Interpretation of IEEE-854 Floating-Point Standard and Definition in the HOL system

Victor A. Carreño
NASA Langley Research Center
Hampton, VA 23681-0001
v.a.carreno@larc.nasa.gov

The ANSI/IEEE Standard 854-1987 for floating-point arithmetic is interpreted by converting the lexical descriptions in the standard into mathematical conditional descriptions organized in tables. The standard is represented in higher-order logic within the framework of the HOL system. The paper is divided in two parts with the first part the interpretation and the second part the description in HOL.

The objective of this work is to provide a representation of the IEEE-854[6] standard in a formal logic system against which implementations can be verified using deductive reasoning. Before the standard could be represented in the formal system it was necessary to extract the meaning of the standard for numerous conditions and cases. Hence, the interpretation of the standard became part of the effort. Paul Miner provided valuable discussions to aid in the standard interpretation and worked in a similar effort to specify the IEEE-854 standard in the PVS system[7].

Previous efforts to represent a floating point arithmetic standard in a formal language include the partial formalization of the ANSI/IEEE-754-1985[5] standard in the Z language by Geoff Barret[1] and in the HOL system by Jing Pang[8]. IEEE-854 is a generalization of IEEE-754. IEEE-854 does not specify encoding formats for floating-point numbers and permits the representation of floating-point numbers in the binary and decimal systems.

The interpretation of the standard is not intended to replace the standard but rather aid in its understanding. Although the standard has been reviewed meticulously to get a full understanding of its meaning, errors probably exist in the interpretation. Any discrepancies between the interpretation and the standard should be considered an interpretation error and the standard should take precedence.

Part 1: Interpretation

1 Introduction

This part of the paper covers the interpretation of ANSI/IEEE Standard 854-1987. The interpretation consists of the definition of 29 tables which address the cases found during floating-point rounding and arithmetic operations. Operations on infinity, zero, and symbolic entries, as well as exceptions and traps are incorporated directly in the definition of each operation rather than in separate sections.

2 Floating-point Numbers and Precisions

This section contains a brief definition of floating-point numbers and floating-point precisions. A floating-point number is a digit string characterized by three components: a sign digit, a signed exponent, and a significand. A floating-point number can have three meanings: 1. a value; 2. an infinite; and 3. not a number (NaN). Values, infinities, and NaNs are further divided into classes. A value could be a normal number, subnormal number, or zero. An infinite could be positive or negative. NaNs could be signaling or quiet. A subnormal number is a nonzero valued number whose magnitude is less than the base raised to the precision's minimum exponent. IEEE-854 defines four precision: single, double, single extended, and double extended. Each precision is defined by the following parameters:

b = the radix or base

p = the number of base-b digits in the significand

E_{max} = the maximum exponent

E_{min} = the minimum exponent

For all precisions, the parameters are subjected to the following constraints:

b shall be either 2 or 10 and shall be the same for all supported precisions

$(E_{max} - E_{min}) / p$ shall exceed 5 and should exceed 10

$b^{p-1} \geq 10^5$

Additional constraints are imposed on the parameters for double, single extended, and double extended precisions. For double precision:

$$b^{p_d} \geq 10b^{2p_s}$$

$$E_{max_d} \geq 8E_{max_s} + 7$$

$$E_{min_d} \leq 8E_{min_s}$$

where the subscripts d and s denote double and single precisions respectively. For extended precision, the following constraints must hold over the base precision:

$$E_{max_e} \geq 8E_{max_b} + 7$$

$$E_{min_e} \leq 8E_{min_b}$$

$$p_e \geq 1.2p_b$$

for $b = 2$ $\quad p_e \geq p_b + \lceil \log_2 (E_{max_b} - E_{min_b}) \rceil$

where the subscript e and b denote the extended and base precisions.

Thus, each precision allows the representation of just the following entities:

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

   s = a natural number defining the algebraic sign

   E = any integer between E_{min} and E_{max}, inclusive

   d_i = a base-b digit $0 \leq d_i \leq (b-1)$

2. Two infinities, $+\infty$ and $-\infty$
3. At least one signaling NaN
4. At least one quiet NaN

3 Exceptions and Traps

Operations on floating-point numbers, defined in succeeding sections, can signal exceptions as a result of performing the operation. The generation of exceptions will depend on operands, results, and operation conditions. Five exceptions are signaled when detected:

Invalid operation

Division by zero

Overflow

Underflow

Inexact

An exception will set a status flag and, if enabled by the user, will invoke an exception handling trap. If exception handlers are implemented then each exception should have a user controlled trap associated with it.

The resulting value on some operations will depend on whether an exception is detected and a trap is enabled. Conditions which will result in exceptions will be defined within the operation's definition.

4 Rounding

Floating-point numbers are intended to be a finite approximation of the real numbers. Rounding is defined in the IEEE-854 standard thus,

> *Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision while signaling the inexact exception (see 7.5).* [6, section 4, page 9]

Four rounding modes are specified in the standard:

> *An implementation of this standard shall provide round to nearest as the default rounding mode.[...] An implementation of this standard shall also provide three user-selectable directed rounding modes: round towards* $+\infty$*, round towards* $-\infty$*, and round towards 0.* [6, section 4.1, page 9]

In addition, depending on the magnitude of the number to be rounded and the rounding mode, rounding can produce infinite floating-points while signaling other exceptions:

> *The rounding modes may affect the signs of zero sums (see 6.3), and do affect the threshold beyond which overflow (see 7.3) and underflow (see 7.4) may be signaled.* [6, section 4, page 9]

Round to near returns the floating-point number with value nearest to the infinitely precise number. If two floating-point number values are equally near, round to near returns the one with least significant digit even. Round to positive infinity returns a floating-point number with the smallest value which is greater than the infinitely precise number. Round to negative infinity returns a floating-point number with the greatest value less than the infinitely precise number. Round to zero returns a floating-point number with the largest magnitude which is less than the infinitely precise number.

