not have a limit, and these result in a \( \text{NaN} \):

\[
\begin{align*}
\text{Inf}_\text{Arit}_\text{Signature} & \\
\text{Arit}_\text{Signature} & \\
\neg (\text{NaN}_x \land \text{NaN}_x) & \\
\text{Inf} & \lor \text{Inf}
\end{align*}
\]

Rounding has already been described, so arithmetic on finite numbers may be defined by using \( \text{FP\_Round} \) to specify the relation of \( r \) to \( \text{FP}' \):

\[
\begin{align*}
\text{Fin}_\text{Arit} & \triangleq (\text{Value\_Spec} \land \text{FP\_Round} \land \text{Sign\_Bit}) \setminus \{r\}.
\end{align*}
\]

Division of a finite nonzero number by zero gives infinity; but division of zero by zero is not a number:

\[
\begin{align*}
\text{Div\_By\_Zero} & \\
\text{Arit\_Signature} & \\
\text{Finite}_x \land \text{Zero}_x & \\
op = \text{div} & \\
(\text{Zero}_x \land \text{NaN}'') \lor (\neg \text{Zero}_x \land \text{Inf}' \land (-1)^{\text{sign}'} = (-1)^{\text{sign}_r} \land \text{sign}')
\end{align*}
\]

*The schema \( S \setminus \{x\} \) is the schema \( S \) with \( x \) "hidden"; i.e., its value is filled in consistent with the specification and the variable is no longer free in the schema. Mathematically, it is existentially quantified: \( \forall x : X \leftrightarrow (S \setminus \{x\}) \equiv \exists x : X \ast S. \)
These partial specifications can be disjoined to give the complete specification of arithmetic with infinity:

\[ \text{Inf\_Arit} \triangleq \text{Inf\_Add\_Sub} \lor \text{Inf\_Mul} \lor \text{Inf\_Div}. \]

None of the exceptional cases returns the rounding exceptions; \( \text{No\_Round\_Exceptions} \) describes this, \( \text{Exc\_Arit} \) describes exceptional arithmetic, and \( \text{FP\_Arit} \) describes the complete relation on \( \text{Arit\_Signature} \):

\[
\begin{align*}
\text{No\_Round\_Exceptions} \triangleq & \text{exceptions}' \cdot \cdot \cdot \\
\text{Exc\_Arit} \triangleq & \text{No\_Round\_Exceptions} \land (\text{Div\_By\_Zero} \lor \text{NaN\_Arit} \lor \text{Inf\_Arit}) \\
\text{FP\_Arit} \triangleq & \text{Fin\_Arit} \lor \text{Exc\_Arit}. \\
\end{align*}
\]

Five different exceptions can occur during the operations. These cover all the different cases when the finite operations do not extend to infinite numbers, division by zero, and when one operand is not a number:

\[
\text{Arit\_Exceptions} ::= \text{NaN\_Op} \mid \text{mul\_Zero\_Inf} \mid \text{div\_Zero} \mid \text{div\_Inf\_Inf} \mid \text{Mag\_sub}
\]

Finally, the whole specification is:

\[ \text{Arit} \triangleq \text{FP\_Arit} \land \text{Exception\_Spec}. \]

III. Refinement

The development method of [5] and [6] is adopted, with some slight adaptations to the syntax. If \( S \) is some schema and \( V \) is a set of variables, we write \( S > V \) to specify a program which implements \( S \), but may only change the variables \( V \) ([6] uses the notation \( V \cdot S \)). Further, we write \( C < S \) for a program which implements \( S \) by only referring to the variables \( C \) ([6] takes the complementary approach of introducing nonprogram constants by the construct all \( c \cdot S \); the declaration used here corresponds to the introduction of all unprimed variables except those in \( C \) and \( V \)). As mathematical operations on schemas, \( C < S > V \) is the schema obtained by hiding all unprimed variables except those of \( C \) and \( V \) and all primed variables except those of \( V \).

