Defining the IEEE-854 Floating-Point Standard in PVS

Paul S. Miner
P.S.Miner@LaRC.NASA.Gov

Abstract

A significant portion of the ANSI/IEEE-854 Standard for Radix-Independent Floating-Point Arithmetic is defined in PVS (Prototype Verification System). Since IEEE-854 is a generalization of the ANSI/IEEE-754 Standard for Binary Floating-Point Arithmetic, the definition of IEEE-854 in PVS also formally defines much of IEEE-754. This collection of PVS theories provides a basis for machine checked verification of floating-point systems. This formal definition illustrates that formal specification techniques are sufficiently advanced that it is reasonable to consider their use in the development of future standards.

KEYWORDS: Floating-point arithmetic, Formal Methods, Specification, Verification.

1　Introduction

This document describes a definition of the ANSI/IEEE-854 [3] Standard for Radix-Independent Floating-Point Arithmetic in the PVS verification system (developed at SRI International) [4]. IEEE-854 is a generalization of the ANSI/IEEE-754 [2] Standard for Binary Floating-Point Arithmetic. Therefore, this formalization of the IEEE-854 can be instantiated to serve as a basis for the formal specification of the more widely used IEEE-754 standard. All that is required is to instantiate the general theory with the appropriate constants, and define the representation formats in accordance with IEEE-754.

This is not the first formalization of an IEEE standard for floating-point arithmetic. Geoff Barrett [1] describes the Z formalization of IEEE-754 used in the development of the INMOS T800 Transputer. Z is a formal specification language with limited mechanized support. The specification presented here uses the PVS specification language which is tightly integrated with the PVS mechanized proof system. Also, the specification presented here is of IEEE-854, not IEEE-754. This formalization in PVS was not based upon the Z specification.

This document will present those portions of the standard that have been defined in PVS. The various features of PVS will be described at the time of their first use. This report highlights some areas of imprecision in the standard and illustrates that formal techniques are sufficiently advanced to consider their use in the development of future standards.

2　Basic Definitions

The document IEEE-854 (hereafter referred to as the standard) describes a parameterized standard for floating-point arithmetic. This section will present the definition of floating-point numbers and introduce mappings between floating-point and real numbers. The standard allows the definition of four precisions of floating-point numbers: single, single extended, double and double extended. Each precision is distinguished by the range of representable values and the number of significant digits. The PVS theories define a fixed, but undetermined precision. It is simple to define any combination of precisions by importing multiple instances of the top-level PVS theory presented here.

2.1　Sets of Values

Section 3.1 of the standard defines the parameters:

> *Four integer parameters specify each precision:*
>
> $$\begin{aligned} b &= \text{\textit{the radix}} \\ p &= \text{\textit{the number of base-b digits in the significand}} \\ E_{max} &= \text{\textit{the maximum exponent}} \\ E_{min} &= \text{\textit{the minimum exponent}} \end{aligned}$$
>
> *The parameters are subject to the following constraints:*
>
> 1. *b shall be either 2 or 10 and shall be the same for all supported precisions*
> 2. *$(E_{max} - E_{min})/p$ shall exceed 5 and should exceed 10*
> 3. *$b^{p-1} \geq 10^5$*

> *The balance between the overflow threshold ($b^{E_{max}+1}$) and the underflow threshold ($b^{E_{min}}$) is characterized by their product ($b^{E_{max}+E_{min}+1}$), which should be the smallest integral power of b that is ≥ 4. [3, page 8]*

From these constraints, it is clear that $b > 1$, $p > 1$, and that $E_{max} > E_{min}$. However, the last quoted sentence on balance between the overflow and underflow thresholds is only a suggestion[1] and need not be followed for an implementation to be compliant. In later sections, we will highlight some consequences of not having a balanced exponent range.

In PVS, these constraints can be defined as follows:

```
IEEE_854 [b,p:above(1),E_max,E_min:integer]: THEORY
  BEGIN

  ASSUMING
    Base_values: ASSUMPTION b=2 or b=10
    Exponent_range: ASSUMPTION (E_max - E_min)/p > 10
    Significand_size: ASSUMPTION b^(p-1)>=10^5
    E_balance: ASSUMPTION
        IF b < 4 THEN E_max + E_min = 1 ELSE E_max + E_min = 0 ENDIF
  ENDASSUMING

  Exponent_balance: LEMMA b^(E_max+E_min) <4 & 4<=b^(E_max+E_min+1)

  E_max_gt_E_min: LEMMA E_max > E_min

  E_min_neg: LEMMA E_min<0

  E_max_pos: LEMMA E_max>0

  IMPORTING IEEE_854_defs[b,p,E_max,E_min]

  END IEEE_854
```

This theory definition has four formal parameters, which correspond to the requirements of the standard. For a fixed n, the type $above(n)$ is defined in the PVS prelude as $\{i : nat | i > n\}$, thus by declaring b and p to be of type $above(1)$, we have $b > 1$ and $p > 1$. The PVS ASSUMING section states the additional constraints on these parameters. These assumptions define proof obligations for any theory that imports IEEE_854. This specification includes the optional constraints given by the standard. A minimally compliant specification would modify assumption Exponent_range and remove both assumption E_balance and lemmas Exponent_balance, E_min_neg, and E_max_pos. The last line of the PVS theory imports the remaining definitions and declarations to complete the specification of floating-point arithmetic for a fixed precision (e.g. one of single, double, single extended, or double extended).

None of the underlying definitions depend directly on the assumptions given in the assuming section, so the theories defining the rest of the standard will use the weaker assumptions that $b > 1$, $p > 1$, and that $E_{max} > E_{min}$. We will not assume that $E_{max} \geq 0$ or that $E_{min} \leq 0$, since these

---

[1] The distinction is between *should* and *shall*. Usage of the word *should* indicates a suggestion as opposed to a requirement.

are not consequences of the required constraints. This will have a limited impact on the remaining definitions.