The following tables summarize the interpretation of the standard for all value ranges of real numbers to be rounded. G_{neg}, G_{pos}, L_{neg}, and L_{pos} represent, respectively, the greatest negative, greatest positive, least negative, and least positive finite floating point number representable in a given precision. The tables give the result and exceptions, if any, of the rounding operation for a given infinite precision number and a rounding mode. Three possible exceptions can be signaled by the rounding operation: underflow, overflow, and inexact.

Overflow detection when the overflow trap handler is implemented and enabled will deliver to the trap handler the infinitely precise result of the operation divided by b^{α} and then rounded. The exponent adjustment α is chosen to be approximately $3((E_{max} - E_{min})/4)$ and should be divisible by twelve.

Table 1: Negative numbers less than G_{neg}

mode	Overflow trap	$r \le -b^{E_{max}+1}$	$-b^{E_{max}+1} < r \le -b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right)$	$-b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right) < r < G_{neg}$
near	disabled	-inf, overflow, inexact	-inf, overflow, inexact	G_{neg}, inexact
	enabled	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact	G_{neg}, inexact
pos_inf	disabled	G_{neg}, overflow, inexact	G_{neg}, inexact	G_{neg}, inexact
	enabled	round(r/b^{α}), overflow, inexact	G_{neg}, inexact	G_{neg}, inexact
neg_inf	disabled	-inf, overflow, inexact	-inf, overflow, inexact	-inf, overflow, inexact
	enabled	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact
zero	disabled	G_{neg}, overflow, inexact	G_{neg}, inexact	G_{neg}, inexact
	enabled	round(r/b^{α}), overflow, inexact	G_{neg}, inexact	G_{neg}, inexact

When the value resulting from a rounding operation is not equal to the infinitely precise number the inexact exception is signaled. The inexact flag is represented by *exc1* and is defined in table 10.

Table 2: Negative numbers greater than or equal to G_{neg} and less than or equal to $-b^{E_{min}}$

mode	$G_{neg} \le r \le -b^{E_{min}}$
all modes	normal, exc1

Underflow detection when the underflow trap handler is implemented and enabled will deliver to the trap handler the infinitely precise result of the operation multiplied by b^{α} and then rounded. The exponent adjustment α is the same as used for overflow. *exc2* is the underflow exception flag defined in table 11. Underflow detection depends on the rounding result, detection scheme selected by the user, and/or traps enabled.

Table 3: Negative numbers greater than $-b^{E_{min}}$ and less than or equal to L_{neg}

mode	underflow trap	$-b^{E_{min}} < r \le -b^{E_{min}} - \frac{1}{2}L_{neg}$	$-b^{E_{min}} - \frac{1}{2}L_{neg} < r$ $r < -b^{E_{min}} - L_{neg}$	$-b^{E_{min}} - L_{neg} \le r \le L_{neg}$
near	disabled or (enabled, no underflow)	$-b^{E_{min}}$, inexact, exc2	$-b^{E_{min}} - L_{neg}$, inexact, exc2	denormal, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
pos_inf	disabled or (enabled, no underflow)	$-b^{E_{min}} - L_{neg}$, inexact, exc2	$-b^{E_{min}} - L_{neg}$, inexact, exc2	denormal, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 3: Negative numbers greater than $-b^{E_{min}}$ and less than or equal to L_{neg}

mode	underflow trap	$-b^{E_{min}} < r \le -b^{E_{min}} - \frac{1}{2}L_{neg}$	$-b^{E_{min}} - \frac{1}{2}L_{neg} < r$ $r < -b^{E_{min}} - L_{neg}$	$-b^{E_{min}} - L_{neg} \le r \le L_{neg}$
neg_inf	disabled or (enabled, no underflow)	$-b^{E_{min}}$, inexact, exc2	$-b^{E_{min}}$, inexact, exc2	denormal, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
zero	disabled or (enabled, no underflow)	$-b^{E_{min}} - L_{neg}$, inexact,exc2	$-b^{E_{min}} - L_{neg}$, inexact, exc2	denormal, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 4: Negative numbers greater than L_{neg}

mode	underflow trap	$L_{neg} < r < \frac{1}{2}L_{neg}$	$\frac{1}{2}L_{neg} \le r < 0$
near	disabled	L_{neg}, inexact, underflow	-0, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
pos_inf	disabled	-0, inexact, underflow	-0, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
neg_inf	disabled	L_{neg}, inexact, underflow	L_{neg}, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 4: Negative numbers greater than L_{neg}

mode	underflow trap	$L_{neg} < r < \frac{1}{2}L_{neg}$	$\frac{1}{2}L_{neg} \le r < 0$
zero	disabled	-0, inexact, underflow	-0, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 5: Zero

mode	$(-0) + (-0)$	$(+0) + (+0)$	*sign(fp1)* $\neq$ *sign(fp2)* *v(fp1)+v(fp2)*=0	$+0 \times fp$	$-0 \times fp$
near	-0	+0	+0	sign(fp)0	sign(-fp)0
pos_inf	-0	+0	+0	sign(fp)0	sign(-fp)0
neg_inf	-0	+0	-0	sign(fp)0	sign(-fp)0
zero	-0	+0	+0	sign(fp)0	sign(-fp)0

sign(fp) is the algebraic sign of the floating point number *fp*. *v(fp)* denotes the value of the finite floating point number *fp* and *v(fp1)+v(fp2)* is the infinitely precise addition of the values of *fp1* and *fp2*.