Given a program \( P \) which implements \( R \) and \( Q \) which implements \( S \), the sequential composition \( P; Q \) satisfies the specification \( R; S \) where

\[ R; S \triangleq \exists V'.(R[V'/V] \land S[V'/V]) \land (\forall V'.R \Rightarrow \exists V''.S'), \]

in which \( V \) denotes the set of variables primed in \( R \) and unprimed in \( S \). To see what this expression means, consider the execution of the program \( P; Q \). Suppose the initial state of the machine is \( \sigma \) and that \( \sigma \) satisfies the precondition of \( R \), the state after execution of \( P \) is \( \sigma_0 \), and the final state is \( \sigma' \). Because \( P \) implements \( R \), it is certain that the values given to the variables by \( \sigma \) and \( \sigma_0 \) satisfy \( R[V_0/V'] \); if \( \sigma_0 \) satisfies the precondition of \( S \), then the values given by \( \sigma_0 \) and \( \sigma' \) satisfy \( S[V_0/V] \). Hence, if \( \sigma_0 \) satisfies the precondition of \( S \), there is some set of variables \( V_0 \) such that \( R[V_0/V'] \land S[V_0/V] \). The second part of the expression ensures that if \( \sigma \) satisfies the precondi-
sequential composition of the specifications of these four procedures. It is often the case that in deriving a program from the specification, the sequential components are derived “backwards” (i.e., the last first). This is because it is easier to derive the condition which the earlier components must achieve from the precondition of the later components. Exceptionally, in this case, it is simpler to decompose the specification first into unpacking, followed by the rest, and then to continue “backwards.”

A. Unpacking

The first stage in the algorithm is to unpack the numbers from their natural number representation into wsign, wexp, and wfrac fields, shifting in the implicit leading fraction bit of Normal numbers so that the new representation bears a relation to the value which is uniform across all the Finite numbers.

Unnormalized

| wsign', wexp', wfrac':0:2^wordlength - 1 |

Finite

| value = (-1)^sign * 2^wexp - bias - wordlength + 1 * wfrac' |
| wsign' = sign * 2^wordlength - 1 |
| wexp' = exp |

The procedure to unpack numbers is specified by

Unpack{nat} a \rightarrow [Finite \land Unnormalized \rightarrow [wsign, wexp, wfrac].

For simplicity, we introduce the convention that the primed and unprimed versions of variables which are not allowed to be altered by an implementation are equated in the specification (i.e., that they are defined to be constants). This means, for instance, that the specification of Unpack may be rewritten omitting the Z.

We wish to find a specification S such that the following refines the overall specification (i.e., the specification of what is left to do after unpacking):

VAR wsign, wexp, wfrac, wsign', wexp', wfrac': (Unpack, \land Unpack); S.

This specification turns out to be

\[
(\forall v : \text{Unnormalized} \land \text{Unnormalized} \rightarrow [\text{wsign}, \text{wexp}, \text{wfrac}].
\]

This is obtained by using the rule for variable introduction. If the variable v does not appear in the schema S, then it may be introduced as a fresh variable, which may be altered by an implementation:

\[
C < S \triangleright V \ \forall v : C < S \triangleright (V \cup \{v\}).
\]

The rule for weakest postspecification (one of the rules available for the introduction of sequential composition) is as follows. The schema S \triangleleft V gives the weakest condition under which specified final values of the program variables V may be found. The schema T describes the first sequential component. If the precondition of T is at least as weak as the precondition of S, then

\[
C < S \triangleright V \ \rightarrow (C < T \triangleright V);
\]

The rule for eliminating program variables is as follows. A program which alters fewer variables is better (i.e., satisfies more specifications) than one which alters more:

\[
C < S > (V \cup \{v\}) \subseteq C < S > V.
\]

To show that the refinement is correct, the first rule is used to introduce the variables wsign, etc. Taking T to be Unnormalized, \land Unnormalized, in the second rule, its precondition is Finite, \land Finite. The precondition of the whole specification turns out to be

\[
\text{Finite, } \land \text{Finite, } \neg (op = \text{div } \land \text{Zero}).
\]

Finally, the third rule is employed to remove nat and exceptions from the variables changed by the first sequential component, and since this component no longer refers to mode, it may be safely removed from the set to which it refers.

B. Rounding

We now proceed to determine what information the rounding procedure requires. 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 an integer:

\[
\text{FP, FP' } \text{Format} = \text{Format'} \land \neg (\text{NaN } \lor \text{NaN'})
\]

\[
\neg \text{abs value} \leq \text{abs value'} \Rightarrow
\]

\[
\text{nat MOD } 2^\text{wordlength - 1} \leq \text{nat'} \text{MOD } 2^\text{wordlength - 1}.
\]

The substitution, e.g., [\_ \_ \_ \_] is used to indicate that 0-subscripted variables have the subscript removed.