Section 3.1 of the standard continues:

Each precision allows for the representation of just the following entities:

1. Numbers of the form $(-1)^s b^E (d_0.d_1 d_2 \cdots d_{p-1})$ where

$$
\begin{aligned}
s &= \text{ an algebraic sign} \\
E &= \text{ any integer between } E_{min} \text{ and } E_{max}, \text{ inclusive} \\
d_i &= \text{ a base-b digit } (0 \le d_i \le b - 1)
\end{aligned}
$$

2. Two infinities, $+\infty$ and $-\infty$

3. At least one signaling NaN

4. At least one quiet NaN [3, page 8]

Item 1 has a slight ambiguity concerning the definition of s. If s is defined as an algebraic sign (e.g. one of $\{+, -\}$), then the expression $(-1)^s$ has no meaning. The PVS specification adopts the definition from IEEE-754 [2], that is, $s \in \{0, 1\}$. This is the most natural choice for s, but several other numeric encodings possess the necessary properties. For example, s could be a base-b digit where an even value denotes positive and an odd value denotes negative.

The PVS specification of values defines a floating-point number (fp_num) using the PVS abstract datatype mechanism [5][2]:

```
IEEE_854_values
  [b,p:above(1),
   E_max:integer,
   E_min:{i:integer | E_max > i}]: THEORY

  BEGIN

  sign_rep: type = {n:nat | n = 0 or n = 1}
  Exponent: type = {i:int | E_min <= i & i <= E_max}
  digits: type = [below(p)->below(b)]

  NaN_type: type = {signal, quiet}
  NaN_data: NONEMPTY_TYPE

  fp_num: datatype
   begin
    finite(sign:sign_rep,Exp:Exponent,d:digits):finite?
    infinite(i_sign:sign_rep): infinite?
    NaN(status:NaN_type, data:NaN_data): NaN?
   end fp_num
```

---

[2]This theory has no assuming section, but there is an explicit assumption in the formal parameters. E_{min} is defined via the dependent type mechanism to be strictly less than E_{max}. By capturing this information in the type of E_{min}, the corresponding importing tccs are trivially satisfied (and hence, not generated). If we used an assuming clause, there would be an explicit proof obligation at each level of the importing hierarchy.

```
[...]
```

The definition of datatype `fp_num` states that the type of floating-point numbers is the disjoint union of three sets: finite numbers, infinite numbers, and Not a Numbers (NaNs). A finite number can be constructed (using constructor `finite`) from an algebraic sign, an integer exponent, and a significand[3]; an infinity can be constructed from an algebraic sign; and a NaN can be constructed from a status flag (i.e. signal or quiet) and data undetermined by the standard.

2.2   Mapping floating-point numbers to reals

The standard implies an intended semantics for the representable numeric values. The function `value` maps the finite floating point numbers to the reals as implicitly specified in the standard. In PVS, reals are treated as a base type. There is no need to import a library of definitions for real arithmetic.

```
fin : var (finite?)

value_digit(d:digits)(n:nat):nonneg_real =
    if n < p then d(n) * b ^ (-n) else 0 endif

value(fin): real =
  (-1) ^ sign(fin) * b ^ Exp(fin) * Sum(p, value_digit(d(fin)))
```

Here, `fin` is declared to be a variable of type (`finite?`), that is, an element of the subtype of `fp_num` that satisfies the predicate `finite?`[4]. Function `value_digit` takes a collection of digits[5] and an index for a particular digit and returns the usual base-b interpretation of a digit determined by its position in the significand. Finally, `value` defines the interpretation of the sign field, the exponent, and sums the values of the digits in the significand. Function `Sum` is defined in a separate PVS theory.

The standard recognizes that this scheme encodes some values redundantly. Furthermore, an implementation may use redundant encodings, so long as it does not distinguish redundant encodings of nonzero numbers. The standard subdivides the encodings into three groups using the following definitions:

> **normal number** *A nonzero number that is finite and not subnormal.*
>
> **subnormal number** *A nonzero floating-point number whose exponent is the precision's minimum and whose leading significant digit is zero.* [3, Section 2, page 8]

These definitions divide the finite numbers into three groups: those that denote zero (i.e. any finite `fp_num` with a significand of all zeros), the subnormal numbers, and the normal numbers. The definitions given above are imprecise. Consider the following finite number:

$$(-1)^0 \times b^{E_{min}+1} \times 0.01\cdots0 \qquad (= b^{E_{min}-1})$$

---

[3] The significand consists of an indexed collection of p base-b digits. The most natural way to define this in PVS is by function type $[\text{below}(p) \to \text{below}(b)]$. The PVS prelude defines $\text{below}(n) : \text{TYPE} = \{i : nat | i < n\}$.

[4] The predicate `finite?` is determined by the `fp_num` datatype declaration.

[5] The PVS specification language is an extended version of higher-order logic. Functions are first-class objects and can be passed as parameters or be defined as the return type of other functions.

This is clearly nonzero. Since the exponent is not the precision's minimum, a strict interpretation of the definition of subnormal may lead us to conclude that this must be a normal number. However,

$$b^{E_{min}-1} = (-1)^0 \times b^{E_{min}} \times 0.1 \cdots 00$$

thus it is equal to a subnormal number so it must also be subnormal. Before we can make the distinctions precise, we need to determine a canonical encoding for each finite floating-point number. The function `normalize` maps each finite `fp_num` to a canonical representative:

```
shift_left(d: digits): digits =
  LAMBDA (i: below(p)): IF (i + 1 = p) THEN 0 ELSE d(i + 1) ENDIF

normalize(fin:(finite?)): recursive (finite?) =
  IF Exp(fin) = E_min or d(fin)(0) /= 0 then
      fin
  ELSE
      normalize(finite(sign(fin),Exp(fin)-1,shift_left(d(fin))))
  ENDIF
measure lambda (fin:(finite?)): Exp(fin) - E_min
```