Table 6: Positive numbers less than L_{pos}

mode	underflow trap	$0 < r \le \frac{1}{2}L_{pos}$	$\frac{1}{2}L_{pos} < r < L_{pos}$
near	disabled	+0, inexact, underflow	L_{pos}, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
pos_inf	disabled	L_{pos}, inexact, underflow	L_{pos}, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
neg_inf	disabled	+0, inexact, underflow	+0, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 6: Positive numbers less than L_{pos}

mode	underflow trap	$0 < r \le \frac{1}{2}L_{pos}$	$\frac{1}{2}L_{pos} < r < L_{pos}$
zero	disabled	+0, inexact, underflow	+0, inexact, underflow
	enabled	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 7: Positive numbers greater than or equal to L_{pos} and less than $b^{E_{min}}$

mode	underflow trap	$L_{pos} \le r \le b^{E_{min}} - L_{pos}$	$b^{E_{min}} - L_{pos} < r < b^{E_{min}} - \frac{1}{2}L_{pos}$	$b^{E_{min}} - \frac{1}{2}L_{pos} \le r < b^{E_{min}}$
near	disabled or (enabled, no underflow)	denormal, exc1,exc2	$b^{E_{min}} - L_{pos}$, exc1, exc2	$b^{E_{min}}$, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
pos_inf	disabled or (enabled, no underflow)	denormal, exc1,exc2	$b^{E_{min}}$, exc1, exc2	$b^{E_{min}}$, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
neg_inf	disabled or (enabled, no underflow)	denormal, exc1,exc2	$b^{E_{min}} - L_{pos}$, exc1, exc2	$b^{E_{min}} - L_{pos}$, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow
zero	disabled or (enabled, no underflow)	denormal, exc1,exc2	$b^{E_{min}} - L_{pos}$, exc1, exc2	$b^{E_{min}} - L_{pos}$, exc1, exc2
	enabled and underflow detected	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow	round($r \times b^{\alpha}$), inexact, underflow

Table 8: Positive numbers greater than or equal to $b^{E_{min}}$ and less than or equal to G_{pos}

mode	$b^{E_{min}} \le r \le G_{pos}$
all modes	normal, exc1

Overflow detection will deliver to the trap handler the infinitely precise result divided by b^{α} and then rounded, when the overflow trap handler is implemented and enabled. The exponent adjustment α is as defined in page 4.

Table 9: Positive numbers greater than G_{pos}

mode	Overflow trap	$G_{pos} < r < b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right)$	$b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right) \le r < b^{E_{max}+1}$	$b^{E_{max}+1} \le r$
near	disabled	G_{pos}, inexact	+inf, overflow, inexact	+inf, overflow, inexact
	enabled	G_{pos}, inexact	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact
pos_inf	disabled	+inf, overflow, inexact	+inf, overflow, inexact	+inf, overflow, inexact
	enabled	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact	round(r/b^{α}), overflow, inexact
neg_inf	disabled	G_{pos}, inexact	G_{pos}, inexact	G_{pos}, overflow, inexact
	enabled	G_{pos}, inexact	G_{pos}, inexact	round(r/b^{α}), overflow, inexact

Table 9: Positive numbers greater than G_{pos}

mode	Overflow trap	$G_{pos} < r < b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right)$	$b^{E_{max}}\left(b - \frac{1}{2}b^{1-p}\right) \le r < b^{E_{max}+1}$	$b^{E_{max}+1} \le r$
zero	disabled	G_{pos}, inexact	G_{pos}, inexact	G_{pos}, overflow, inexact
	enabled	G_{pos}, inexact	G_{pos}, inexact	round(r/b^{α}), overflow, inexact

Exception *exc1* depends on whether or not the rounded result is equal to the infinitely precise number. *exc1* is *inexact* if *r* and *round r* are not equal, and no exception if they are equal.

Table 10: *exc1* inexact exception flag

mode	(round r) = r	(round r) $\neq r$
all	inexact = false	inexact = true

Underflow exception, *exc2*, depends on the detection of tininess, loss of accuracy, and whether the underflow trap is enabled or disabled. When the underflow trap is disabled, both tininess and loss of accuracy must be detected to signal underflow. When the underflow trap is enabled detection of tininess results in an underflow flag.

Table 11: *exc2* underflow exception flag

mode	underflow trap disabled	underflow trap enabled
all	tiny $\wedge$ loss_acc = underflow	tiny = underflow

Detection of tininess and loss of accuracy is user selectable. Tininess can be detected before or after rounding.

Table 12: Tininess detection before rounding

mode	$0 < \lvert r \rvert < b^{E_{min}}$	$b^{E_{min}} \le \lvert r \rvert$
all	true	false

Table 13: Tininess detection after rounding

mode	$-b^{E_{min}} < r \le -b^{E_{min}} - \frac{1}{2}L_{neg}$	$0 < \lvert r \rvert < b^{E_{min}} - \frac{1}{2}L_{pos}$	$b^{E_{min}} - \frac{1}{2}L_{pos} \le r < b^{E_{min}}$	$b^{E_{min}} \le \lvert r \rvert$
near	false	true	false	false
pos_inf	true	true	false	false
neg_inf	false	true	true	false
zero	true	true	true	false

Loss of accuracy can be detected by denormalization loss or by inexact. Detection of denormalization loss is defined in IEEE-854 as follows:

> *A denormalization loss: When the delivered results differs from what would have been computed were the exponent range unbound.* [6, section 7.4, page 15]