\^[1] The schema ZS is defined to be S: S' \rightarrow S'; i.e., the variables of S are constant.
This can be used to see that the number of least modulus with a modulus greater than a given finite number is obtained by incrementing its representation as an integer:

The variable \textit{guard} has the value 0 precisely when the approximation with less modulus is closer; \textit{sticky} has the value 0 precisely when both approximations are equally.

\[
\begin{align*}
\text{Succ} & \quad FP; FP_0 \\
\text{Finite} \land \text{Format} & = \text{Format}_0 \\
\text{abs value} & < \text{abs value}_0 \\
\forall FP' \mid \text{abs value} < \text{abs value}' \bullet \text{abs value}_0 \leq \text{abs value}'
\end{align*}
\]

\[
FP; FP_0 \mid \text{Finite} \land \text{Format} = \text{Format}_0 \land \text{nat}_0 = \text{nat} + 1 \Rightarrow \text{Succ}.
\]

From this result, the consistency of \textit{Preferred} in Section II-B can be deduced.

In turn, this means that if the approximation of less modulus is known, only enough extra information to determine the four predicates in \textit{RoundToNearest} is needed to return the correct value. The familiar \textit{guard} and \textit{sticky} bits, defined below, contain this information:

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

This is, however, not quite enough information to return the correct overflow condition. If \( r \geq 2^{E\text{Max}' - \text{Bias}'} \), this information is lost. Conversely, it is not possible to determine the overflow condition before rounding, as the condition \textit{Inf} cannot be tested until the final result is calculated. Thus, it is necessary to divide \textit{Exception Spec} into two parts. The \textit{inexact} and \textit{underflow} conditions can be determined before or after rounding. The design decision is made so that as many exception conditions as possible will be determined after rounding in order that the precondition of the module be simpler.

\[
\begin{align*}
\text{Exception Before} & \\
\text{Exception Signature} & \\
\text{overflow} & \in \text{exceptions}' \quad \land \quad \text{abs } r \geq 2^{E\text{Max}' - \text{Bias}'}
\end{align*}
\]

\[
\begin{align*}
\text{Exception After} & \\
\text{Exception Signature; exceptions:} & \mathbb{N} \land \text{Exceptions} \\
\text{overflow} & \in \text{exceptions} \quad \land \quad \text{abs } r \geq 2^{E\text{Max}' - \text{Bias}'} \\
\text{inexact} & \in \text{exceptions}' \quad \land \quad r \neq \text{value}' \\
\text{overflow} & \in \text{exceptions}' \quad \land \quad \text{overflow} \in \text{exceptions} \lor \text{Inf}' \\
\text{underflow} & \in \text{exceptions}' \quad \land \quad \text{Denorm}'
\end{align*}
\]
Using the weakest post specification rule with the schema $T$ being $\text{Exception\_Before}$ with the primes removed from its variables, the following is seen to be a valid refinement:

$$\text{Exception\_Spec} \triangleq (\text{Exception\_Before}; \text{Exception\_After}).$$

If we have the approximation of less modulus, the guard and sticky bits, and an overflow indication, there is enough information to determine the correct result and the correct exception conditions. Thus, a real number may be represented prior to rounding as follows:

$$\text{Packed} \triangleq (\text{AFP} \bullet \text{Bounds}) \land \text{External}\,$$

$$\land \text{Exception\_Before}[\_.\text{'-'}].$$

The rounding procedure may be specified as follows:

$$\text{Round} \triangleq \{\text{guard}, \text{sticky}, \text{mode}\}$$

$$\text{< Packed } \land \text{FP\_Round } \land \text{Exception\_After } \triangleright$$

$$\{\text{nat, exceptions}\}.$$ 

We require a specification $S$ such that the following is a refinement of the remaining specification $(\ast)$:

$$\text{VAR guard, sticky: } S; \text{ Round.}$$

This turns out to be

$$\{\text{nat, nat, mode}\}$$

$$\text{< Value\_Spec } \land \text{Sign\_Bit } \land \text{Packed' } \triangleright$$

$$\{\text{wfrac, wexp, wsign, wsign, wexp, wfrac, guard, sticky, nat, exceptions}\}.$$ 

$$\text{FinArit} \triangleq \{\text{nat, wsign, wexp, wfrac, nat, wsign, wexp, wfrac, mode}\}$$

$$\text{< (Unnormalized; Unnormalized; Normal') } \land \text{Value\_Spec } \land \text{Sign\_Bit } \triangleright$$

$$\{\text{wsign, wexp, wfrac}\}.$$ 

It is obtained as follows. First, note that the hiding of the variable $r$ in $\text{Fin\_Arit}$ may be removed since it is hidden by the $\text{<, >}$ construct. We take $\text{Value\_Spec }\land \text{Sign\_Bit }\land \text{Packed'}$ as the first component schema ($T$ in the formalization of the rule). The conjuncts $\text{Value\_Spec}$ and $\text{Sign\_Bit}$ are redundant in the second component because there is enough information in $\text{Packed}$ to produce the correct result.