Recursive[6] function `normalize` repeatedly shifts the significand left one digit and decrements the exponent until either the exponent is the precision's minimum or the most significant digit is nonzero. Since our goal in defining function `normalize` is to map each finite number to a canonical representative, we must prove that the result of this function preserves the value of its argument. The following lemma has been proven in PVS:

```
normal_value: LEMMA
    value(fin) = value(normalize(fin))
```

We can now make the distinctions between finite numbers precise. We do this by defining three predicates: `zero?`, `normal?`, and `subnormal?`.

```
zero?(fp:fp_num):bool =
  IF finite?(fp) THEN value(fp)=0 ELSE FALSE ENDIF

normal?(fp: fp_num): bool =
  IF finite?(fp) THEN d(normalize(fp))(0) > 0 ELSE FALSE ENDIF

subnormal?(fp: fp_num): bool =
  IF finite?(fp) THEN not zero?(fp) &
                      Exp(normalize(fp)) = E_min &
                      d(normalize(fp))(0) = 0
  ELSE FALSE
  ENDIF
```

We can provably partition the finite numbers into three sets:

---

[6]PVS requires that all functions be total, so any definition by recursion involves a proof that the recursion terminates. The evidence needed for this proof is given by a measure function that must decrease in each recursive call according to a well-founded relation. The default well-founded relation is '<' defined on the natural numbers.

```
finite_cover: LEMMA zero?(fin) OR normal?(fin) OR subnormal?(fin)

finite_disjoint1: LEMMA NOT (zero?(fin) & normal?(fin))

finite_disjoint2: LEMMA NOT (zero?(fin) & subnormal?(fin))

finite_disjoint3: LEMMA NOT (normal?(fin) & subnormal?(fin))
```

Lemma `finite_cover` states that every finite floating-point number either zero, normal, or subnormal. The remaining lemmas assert that these sets are mutually disjoint.

There are several simple lemmas that can be proven about `value`. We introduce the following definitions for the maximum and minimum representable values within a precision:

```
max_significand:digits =
   (lambda (i:below(p)): b-1)
min_significand: digits =
   (lambda (i: below(p)): IF i < p - 1 THEN 0 ELSE 1 ENDIF)
d_zero: digits = lambda (i: below(p)): 0

pos : sign_rep = 0
neg : sign_rep = 1

max_fp_pos : fp_num = finite(pos,E_max,max_significand)
min_fp_pos : fp_num = finite(pos,E_min,min_significand)
pos_zero   : fp_num = finite(pos,E_min,d_zero)
```

With these definitions we can prove in PVS that the function `value` returns the correct value for these floating-point numbers.

```
max_fp_correct: LEMMA
    value(max_fp_pos) = b ^ (E_max + 1) - b ^ (E_max - (p - 1))

min_fp_correct: LEMMA
    value(min_fp_pos) = b ^ (E_min - (p - 1))

value_of_zero: LEMMA
    value(pos_zero) = 0
```

Function `value` only specifies part of the relationship between reals and floating-point numbers. It serves to interpret finite floating-point numbers as reals. The next section addresses mapping reals to floating-point numbers.

2.3 Mapping reals to floating-point numbers

To map reals to floating-point, we define the following functions:

```
sign_of(r:real): sign_rep = IF r < 0 THEN neg ELSE pos ENDIF
Exp_of(px:posreal): {i:int| b^i <= px & px < b^(i+1)}
truncate(E:integer,nnx:nonneg_real): digits =
     (lambda (i:below(p)): mod(floor(nnx/(b^(E-i))),b))
```

Function `sign_of` returns the algebraic sign of a real number, adopting the convention that the sign of 0 is positive. Function `Exp_of` is completely defined using the dependent type and predicate subtype features of PVS. The range type of `Exp_of` depends on its argument `px`, and is constrained to be an integer that satisfies the given predicate. PVS generates a type-correctness condition (TCC) for this definition.[7] The TCC is discharged by showing that for all positive reals `px` there is an integer that satisfies the predicate. Function `truncate` uses the `mod` and `floor` functions to determine each digit in the significand for a given exponent `E`. These three functions allow us to define the following conversion from real numbers to floating-point numbers:

```
real_to_fp(r): fp_num =
  IF abs(r) >= b^(E_max+1) THEN
     infinite(sign_of(r))
  ELSIF abs(r) < b^E_min THEN
     finite(sign_of(r), E_min, truncate(E_min, abs(r)))
  ELSE
     finite(sign_of(r), Exp_of(abs(r)), truncate(Exp_of(abs(r)), abs(r)))
  ENDIF
```

Function `real_to_fp` converts an arbitrary real into a floating-point representation. If the real is outside the range of representable values, an appropriately signed infinity is returned. If the real is too small to be represented, it gets mapped to an appropriately signed zero. This definition provides an approximation of reals by rounding toward zero. However, the standard calls for four different rounding modes. The next section describes the various rounding mode and shows how this definition may be used in a general conversion from reals to floating-point.

3 Rounding

Floating-point numbers serve as a computable approximation of real numbers. The standard specifies four means of approximating reals by floating-point numbers. The user has the ability to select the rounding mode from among these four. In Section 4, the standard states:

> ...every operation specified in section 5 shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then that result rounded according to one of the modes in this section. [3, page 9]

The operations in section 5 referenced by this clause consist of the basic arithmetic operations: add, subtract, multiply, divide; remainder and square root; comparisons; conversions between precisions; and conversions to integers and integer valued floats. The discussion of rounding modes will be done in conjunction with the specification of the operations given in section 5 of the standard. The default mode is round to nearest. The standard states:

> An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant digit even shall be delivered. [3, Section 4.1, page 9]

In addition, the standard continues:

---

[7]A TCC is a proof obligation generated by PVS that is sufficient to show that a given term is well typed. PVS has an undecidable type system, so sometimes the user must provide a proof that a term is correctly typed.

An implementation of this standard shall also provide three user-selectable directed rounding modes: round towards +∞, round towards −∞, and round towards 0. [3, Section 4.2, page 9]

These rounding modes will first be defined for conversion of reals to integers.