An unbound exponent range gives p digits of accuracy regardless of the number's magnitude. Consider for example the number

$$r = b^{E_{min}} \underbrace{0.00\ldots0}_{p}\underbrace{d_{p-1}00\ldots0d_{2p-1}}_{p}$$

where,

$$d_{p-1} = \alpha \ne 0 \text{ and } d_{2p-1} = \beta \ne 0$$

This number, when rounded to near with the exponent range unbound, will result in:

$$b^{E_{min} - (p-1)} d_0.00\ldots0d_{p-1}$$

where,

$$d_0 = \alpha \text{ and } d_{p-1} = \beta$$

Rounding to near with the exponent range bounded will give:

$$b^{E_{min}} 0.00\ldots00d_{p-1}$$

where,

$$d_{p-1} = \alpha$$

The loss of accuracy due to rounding with the exponent range bounded is $\lvert r - (\text{round } r) \rvert = \beta \times b^{E_{min} - (2p-1)}$

In general, when $|r - (\text{round } r)| > \frac{1}{2} b^{\lfloor \log_b r \rfloor + (1-p)}$ (for rounding to near) the delivered result with exponent bound will differ from the result with exponent unbound and loss of accuracy shall be detected. Other rounding modes have different detection thresholds as given in the next table.

Table 14: Loss of accuracy detection by denormalization loss

mode	$-b^{E_{min}} < r \le -b^{E_{min} + (1-p)}$	$0 < \|r\|$ $\|r\| < b^{E_{min} + (1-p)}$	$b^{E_{min} + (1-p)} \le r < b^{E_{min}}$	$b^{E_{min}} \le \|r\|$
near	$\|r - (\text{round } r)\| > \left(\frac{1}{2} b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	true	$\|r - (\text{round } r)\| > \left(\frac{1}{2} b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	false
pos_inf	$((\text{round } r) - r) \ge \left(b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	true	$((\text{round } r) - r) > 0$	false
neg_inf	$(r - (\text{round } r)) > 0$	true	$(r - (\text{round } r)) \ge \left(b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	false
zero	$((\text{round } r) - r) \ge \left(b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	true	$(r - (\text{round } r)) \ge \left(b^{\lfloor \log_b r \rfloor + (1-p)}\right)$	false

Table 15: Loss of accuracy detection by inexact

mode	$\forall r$
all	$(\text{round } r) \ne r$

5 Operations

Implementations conforming to the IEEE-854 standard must provide the add, subtract, multiply, divide, square root, remainder, round to floating point integer, conversion between precisions, conversion between floating point and integer numbers, conversions between floating point numbers and decimal strings, and compare operations. The arithmetic operations are shown in tabular form for all floating-point arguments.

5.1 Arithmetic

Table 16: Floating-point addition

fp1 add fp2		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
fp2	sig NaN	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
	quiet NaN	quiet NaN, invalid	fp1 $\vee$ fp2	fp2	fp2	fp2	fp2	fp2
	-0	quiet NaN, invalid	fp1	-0	round(0)	fp1	-inf	+inf
	+0	quiet NaN, invalid	fp1	round(0)	+0	fp1	-inf	+inf
	finite $\neq 0$	quiet NaN, invalid	fp1	fp2	fp2	round (v(fp1)+ v(fp2))	-inf	+inf
	-inf	quiet NaN, invalid	fp1	-inf	-inf	-inf	-inf	quiet NaN, invalid
	+inf	quiet NaN, invalid	fp1	+inf	+inf	+inf	quiet NaN, invalid	+inf

v(fp) denotes the value of the finite floating point number *fp* and *v(fp1)+v(fp2)* is the infinitely precise addition of the values of *fp1* and *fp2*.

Floating-point subtraction is defined in terms of floating-point addition. The unary negation operation “-” will change the algebraic sign of a floating-point number by changing its sign digit. The negation operation will change the sign of finites and infinities and will leave NaNs unchanged.

Table 17: Floating-point subtraction

fp1 sub fp2	fp1
fp2	fp1 add (-fp2)

Table 18: Floating-point multiplication

fp1 mul fp2		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
fp2	sig NaN	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
	quiet NaN	quiet NaN, invalid	fp1∨fp2	fp2	fp2	fp2	fp2	fp2
	-0	quiet NaN, invalid	fp1	+0	-0	sign(-fp1) 0	quiet NaN, invalid	quiet NaN, invalid
	+0	quiet NaN, invalid	fp1	-0	+0	sign(fp1)0	quiet NaN, invalid	quiet NaN, invalid
	finite $\neq 0$	quiet NaN, invalid	fp1	sign(-fp2) 0	sign(fp2)0	round(v(fp1) x v(fp2))	sign(-fp2) inf	sign(fp2) inf
	-inf	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	sign(-fp1) inf	+inf	-inf
	+inf	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	sign(fp1) inf	-inf	+inf

Table 19: Floating-point division

fp1 div fp2		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
fp2	sig NaN	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
	quiet NaN	quiet NaN, invalid	fp1$\vee$fp2	fp2	fp2	fp2	fp2	fp2
	-0	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	sign(-fp1) inf, div_zero	+inf, div_zero	-inf, div_zero
	+0	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	sign(fp1) inf, div_zero	-inf, div_zero	+inf, div_zero
	finite $\neq 0$	quiet NaN, invalid	fp1	sign(-fp2) 0	sign(fp2)0	round(v(fp1) $\div$ v(fp2))	sign(-fp2) inf	sign(fp2) inf
	-inf	quiet NaN, invalid	fp1	+0	-0	sign(-fp1) 0	quiet NaN, invalid	quiet NaN, invalid
	+inf	quiet NaN, invalid	fp1	-0	+0	sign(fp1)0	quiet NaN, invalid	quiet NaN, invalid

The reminder operation *x REM y* is defined by $x - (y \times n)$ for non-zero values of *y,* where *n* is the integer nearest to *x/y.*