C. Packing

This representation $\text{Packed}$ is too complicated for the immediate result of a calculation: we require a form which has a sign, exponent, and fraction, but which contains enough information to produce a $\text{Packed}$ number. If the exponent is considered to be unbounded above (this assumption causes no problems since the largest exponent which can be produced from finite arithmetic is less than $2^{\text{wordlength}}$) and we demand that the fraction be at least $2^{\text{wordlength} - 1}$ when the exponent is not $\text{EMin}$, a condition for an extra digit of accuracy is easy to formulate. The condition given here is stronger than necessary, but simpler than the weakest condition:

$$\text{Normal}$$

$$\begin{align*}
 r & \triangleq \text{Unsimplified} \land \text{Unsimplified} \\
 wsign & , \text{wexp, wfrac}: 0, 2^{\text{wordlength}} - 1 \\
 \text{wexp} & \geq \text{EMin} \\
 \text{wexp} & > \text{EMin} \Rightarrow \text{wfrac} \geq 2^{\text{wordlength} - 1} \\
 \text{abs (approx - exact)} & < 1 \\
 \text{abs exact} & \geq \text{abs approx} \\
 \text{nonint approx} & = 0 \Rightarrow \text{nonint exact} = 0 \\
 \text{where approx} & = (-1)^{\text{wsign}} \times 2^{1 - \text{expwidth}} \times \text{wfrac} \\
 \text{exact} & = 2^{\text{sign}} \times \text{wexp} + 2^{\text{fractwidth}} \times r
\end{align*}$$

The procedure to convert a $\text{Normal}$ number to a $\text{Packed}$ number is specified by

$$\text{Pack} \triangleq \{\text{wsign, wexp, wfrac}\}$$

$$\text{< Normal; Packed’ } \triangleright$$

$$\{\text{nat, guard, sticky, exceptions}\}.$$ 

and as before, we wish to find a suitable sequential refinement of the remaining specification.

D. Arithmetic

We claim the required refinement is

$$\text{FinArit; Pack}$$

where

This time, the first component schema is the body of $\text{FinArit}$, and the reasoning is similar to that for the last refinement.

So the total refinement that has been produced is

$$\text{VAR wsign, wexp, wfrac, wsign, wexp, wfrac; (Unpack, Unpack; )}$$

$$\text{VAR guard, sticky; FinArit; Pack; Round}$$

and since $\text{FinArit}$ does not alter or refer to the variables guard or sticky, this may be transformed to

$$\text{VAR wsign, wexp, wfrac, wsign, wexp, wfrac; (Unpack, Unpack; )}$$

$$\text{FinArit;}$$

$$\text{VAR guard, sticky; Pack; Round.}$$
Now, the components of the decomposed specification may be transformed into implementations independently of each other in a similar manner.

IV. Conclusions

It is often heard said that formal methods can be applied only to practically insignificant problems, that development costs in large products are too high, and that the desired reliability is still not achieved. The problem presented here is only a part of a large body of work which has been undertaken to implement a proven-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 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, from a simple mistype in the program, or from a genuine error in the proof. Because of the steps in the development, the effect of this can be limited. Either a fragment of the 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 her 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 exception conditions were to change, it would be possible to prove that only the second part of the rounding module and, perhaps, its precondition need be changed. The modifications can take place without having to resort to various pieces of code. Likewise, in the development stage, the formalism exists 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 nonalgorithmic 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 formalize seemingly intractable descriptions. This paper has assumed some familiarity with [11], 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.

ACKNOWLEDGMENT

Thanks are due to D. Shepherd, M. H. Goldsmith, C. Morgan, A. W. Roscoe, C. A. R. Hoare and J. C. P. Woodcock for comments and encouragement.

REFERENCES


Geoff Barrett received the B.A. degree in mathematics and the M.Sc. degree in computation from Oxford University in 1984 and 1985, respectively.

He has been employed as a Research Officer at the Programming Research Group, Oxford University, since 1985. His main research interests lie in the fields of software and hardware verification and the semantics of parallelism.