3.1 Floating-point to integer

The simplest rounding operation to consider is converting a floating-point number to an integer (or to an integral valued floating-point number). From Section 2.1, we defined function `value` to map a floating-point number to a real. All that remains is to convert this real to an integer. Section 5.4 of the standard states:

Conversion to integer shall be effected by rounding as specified in Section 4. [3, page 10]

Section 5.5 adds:

It shall be possible to round a floating-point number to an integral valued floating-point number in the same precision. [3, page 10]

The four rounding modes are specified as an enumerated type in PVS, leading to the following definition of function `round`:

```
sgn(r:real): int = IF r >= 0 THEN 1 ELSE -1 ENDIF

round(r:real,mode:rounding_mode): integer =
  CASES mode of
    to_nearest: round_to_even(r),
    to_zero:    sgn(r) * floor(abs(r)),
    to_pos:     ceiling(r),
    to_neg:     floor(r)
  ENDCASES
```

This definition makes use of the floor, ceiling, and absolute value functions to define the directed roundings. Round to nearest requires an additional function definition.

```
round_to_even(r:real): integer =
  IF r - floor(r) < ceiling(r) - r THEN floor(r)
  ELSIF ceiling(r) - r < r - floor(r) THEN ceiling(r)
  ELSIF floor(r) = ceiling(r) THEN floor(r)
  ELSE 2 * floor(ceiling(r) / 2)
  ENDIF
```

Function `round_to_even` rounds an arbitrary real to the nearest integer. The typical cases are defined using the integer floor and ceiling functions. The difficult case is the fourth alternative where the fractional part of r is $1/2$. The expression

$$2 \times \left\lfloor \frac{\lceil r \rceil}{2} \right\rfloor$$

will round any real number r to the nearest even integer.

To demonstrate the correctness of these definitions, the following lemmas have been proven in PVS:

8

```
round_to_even1: LEMMA
   abs(r - round_to_even(r)) <= 1 / 2


round_to_even2: LEMMA
   abs(r - round_to_even(r)) = 1 / 2
       => integer_pred(round_to_even(r) / 2)


round1: LEMMA abs(r - round(r,mode)) < 1
```

Two of these lemmas illustrate the correctness of the definition of **round_to_even**. The first states that **round_to_even**(r) has an approximation error of at most $1/2$. The second states that when it is in error by exactly $1/2$, **round_to_even** returns an even integer. In addition, Lemma **round1** illustrates the correctness of **round**. The approximation error is always less than 1.

We can use **round** to define a function mapping a finite floating-point to an integer (and to an integral valued floating-point number):

```
fp_to_int(fin,mode):integer = round(value(fin),mode)


fp_to_int_fp(fin,mode): fp_num =
     real_to_fp(round(value(fin),mode))
```

Ideally, function **fp_to_int_fp** should return an object of type (**finite?**). However, since there are no constraints to ensure that $E_{max} \geq p$, it is not possible to prove the resulting TCC.[8]

3.2 Rounding reals

The function **round** can be used in conjunction with **real_to_fp** to define a general rounding function in accordance with the standard. It is necessary to first scale a nonzero real so that its scaled value is between b^{p-1} and b^p. Function **round** can be used to adjust the scaled value accordingly, and the result can be scaled back to its original magnitude. The resulting real will have at most p significant base-b digits, so **real_to_fp** can be used to map it into an appropriate floating-point representation. The standard describes a number of different possible return values if r is outside the range of **normal** floating-point numbers. It is not practical to introduce all the possible scenarios here. The exceptional cases will be presented in a later section of this paper.

```
scale(px):{i:int|b^(i+p-1)<=px & px < b^(i+p)} = Exp_of(px)-(p-1)

scale_correct: lemma b^(p-1)<=px/b^scale(px) & px/b^scale(px)< b^p

over_under?(r): bool = (r/=0 & (abs(r)>max_pos or abs(r)<b^E_min))

round_scaled(r:nzreal,mode:rounding_mode): real =
  b^(scale(abs(r)))*round( b^(-scale(abs(r)))*r,mode)

fp_round(r, mode): real =
```

---

[8]If $E_{max} < 0$, then the maximum representable floating-point number may round to 1. Function **real_to_fp** maps 1 to $+\infty$ in this case. More generally, if $E_{max} < p$, a floating-point converted to an integer using the rounding modes may map back to ∞. From this observation, one may conclude that a reasonable instance of this standard should adhere to the suggested constraint on a balanced exponent range.

```
IF r = 0 THEN 0
ELSIF over_under?(r) then
  round_exceptions(r,mode)
ELSE round_scaled(r,mode)
ENDIF
```

Function `scale` has a type defined in the same manner as `Exp_of` (Section 2.3), however, it also has a body explicitly defining the function. Function `fp_round` performs the necessary scaling to appropriately round an arbitrary real. The following lemmas show that these definitions have the desired properties:

```
round_0: LEMMA fp_round(0, mode) = 0

round_error: LEMMA
      r /= 0 & NOT over_under?(r)
        => abs(r - fp_round(r, mode))
             <  b ^ (Exp_of(abs(r)) - (p - 1))

round_near:  LEMMA
      r /= 0 & NOT over_under?(r)
        => abs(r - fp_round(r, to_nearest))
             <= b ^ (Exp_of(abs(r)) - (p - 1)) / 2

round_pos:   LEMMA
      NOT over_under?(r) => fp_round(r, to_pos) >= r

round_neg:   LEMMA
      NOT over_under?(r) => fp_round(r, to_neg) <= r

round_zero:  LEMMA
      NOT over_under?(r)
        => abs(fp_round(r, to_zero)) <= abs(r)
```

Lemma **round_0** shows that rounding 0 returns 0 regardless of the rounding mode. Lemma **round_error** shows that the approximation error is less than one "least significant digit". Lemma **round_near** states that the approximation error for mode **to_nearest** is $\leq$ one-half the least significant digit. Lemmas **round_pos**, **round_neg**, and **round_zero** show that **fp_round** rounds in the proper direction for these modes.