Table 20: Floating-point remainder

fp1 REM fp2		fp1: sig NaN	fp1: quiet NaN	fp1: -0	fp1: +0	fp1: finite $\neq 0$	fp1: -inf	fp1: +inf
fp2	sig NaN	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
fp2	quiet NaN	quiet NaN, invalid	fp1\/fp2	fp2	fp2	fp2	fp2	fp2
fp2	-0	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
fp2	+0	quiet NaN, invalid	fp1	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid	quiet NaN, invalid
fp2	finite $\neq 0$	quiet NaN, invalid	fp1	-0	+0	v(fp1) - (v(fp2) x n) n = nearest integer (v(fp1)/v(fp2))	quiet NaN, invalid	quiet NaN, invalid
fp2	-inf	quiet NaN, invalid	fp1	fp1	fp1	fp1	quiet NaN, invalid	quiet NaN, invalid
fp2	+inf	quiet NaN, invalid	fp1	fp1	fp1	fp1	quiet NaN, invalid	quiet NaN, invalid

When two possible values of *n* are equally near to *x/y* then *n* is even. If $x - (y \times n)$ is zero then *x REM y* is +0 for positive *x* and -0 for negative *x* regardless of the rounding mode. Infinite arithmetic is defined in IEEE-854 as the limiting case of real arithmetic. To define the remainder function when *x* is finite and *y* is infinite, the limit is calculated as $\lim_{y \to \infty} (x - (y \times n)) = x$.

5.2 Square Root

The square root operation is defined for all non negative floating-point numbers. The square root of -0 shall be -0.

Table 21: Square root

SQR fp1	fp1							
	sig NaN	quiet NaN	-0	+0	finite < 0	finite > 0	-inf	+inf
	quiet NaN, invalid	fp1	-0	+0	quiet NaN, invalid	round $\sqrt{v(fp1)}$	quiet NaN, invalid	+inf

5.3 Floating-point Precision Conversion

Conversion between floating-point numbers of all precisions shall be possible. When converting from a lower to a higher precision the result will be exact. Conversion from a higher to lower precision may signal inexact.

Table 22: Floating-point precision conversions

fp1 to fp2		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
fp2 precision	narrower precision	quiet NaN, invalid	fp1	-0	+0	round v(fp1)	-inf	+inf
	wider precision	quiet NaN, invalid	fp1	fp1	fp1	fp1	fp1	fp1

5.4 Conversion Between Floating-point and Integer

Standard IEEE-854 specifies that compliant implementations must provide conversion between floating-point and integer number encodings. However, integer encoding is not in the scope of IEEE-854. When conversion from a floating-point to an integer number precludes a faithful representation, an invalid exception shall be raised. An exception may arise due to conversion of NaNs, infinities, or on overflow when the floating point value exceeds the maximum value of the integer encoding. When a representation does not exists for NaNs or infinities in the integer encoding, or if overflow occurs, the conversion result represented in table 23 by *res_exc4* is *undefined, invalid*.

exc3 exception flag is *inexact* if the value after conversion is not equal to the floating-point value before conversion. Conversion from an integer to a floating-point number should always be exact. If integer encodings of -0 and +0 is possible then conversion between floating-point and integers shall preserve the zero sign. When no such encoding exists for integer numbers, conversion of zero should result in +0.

Table 23: Floating-point to integer conversion

fp1 to Integer		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
rounding mode	near	quiet NaN, invalid or res_exc4	fp1 or res_exc4	-0 or 0	+0 or 0	(if $\lfloor v(fp1) \rfloor$ is even then $\left\lfloor \frac{\lceil 2 \times v(fp1) \rceil}{2} \right\rfloor$,exc3 else $\left\lceil \frac{\lfloor 2 \times v(fp1) \rfloor}{2} \right\rceil$,exc3) or res_exc4	-inf or res_exc4	+inf or res_exc4
	pos_inf	quiet NaN, invalid or res_exc4	fp1 or res_exc4	-0 or 0	+0 or 0	($\lceil v(fp1) \rceil$, exc3) or res_exc4	-inf or res_exc4	+inf or res_exc4
	neg_inf	quiet NaN, invalid or res_exc4	fp1 or res_exc4	-0 or 0	+0 or 0	($\lfloor v(fp1) \rfloor$, exc3) or res_exc4	-inf or res_exc4	+inf or res_exc4
	zero	quiet NaN, invalid or res_exc4	fp1 or res_exc4	-0 or 0	+0 or 0	(if fp1 > 0 then $\lfloor v(fp1) \rfloor$, exc3 else $\lceil v(fp1) \rceil$, exc3) or res_exc4	-inf or res_exc4	+inf or res_exc4

exc3 = IF integer(fp)= value(fp) THEN inexact=false ELSE inexact=true
res_exc4 = undefined, invalid

Table 24: Integer to floating point conversion

Integer to fp	Integer		
	-0	+0 or 0	finite $\neq 0$
fp	-0	+0	v(fp)=(value(integer))

5.5 Round Floating-point Number to Integral Value

Conversion from floating-point to integral valued floating-point rounds a floating-point number, according to the rounding mode, to a floating-point in the same precision with an integer value.

Table 25: Floating-point to integral valued floating-point

fp1 to fp2		fp1						
		sig NaN	quiet NaN	-0	+0	finite $\neq 0$	-inf	+inf
rounding mode	near	quiet NaN, invalid	fp1	-0	+0	v(fp2) = (if $\lfloor v(fp1) \rfloor$ is even then $\left\lfloor \frac{\lceil 2 \times v(fp1) \rceil}{2} \right\rfloor$ else $\left\lceil \frac{\lfloor 2 \times v(fp1) \rfloor}{2} \right\rceil$),exc3	-inf	+inf
	pos_inf	quiet NaN, invalid	fp1	-0	+0	v(fp2) = $\lceil v(fp1) \rceil$, exc3	-inf	+inf
	neg_inf	quiet NaN, invalid	fp1	-0	+0	v(fp2) = $\lfloor v(fp1) \rfloor$, exc3	-inf	+inf
	zero	quiet NaN, invalid	fp1	-0	+0	v(fp2) = (if v(fp1) > 0 then $\lfloor v(fp1) \rfloor$ else $\lceil v(fp1) \rceil$), exc3	-inf	+inf

exc3 = IF v(fp1) = v(fp2) THEN inexact = false ELSE inexact = true

5.6 Conversion Between Floating-point and Decimal String

Decimal strings are strings of characters representing decimal numbers. The format of decimal strings is not covered by IEEE-854. An uninterpreted function *format* is defined which takes the floating-point value *v(fp)* and converts it to a decimal string value represented by $\pm M \times 10^{\pm N}$. The function *value* extracts the value of a decimal string.

Table 26: Decimal string to floating-point conversion

DS to fp	DS							
	quiet NaN	signaling NaN or NaN	unrecog-nizable string	-0	+0 or 0	-inf or -infinity	+inf or +infinity or inf or infinity	value(DS)
fp	quiet NaN	signaling NaN	quiet NaN, invalid	-0	+0	-inf	+inf	round(value(DS))

Table 27: Floating-point to decimal string conversion

fp to DS	fp						
	quiet NaN	signaling NaN	-0	+0	-inf	+inf	finite $\neq 0$
DS	'NaN'	'NaN', invalid	value(DS) =0	value(DS) =0	'-inf' or '-infinity'	'inf' or 'infinity'	format(v(fp))

The function *format* must have the property $round(value(format(v(fp)))) = fp$ when rounding to nearest and conversions from floating-point numbers to decimal strings are performed such that *M* has *D* digits of precision where $D = \begin{cases} \lceil p \log_{10}(2) + 1 \rceil & b = 2 \\ p & b = 10 \end{cases}$

5.7 Comparison

For any two arbitrary floating-point numbers, one and only one of the following relations must hold: "less than", "equal", greater than", or "unordered". Comparison operations can be implemented in two possible ways: 1) A comparison will return as a result one of the four relations above; 2) A predicate defines a specific relation between two floating-point numbers and the comparison returns true if the predicate holds on the arguments and false otherwise, possibly together with an exception. Tables 28 and 29 describe the relation and predicate implementation options. The predicates in Table 29 are defined in terms of the relations of Table 28. Note that for Table 28, if *fp1* is a NaN and *fp1 = fp2* the result of a comparison is still unordered. That is, a NaN compares unordered with itself.

Table 28: Floating-point comparison: relations

fp1 comp fp2		fp1: sig NaN	fp1: quiet NaN	fp1: negative finite $\neq 0$	fp1: -0	fp1: +0	fp1: positive finite $\neq 0$	fp1: -inf	fp1: +inf
fp2	sig NaN	unordered, invalid	unordered, invalid	unordered, invalid	unordered, invalid	unordered, invalid	unordered, invalid	unordered, invalid	unordered, invalid
fp2	quiet NaN	unordered, invalid	unordered	unordered	unordered	unordered	unordered	unordered	unordered
fp2	negative finite $\neq 0$	unordered, invalid	unordered	less than or equal or greater than	greater than	greater than	greater than	less than	greater than
fp2	-0	unordered, invalid	unordered	less than	equal	equal	greater than	less than	greater than
fp2	+0	unordered, invalid	unordered	less than	equal	equal	greater than	less than	greater than
fp2	positive finite $\neq 0$	unordered, invalid	unordered	less than	less than	less than	less than or equal or greater than	less than	greater than
fp2	-inf	unordered, invalid	unordered	greater than	greater than	greater than	greater than	equal	greater than
fp2	+inf	unordered, invalid	unordered	less than	less than	less than	less than	less than	equal

Table 29: Floating-point comparison: predicates

fp1 predicate fp2			relation			
predicates			less than	equal	greater than	unordered
ASCII	Fortran	Math.				
=	.EQ.	$=$	false	true	false	false
?<>	.NE.	$\neq$	true	false	true	true
>	.GT.	$>$	false	false	true	false, invalid
>=	.GE.	$\geq$	false	true	true	false, invalid
<	.LT.	$<$	true	false	false	false, invalid
<=	.LE.	$\leq$	true	true	false	false, invalid
?	.UN.		false	false	false	true

Part 2: Definition in HOL

1 Introduction

This part of the paper presents the definition of the IEEE-854 standard in the HOL system[2].The standard is formalized using the higher-order logic language available in the system. The HOL system's logic is Church's simple theory of types with polymorphic and definitional extensions. The HOL system is a general purpose mechanized theorem prover. The system supports both forward and backward proofs. The forward proof style applies inference rules to existing theorems to obtain new theorems and eventually the desired theorem. Backward or goal oriented proofs start with the goal to be proven. Tactics are applied to the goal and subgoals until the goal is decomposed into simpler existing theorems or axioms.

By defining the IEEE-854 standard in the HOL system, it is possible to show that the standard meets given requirements. Desirable properties of the standard can be formulated in the logic and proofs can be constructed in the system to show that the formalization of the standard complies with stated properties.

The system basic language includes the natural numbers and boolean type. John Harrison's reals library[4] and Elsa Gunter's integer library[3] are used, respectively, for the definition of the real and integer types. The real and integer numbers are used as part of the IEEE-854 formalization. In the HOL system the symbol ? represents $\exists$, ! represents $\forall$, and @ is the choice or Hilbert operator. Entries in the HOL system are represented by the courier (type-writer) font.