4   Operations

Section 5 of the standard states:

> All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, ...
>
> ..., each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's precision. [3, page 10]

Based on this description, the PVS description will define the operations using the corresponding real arithmetic functions.

4.1 Arithmetic definitions

The PVS definitions of the four basic arithmetic operations are similar. The enumerated type

```
fp_ops : type = {add, sub, mult, div}
```

simplifies the definition of these functions. The basic definition for an arithmetic operation is illustrated by the following definition for fp_add; the definitions for fp_sub and fp_mult are nearly identical.

```
fp_add(fp1, fp2, mode): fp_num =
  IF finite?(fp1) & finite?(fp2) THEN fp_op(add, fp1, fp2, mode)
  ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan(add, fp1, fp2)
  ELSE fp_add_inf(fp1, fp2)
  ENDIF
```

The function definition invokes one of three functions depending on the arguments. If both arguments are finite, then this function invokes the corresponding real function applied to the values of the arguments. If one argument is a NaN, then the rules for operations on NaNs are invoked. When one of the arguments is infinite, the result required by the standard is returned. Each of these cases will be described in more detail in the following sections.

The definition of division is a little more complicated, in that division by zero requires special treatment:

```
fp_div(fp1, fp2, mode): fp_num =
  IF finite?(fp1) & finite?(fp2)
    THEN IF zero?(fp2)
      THEN IF zero?(fp1)
        THEN invalid                      %raise invalid
      ELSE infinite(mult_sign(fp1, fp2)) %raise divide_by_zero
      ENDIF
    ELSE fp_op(div, fp1, fp2, mode)
    ENDIF
  ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan(div, fp1, fp2)
  ELSE fp_div_inf(fp1, fp2)
  ENDIF
```

If the second argument is zero and the first is not, then the function returns an appropriately signed infinity (and will later be modified to raise the divide by zero exception). If both operands are zero, the invalid exception is raised (here denoted by an arbitrary NaN named invalid). Otherwise, division reduces to the same basic format as the other operators.

4.1.1 Arithmetic with finite operands

When both operands are finite, the formal specification of floating-point arithmetic consists of converting the finite floating-point numbers to real numbers, performing the appropriate arithmetic function, and then converting the resulting real number back to floating-point format. The following PVS text accomplishes this:

```
apply(op,fin1,(fin2:fin| div?(op) => not zero?(fin))): real =
  cases op of
    add:  value(fin1) + value(fin2),
    sub:  value(fin1) - value(fin2),
    mult: value(fin1) * value(fin2),
    div:  value(fin1) / value(fin2)
  endcases

fp_op(op, fin1, (fin2: fin | div?(op) => NOT zero?(fin)), mode): fp_num =
  LET r = fp_round(apply(op, fin1, fin2), mode)
    IN IF r = 0 THEN signed_zero(op, fin1, fin2, mode)
      ELSE real_to_fp(r)
      ENDIF
```

Function fp_op does the appropriate conversions and calls function apply to perform the appropriate arithmetic operation. If the rounded result is zero, function signed_zero (Section 4.1.4) is invoked to return a correctly signed zero. Otherwise, real_to_fp converts the result to a floating-point number. Function apply uses the dependent type and predicate subtype mechanisms of PVS to restrict the domain of its third argument to nonzero numbers when the operation is div. Without this type restriction, it would not be possible to justify the use of '/' in the definition of apply.

4.1.2 Arithmetic on Infinities

The standard defines well behaved operations involving infinite arguments. It states:

> *Infinity arithmetic shall be construed as the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is, $-\infty < $ (every finite number) $< +\infty$.* [3, Section 6.1]

This requires special treatment for each arithmetic operator, the example given here is for floating-point addition.

```
fp_add_inf(num1, (num2: num | infinite?(num1) OR infinite?(num))): fp_num
    =
  IF infinite?(num1) & infinite?(num2) THEN
    IF (i_sign(num1) = i_sign(num2)) THEN num1
    ELSE invalid
    ENDIF
  ELSIF infinite?(num1) THEN num1
  ELSE num2
  ENDIF
```

Function fp_add_inf takes two numeric arguments (i.e. either finite or infinite, but not NaN), one of which must be an infinity. If only one argument is an infinity, that argument is the return value. If both are infinite and have the same sign, then either argument is the correct return value. However, if the two infinite arguments have different signs, the invalid exception must be signaled. In the definition here, fp_add_inf returns a NaN value invalid. This will serve as a placeholder until the PVS specification is revised to properly deal with exceptions. Infinity arithmetic for the other operators is defined similarly.

4.1.3 Arithmetic on NaNs

The standard does not specify interpretation of NaNs, however, it does constrain the behavior of operations given NaN arguments. The main motivation for these constraints is to enable use of NaNs either for diagnostic information, or for implementation dependent enhancements to the operations.

Section 6.2 of the standard states:

> Every operation involving a signaling NaN or invalid operation (7.1) shall, if no trap occurs and if a floating-point result is to be delivered, deliver a quiet NaN as its result.

> Every operation involving one or two input NaNs, none of them signaling, shall signal no exception, but, if a floating-point result is to be delivered, shall deliver as its result a quiet NaN, which should be one of the input NaNs. [3]

The following PVS specification captures the various cases for dealing with NaN arguments. Function fp_quiet is constrained via the PVS dependent type mechanism to return one of its arguments. Function fp_signal tests to see if the invalid trap is enabled; if not, a quiet NaN is returned.

```
fp_quiet(op,fp1,(fp2| NaN?(fp1) OR NaN?(fp2))): {nan| nan=fp1 or nan=fp2}

fp_signal(op, fp1,(fp2| NaN?(fp1) OR NaN?(fp2))): fp_num =
  IF trap_enabled?(invalid_operation) THEN invalid
  ELSE fp_quiet(op, mk_quiet(fp1), mk_quiet(fp2))
  ENDIF

fp_nan(op, fp1, (fp2| NaN?(fp1) OR NaN?(fp2))): fp_num =
  IF signal?(fp1) OR signal?(fp2) THEN fp_signal(op, fp1, fp2)
  ELSE fp_quiet(op, fp1, fp2)
  ENDIF
```