2 Floating-point Numbers and Precisions

The four parameters defining a precision, *b*, *p*, *Emax*, and *Emin*, are defined in the HOL system by declaring *b* as a constant and placing constraints on the values of *p*, *Emax*, and *Emin*. *b* and *p* range over the natural numbers (type ":num") and *Emax* and *Emin* range over the integers (type ":integer"). The value of constant *b* is either 2 or 10.

```
new_definition(`b`,
"b = @n.(n=2)\/(n=10)");;
```

The formula "`@n.(n=2)\/(n=10)`" can be read "chose an n such that n=2 or n=10." The number of digits *p* is restricted in all precisions by the constraint $b^{p-1} \geq 10^5$ which is algebraically equivalent to

$$b = 2, \qquad p > 17$$

$$b = 10, \qquad p > 5$$

```
new_definition(`Sig`,
"Sig p = ((b = 2) ==> (17 < p))/\
         ((b =10) ==> (5 < p))");;
```

The constraint $(E_{max} - E_{min}) / p > 5$ is imposed on the values of *Emax*, *Emin* and *p* by the definition,

```
new_definition(`single`,
"single pr emax emin = (INT(5*pr) below (emax minus emin))");;
```
[1]

which must be true for single precision as well as all other precisions.

Additional constraints are imposed on the parameters for double, single extended, and double extended precisions. For double precision:

$$b^{p_d} \geq 10b^{2p_s}$$

$$E_{max_d} \geq 8E_{max_s} + 7$$

$$E_{min_d} \leq 8E_{min_s}$$

where the subscripts *d* and *s* denote double and single precision respectively, and is given by the definition,

```
new_definition(`double`,
"double ps pd emax_s emin_s emax_d emin_d =
(single pd emax_d emin_d)/\
(b = 2) ==> ((4 + (2*p_s)) <= p_d)/\
```

1. The natural, integer, and real numbers are different types in the HOL system and different operators and relations are defined on these types. The relations below, below_or_e, minus, plus, and times on the integers have the obvious meaning of less than, less than or equal, subtraction, addition, and multiplication, respectively. The operator INT takes a natural number and maps it into an integer number.

```
(b =10) ==> ((1 + (2*p_s)) <= p_d)/\
(((INT 8 times emax_s) plus INT 7) below_or_e emax_d)/\
((emin_d below_or_e (INT 8 times emin_s)))");;
```

For extended precision, the following constraints must hold over the base precision:

$$E_{max_e} \geq 8E_{max_b} + 7$$

$$E_{min_e} \leq 8E_{min_b}$$

$$p_e \geq 1.2p_b$$

$$\text{for } b = 2 \qquad p_e \geq p_b + \lceil \log_2 (E_{max_b} - E_{min_b}) \rceil$$

where the subscripts e and b denote the extended and base precision. These constraints are defined by,

```
new_definition(`extended`,
"extended p_b p_e emax_b emin_b emax_e emin_e =
(((INT 8 times emax_b) plus INT 7) below_or_e emax_e)/\
(emin_e below_or_e (INT 8 times emin_b))/\
(&p_e real_ge ((&1 real_add (&2/(&10))) real_mul (& p_b)))/\
((b=2) ==>
((p_b + ceiling(log 2 (& (FST (REP_integer(emax_b minus emin_b))))))
 <= p_e))");;
```
[2]

A floating-point number of any given precision must have an exponent value within the precision maximum and minimum exponent. The digits must be b-radix based.

```
new_definition(`precis`,
"precis emax emin fp =
(emin below_or_e (exponent fp))/\
((exponent fp) below_or_e emax)/\
(!n.(digits fp)n < b)");;
```

An implementation of the IEEE-854 will assign specific values to b, p, E_{max}, and E_{min}. These values must be shown to comply with the restrictions above. For example, for an implementation with single and double precision with values,

b=2, p_s = 24, Emax_s = 127, Emin_s = -126, p_d = 53, Emax_d = 1023, and Emin_d = -1022

we must show that,

```
"b=2 ==> (b=2) \/ (b=10)";;
"(b=2)/\(p_s=24) ==> (((b = 2) ==> (17 < p_s))/\
```

2. The ceiling function in this definition takes a real number as its argument and returns a natural number. The ceiling function of "`x:real`" is the least positive integer "`n:num`" greater than or equal to *x*. If x is negative, ceiling of *x* is zero. *log* 2 *x* is the logarithm base 2 of *x*. Arithmetic operators on the real numbers are prefixed by *real* as for example in *real_add* for the binary infixed addition operation. The operator "&" takes a natural number and maps it to the reals. The numbers &1, &2, ... are reals with values 1, 2, ...

```
                        ((b =10) ==> (5 < p_s)))";;
"single 24 (INT 127) (neg (INT 127))";;
"double 24 53 (INT 127) (neg (INT 127)) (INT 1023)
                                            (neg (INT 1022))";;
```

2.1 Floating-point Number Representation

Floating-point numbers are represented in HOL by their meaning: a value, and infinite, and a NaN. A new type is created to define floating point numbers:

```
define_type `fp_num`
'fp_num = finite (num#integer#(num -> num)) |
infinite num |
NaN (NaN_type#num)';;
```

"`finite`", "`infinite`", and "`NaN`" become type constructors that when applied to a triple of type "`:(num#integer#(num -> num))`", an element of type "`:num`", and a pair of type "`:(NaN_type#num)`", respectively, will return an element of type "`:fp_num`".