4.1.4 The Algebraic Sign

The standard specifies a set of rules for the algebraic sign of an arithmetic result. There are two scenarios where special care is required to get the algebraic sign correct: arithmetic involving infinities (including division by zero), and arithmetic operations that deliver a result of zero. The cases involving infinities have been addressed above. When an arithmetic function evaluates to zero, we need to determine whether to return $+0$ or -0. For multiplication and division, the sign is "+" if and only if both arguments have the same sign. For addition and subtraction, the standard states:

> 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 $-\infty$, in which mode that sign shall be "$-$." However, $x + x = x - (-x)$ retains the same sign as x even when x is zero. [3, page 13]

The above definition of fp_op invokes signed_zero for all zero results. This function does the case analysis required to return a correctly signed floating-point zero.

```
signed_zero(op, fin1, fin2, mode): {fin | zero?(fin)} =
  CASES op OF
```

```
add:
    IF zero?(fin1)
            & zero?(fin2) & sign(fin1) = sign(fin2) THEN fin1
    ELSIF to_neg?(mode) THEN neg_zero
    ELSE pos_zero
    ENDIF,
sub:
    IF zero?(fin1)
            & zero?(fin2) & sign(fin1) /= sign(fin2) THEN fin1
    ELSIF to_neg?(mode) THEN neg_zero
    ELSE pos_zero
    ENDIF,
mult:
    IF sign(fin1) = sign(fin2) THEN pos_zero
    ELSE neg_zero
    ENDIF,
div:
    IF sign(fin1) = sign(fin2) THEN pos_zero
    ELSE neg_zero
    ENDIF
ENDCASES
```

4.2 Remainder

Section 5.1 of the standard defines the remainder function as follows:

> When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by
> the mathematical relation $r = x - y \cdot n$, where n is the integer nearest the exact value
> of x/y; whenever $|n - x/y| = 1/2$, then n is even. ...If $r = 0$, its sign shall be that of
> x. [3]

The function round_to_even, defined in section 3.1, gives us the necessary means to compute n
from the above description. The definition of the floating-point remainder function, fp_rem, is
straightforward in PVS.

```
REM(fin1, (fin2:fin|not zero?(fin))): fp_num =
  let x = value(fin1),
      y = value(fin2) in
  if (x - y * round_to_even(x/y)) = 0
     then finite(sign(fin1),E_min,d_zero)
     else real_to_fp(x - y * round_to_even(x/y))
  endif

fp_rem(fp1, fp2): fp_num =
  IF finite?(fp1) & finite?(fp2)
    THEN IF zero?(fp2)
      THEN invalid
    ELSIF zero?(REM(fp1, fp2)) THEN finite(sign(fp1),E_min,d_zero)
    ELSE REM(fp1, fp2)
```

```
      ENDIF
    ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan_rem( fp1, fp2)
    ELSIF infinite?(fp1) THEN invalid
    ELSE fp1
    ENDIF
```

According to the standard, the remainder function is always exact (i.e. no rounding error). This fact has not yet been proven in PVS.

4.3 Square Root

PVS does not have a built-in definition for the square root function. It can be defined by the expression:

```
  sqrt(px): {py | py * py = px}
```

This definition generates a TCC to prove that the range is nonempty for all positive reals px. It carries in its type signature the relevant information about the square root function.

The specification of the floating-point square root operation is:

```
  fp_sqrt(fp, mode): fp_num =
    IF NaN?(fp) THEN NaN_sqrt(fp)
    ELSIF zero?(fp) THEN fp
    ELSIF finite?(fp)
      THEN IF sign(fp) = pos
        THEN real_to_fp(fp_round(sqrt(value(fp)), mode))
      ELSE invalid
      ENDIF
    ELSIF i_sign(fp) = pos THEN fp
    ELSE invalid
    ENDIF
```

4.4 Conversion between precisions

The standard does not require any combination of precisions. However, if more than one precision is supported, the standard requires conversions between all supported precisions. A specification involving multiple precisions can easily be defined by importing multiple instances of theory IEEE_854. The basic operations for defining conversions between precisions are included in the PVS specification.

4.5 Floating-point ↔ decimal string

The PVS specification does not yet define conversions between floating-point numbers and decimal strings. The standard places no restrictions on the decimal string format, so this cannot be addressed fully until an implementation is defined.

4.6 Comparisons

The standard defines the comparison operations as follows:

Four mutually exclusive relations are possible: "less than," "equal," "greater than," and "unordered." The last case arises only when at least one operand is a NaN. ...

The result of a comparison shall be delivered in one of two ways at the implementor's option: either as a condition code identifying one of the four relations listed above, or as a true/false response to a predicate that names the specific predicate desired. [3, Section 5.7, page 12]

The first option is simple to define in PVS. We simply extend the valuation function to provide a value for the infinities, and then define the comparison function using the corresponding real relations.

```
comparison_code: type = {gt, lt, eq, un}

fp_compare((fp1, fp2: fp_num)): comparison_code =
  IF NaN?(fp1) OR NaN?(fp2) THEN un
  ELSIF n_value(fp1) > n_value(fp2) THEN gt
  ELSIF n_value(fp1) < n_value(fp2) THEN lt
  ELSE eq
  ENDIF
```

For each element of an enumerated type, PVS automatically generates a predicate recognizer. Thus, we can also use the above definition to support our formal specification of the second alternative for realizing floating-point comparisons. The following are the predicate forms that the standard requires.

```
% shall include
  eq?(fp1,fp2) :bool = eq?(fp_compare(fp1,fp2))
  ne?(fp1,fp2) :bool = not eq?(fp_compare(fp1,fp2))
  gt?(fp1,fp2) :bool = gt?(fp_compare(fp1,fp2))
  ge?(fp1,fp2) :bool = gt?(fp_compare(fp1,fp2)) or eq?(fp_compare(fp1,fp2))
  lt?(fp1,fp2) :bool = lt?(fp_compare(fp1,fp2))
  le?(fp1,fp2) :bool = lt?(fp_compare(fp1,fp2)) or eq?(fp_compare(fp1,fp2))

% should include
  un?(fp1,fp2) :bool = un?(fp_compare(fp1,fp2))
```

All that remains is to correctly merge exception handling with the above definitions.

5   Exceptions

Section 7 of the standard states:

There are five types of exceptions that shall be signalled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be associated a trap under user control, as specified in Section 8. ... In some cases the result is different if a trap is enabled.

For each type of exception, the implementation shall provide a status flag that shall be set on any occurrance of the corresponding exception when no corresponding trap occurs. ...

> *The only exceptions that can coincide are inexact with overflow and inexact with underflow.* [3, page 13]

The combination of exceptions and traps suggests that we need to modify the style of the PVS specification from a purely functional specification to a process based specification. The standard requires that we signal exceptions only if the trap for that exception is taken. However, in the current functional style we cannot model transfer of control to a trap handler.

This can be overcome by having each of the previously defined functions return a pair of values, the second element of the pair is either an indication of the exception to be signaled, or an identifier to determine which trap handler to invoke.

The potential values for this identifier are determined by the following datatype declaration:

```
exception: DATATYPE
  BEGIN
    invalid_operation       : invalid?
    division_by_zero        : div_by_zero?
    overflow                : overflow?
    underflow(exact: bool)  : underflow?
    inexact                 : inexact?
    no_exceptions           : no_exceptions?
  END exception

trap_enabled?(e:exception):bool % = ?
```

To incorporate this strategy, the types of the previously defined functions need to be modified slightly. We will illustrate the changes by working through a single example, fp_add. Each operation shall now return a pair. The first element will be an fp_num and the second will be the exception status.

```
fp_add_x(fp1, fp2, mode): [fp_num, exception] =
  IF finite?(fp1) & finite?(fp2) THEN fp_op_x(add, fp1, fp2, mode)
  ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan_x(add, fp1, fp2)
  ELSE fp_add_inf_x(fp1, fp2)
  ENDIF
```

The modifications required for infinity arithmetic are trivial. Section 6.1 of the standard states:

> *Arithmetic on ∞ is always exact and therefore shall signal no exceptions, except for the invalid operations specified for ∞ in Section 7.1.* [3, page 13]

The definition for fp_add_inf_x is:

```
fp_add_inf_x(num1,
            (num2: num | infinite?(num1)
                          OR infinite?(num)))
 : [fp_num, exception]
   =
 IF infinite?(num1) & infinite?(num2) THEN
   IF (i_sign(num1) = i_sign(num2)) THEN
       (num1, no_exceptions)
```

17

```
      ELSE (invalid, invalid_operation)
      ENDIF
   ELSIF infinite?(num1) THEN
         (num1, no_exceptions)
   ELSE  (num2, no_exceptions)
   ENDIF
```

The operations on NaNs require similar modifications. The difficult cases in handling exceptions occur with the rounding operations involved in fp_op_x.

```
fp_op_x(op,fin1,(fin2:fin| div?(op) => not zero?(fin)),mode):
  [fp_num, exception]
         =
  LET rp = fp_round_x(apply(op, fin1, fin2), mode)
    IN IF proj_1(rp) = 0 THEN (signed_zero(op, fin1, fin2, mode),proj_2(rp))
      ELSE real_to_fp_x(rp)
      ENDIF

real_to_fp_x(r,e) : [fp_num, exception] = (real_to_fp(r),e)
```

Function fp_round_x is defined by:

```
fp_round_x(r, mode): [real,exception] =
  IF r = 0 THEN (0,no_exceptions)
  ELSIF over_under?(r) then
    round_exceptions_x(r,mode)
  ELSE (round_scaled(r,mode),is_exact?(r,mode))
  ENDIF

is_exact?(r:nzreal,mode):exception =
  IF round_scaled(r,mode) = r then no_exceptions ELSE inexact ENDIF
```

All that remains is to define round_exceptions_x. This is a rather complicated definition that breaks down into two cases. The first case is a potential overflow; the second is a potential underflow.

```
x: var (over_under?)

round_exceptions_x(x,mode): [fp_num, exception] =
  IF abs(r)>max_pos THEN
          overflow(x,mode)
     ELSE underflow(x,mode)
     ENDIF
```

Full descriptions of functions overflow and underflow will appear in the corresponding section below.

5.1   Invalid Operation

The invalid operation exception does not require that a special value be delivered to a trap handler. The cases where the invalid operation exception is raised for arithmetic operations have been handled above.

5.2 Division by Zero

The standard does not require any special treatment when the trap is enabled, but it requires that an appropriately signed infinity be delivered when the exception is raised. The modified fp_div is:

```
fp_div_x(fp1, fp2, mode): [fp_num,exception] =
  IF finite?(fp1) & finite?(fp2)
    THEN IF zero?(fp2)
      THEN IF zero?(fp1)
        THEN (invalid, invalid_operation)
      ELSE (infinite(mult_sign(fp1, fp2)), division_by_zero)
      ENDIF
    ELSE fp_op_x(div, fp1, fp2, mode)
    ENDIF
  ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan_x(div, fp1, fp2)
  ELSE fp_div_inf_x(fp1, fp2)
  ENDIF
```

5.3 Overflow

The result of an overflow is determined by both the rounding mode and overflow trap status.

> *The overflow exception shall be signaled whenever the destination precision's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded.*
>
> . . .
>
> *Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by b^α and then rounding.* [3, page 14, section 7.3]

The standard continues by relating the possible values of α to the exponent range. The given relation relies on the assumption that the exponent range is balanced around zero.

The overflow threshold is different for each of the rounding modes. The rounding mode also determines the result when an overflow occurs. The PVS definition is

```
trap_over((r1:nzreal), (r2: real), (mode: rounding_mode)): real =
  IF trap_enabled?(overflow) THEN round_scaled(r1 * b ^ (-alpha), mode)
  ELSE r2
  ENDIF

overflow((r: nzreal | abs(r) > max_pos), (mode: rounding_mode)):
    [real, exception] =
  CASES mode OF
    to_nearest:
        IF abs(r)
              >= b ^ (E_max + 1)
                  - (1 / 2) * b ^ (E_max + 1 - p)
          THEN (trap_over(r, (sgn(r) * infinity), mode),overflow)
        ELSE (sgn(r) * max_pos, inexact)
```

```
        ENDIF,
    to_zero:
        IF abs(r) >= b ^ (E_max + 1)
          THEN (trap_over(r, (sgn(r) * max_pos), mode),overflow)
        ELSE (sgn(r) * max_pos, inexact)
        ENDIF,
    to_pos:
        IF r > max_pos THEN (trap_over(r, infinity, mode),overflow)
        ELSIF r <= -b ^ (E_max + 1)
          THEN (trap_over(r, max_neg, mode),overflow)
        ELSE (max_neg, inexact)
        ENDIF,
    to_neg:
        IF r < max_neg THEN (trap_over(r, -infinity, mode),overflow)
        ELSIF r >= b ^ (E_max + 1)
          THEN (trap_over(r, max_pos, mode),overflow)
        ELSE (max_pos, inexact)
        ENDIF
    ENDCASES
```

5.4 Underflow

For results less than $b^{E_{min}}$, we may not be able to preserve p significant digits. The PVS specification includes a special rounding function for these cases of potential underflow.

```
round_under((r: nzreal | abs(r) < b ^ E_min), (mode: rounding_mode)): real
  = b^(E_min - (p - 1))*round(b^(-(E_min - (p - 1)))*r,mode)
```

The following correctness results have been proven in PVS about round_under:

```
round_under_error: LEMMA
      abs(r) < b ^ E_min
        => abs(r - round_under(r, mode)) < b ^ (E_min - (p - 1))


round_under_near: LEMMA
      abs(r) < b ^ E_min
        => abs(r - round_under(r, to_nearest))
               <= b ^ (E_min - (p - 1)) / 2
```

In addition, round_under rounds in the correct direction for the directed rounding modes. This function is the core of the definition of function underflow.

In the absence of traps, underflow is signaled when a result is both tiny and inaccurate. Each of these conditions may be defined in two distinct ways. Tininess occurs when a result is less than $b^{E_{min}}$; it may be detected either before or after rounding. The PVS specification uses the following predicate to signal tininess:

```
tiny?((r: nzreal | abs(r) < b ^ E_min), (mode: rounding_mode)): bool =
  IF tiny_flag THEN abs(round_scaled(r, mode)) < b ^ E_min ELSE TRUE ENDIF
```

Boolean constant `tiny_flag` is used to signify which method a particular implementation is using to signal tininess; if `tiny_flag = true`, then tininess is detected after rounding, if `tiny_flag = false` we have already satisfied the *before rounding* test for tininess (via the constraints on argument r).

Similarly, loss of accuracy may also be detected in one of two ways, either a loss due to denormalization or an inexact result. The PVS predicate detecting loss of accuracy is given by:

```
inaccurate?((r: nzreal | abs(r) < b ^ E_min), (mode: rounding_mode)): bool =
  IF inaccurate_flag THEN (round_scaled(r,mode)/=round_under(r,mode))
  ELSE (r/=round_under(r,mode))
  ENDIF
```

If the underflow trap is enabled, it is taken whenever tininess is detected. On a trapped underflow, the result must be scaled by α. Otherwise the both tininess and loss of accuracy must occur for underflow to be signaled. These cases are captured in the PVS definition of `underflow`:

```
underflow((r: nzreal | abs(r) < b ^ E_min), (mode: rounding_mode)):
    [real, exception] =
  IF tiny?(r, mode)
    THEN IF trap_enabled?(underflow(FALSE))
      THEN (round_scaled(r * b ^ alpha, mode),
            underflow(exact_underflow(r, mode)))
    ELSIF inaccurate?(r, mode) THEN (round_under(r, mode), underflow(TRUE))
    ELSE (round_under(r, mode),
          IF r = round_under(r, mode) THEN no_exceptions
          ELSE inexact
          ENDIF)
    ENDIF
  ELSE (round_under(r, mode), inexact)
  ENDIF
```

This completes the definition of rounding in the presence of exceptions.

5.5 Inexact

The delivered result of a function does not change when the inexact exception is signalled. The signal is raised whenever the value of the delivered result is different from the infinitely precise intermediate result (i.e. inexact is signalled when rounding occurs). This can be computed with respect to function `round` by using function `is_exact?`.

Inexact may also be signaled in conjunction with overflow or underflow. These cases were addressed above.

6 Traps

The PVS specification does not address traps other than the declaration of the predicate `trap_enabled?` which is used to test whether a particular trap is enabled.

7 Concluding Remarks

This document described a partial definition of the IEEE-854 Standard for Radix-Independent Floating-Point Arithmetic in the PVS verification system. In most instances, there was a straightforward definition of the IEEE-854 features using the PVS specification language. Formal techniques are sufficiently mature that it is reasonable to consider use of formal specification techniques in the development of future standards.

The constraints enumerated in IEEE-854 for floating-point arithmetic are a generalization the IEEE-754 Standard for Binary Floating-Point Arithmetic. Therefore, this formalization of the IEEE-854 standard can be instantiated to serve as a basis for the formal specification of IEEE-754 arithmetic. All that is required is to instantiate the general theory with the appropriate constants, and define the representation formats in accordance with IEEE-754.

The PVS theories described in this document provide a core formal basis for verifying any proposed instance of IEEE floating-point arithmetic. We plan to explore the verification of floating-point systems with respect to the formal description presented here.

Acknowledgements

References

[1] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[2] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754-1985.

[3] IEEE. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987. ANSI/IEEE Std 854-1987.

[4] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[5] N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.