A new type is used in the definition of "`fp_num`" above which defines signaling and quiet NaNs:

```
define_type `NaN_type` `NaN_type = signal | quiet`;;
```

The following definitions for identifying and manipulating floating-point(fp) numbers are used in the specification of floating-point operations:

```
new_definition(`is_finite`,
"is_finite fp = (?X.fp = (finite X))");;

new_definition(`is_infinite`,
"is_infinite fp = (?X.fp = (infinite X))");;

new_definition(`is_NaN`,
"is_NaN fp = (?X.fp = (NaN X))");;

new_definition(`i_finite`,
"i_finite fp = (@X.fp = (finite X))");;

new_definition(`i_infinite`,
"i_infinite fp = (@X.fp = (infinite X))");;

new_definition(`i_NaN`,
"i_NaN fp = (@X.fp = (NaN X))");;
```

The first three definitions are predicates which return *true* when applied to a finite, infinite, and NaN fp number, respectively, and *false* otherwise. The last three definitions are the inverse of the respective type constructors and will return the argument of the constructor when applied to the appropriate fp number. The theorems,

```
|- !z.i_finite (finite z) = z
|- !z.i_infinite (infinite z) = z
|- !z.i_NaN (NaN z) = z
```

illustrate the action of the inverse functions.

Additional definitions following extract the elements of the arguments for floating point constructors and identify properties of the argument:

```
new_definition(`fp_sign_d`,
"fp_sign_d fp = (@n.(n = (FST (i_finite fp)))\/
                (n = (i_infinite fp)))");;

define_type `fp_sign`
`fp_sign = positive | negative`;;

new_definition(`fp_sign`,
"fp_sign fp = (EVEN (fp_sign_d fp)) => positive | negative");;

new_definition(`exponent`,
"exponent (s:num,Exp:integer,dig:num -> num) = Exp");;

new_definition(`digits`,
"digits (s:num,Exp:integer,dig:num -> num) = dig");;

new_definition(`fp_is_pos`,
"fp_is_pos fp = fp_sign fp = positive");;

new_definition(`fp_is_neg`,
"fp_is_neg fp = fp_sign fp = negative");;

new_definition(`fp_is_zero`,
"fp_is_zero fp = (!n.(fp_digits fp)n = 0)");;
```

The greatest and least magnitudes for a finite floating point number is given by:

```
new_definition(`Gpos`,
"Gpos (emax:integer) = (0,emax,(\d:num.b-1))");;

new_definition(`Gneg`,
"Gneg (emax:integer) = (1,emax,(\d:num.b-1))");;

new_definition(`Lpos`,
"Lpos p (emin:integer) = (0,emin,\d:num.(d = (p-1)) => 1 | 0)");;

new_definition(`Lneg`,
"Lneg p (emin:integer) = (1,emin,\d:num.(d = (p-1)) => 1 | 0)");;
```

3 Exceptions and Traps

Operations on floating-point numbers can signal exceptions as a result of performing the operation. Exceptions are declared as a new type:

```
define_type `except` `except_type = invalid | div_by_zero |
overflow_w_inex | underflow | underflow_w_inex | inexact |
no_excep`;;
```

A signaling exception will set a status flag and, if enabled by the user, will invoke an exception handling trap. If exceptions handlers are implemented then each exception should have a user controlled trap associated with it.

The resulting value on some operations will depend on whether an exception is detected and/or a trap is enabled. Conditions which will result in exceptions will be defined within the operation's definition. The status of the exception traps are defined by the 5-tuple (*invalid, div_by_zero, overflow, underflow, inexact*). The following functions extract the status of each of the exception traps:

```
new_definition(`invalid_t`,
"invalid_t (t1:bool,t2:bool,t3:bool,t4:bool,t5:bool) = t1");;

new_definition(`div_by_zero_t`,
"div_by_zero_t (t1:bool,t2:bool,t3:bool,t4:bool,t5:bool) = t2");;

new_definition(`overflow_t`,
"overflow_t (t1:bool,t2:bool,t3:bool,t4:bool,t5:bool) = t3");;

new_definition(`underflow_t`,
"underflow_t (t1:bool,t2:bool,t3:bool,t4:bool,t5:bool) = t4");;

new_definition(`inexact_t`,
"inexact_t (t1:bool,t2:bool,t3:bool,t4:bool,t5:bool) = t5");;
```

4 Rounding

Rounding will take an infinitely precise number *r*, characterized in the HOL system by the real numbers, and convert it into a floating-point representation. Four rounding modes are specified in the standard. The rounding mode is declared as a new type:

```
define_type `round_m` `round_m = to_near |
to_pos_inf | to_neg_inf | to_zero`;;
```

The rounding operation is defined by a family of functions to cover all rounding modes and argument values. The first set of function is defined for values of *r* which will generate a finite floating-point representation. A function is defined for each of the four rounding modes. These four functions take a real number, a rounding precision, and a destination precision predicate and return a finite floating-point number. The real number argument must have value such that it can be represented by a finite floating point for the given destination precision. Round to near is defined by,

```
new_definition(`round2near`,
```

```
"round2near r p precis =
  (?fp1.precis fp1  /\
(!fp.(precis fp) /\~(fp_value fp p  = fp_value fp1 p) ==>
abs(fp_value fp1 p real_sub r) real_lt abs(fp_value fp p real_sub r))) =>
  (@fp1.precis fp1  /\
(!fp.(precis fp) /\~(fp_value fp p = fp_value fp1 p) ==>
abs(fp_value fp1 p real_sub r) real_lt abs(fp_value fp p real_sub r))) |
   @fp1.(precis fp1) /\
(!fp.(precis fp) ==>
abs(fp_value fp1 p real_sub r) real_le abs(fp_value fp p real_sub r)) /\
             (EVEN ((digits fp1)(p-1)))");;
```

Round to near will return a floating-point number with a unique value nearest to the real number, if one exists. If two floating point numbers have values equally near, round to near will return the one with least significant digit even. Round to near uses the function "`fp_value`" which extracts the value of a floating-point number returning a real number. The function "`fp_value`" is defined by,

```
new_definition(`fp_value`,
"fp_value (s,Exp,dig) p =
 ((real_neg (& 1)) pow s) real_mul
 ((NEG Exp => (real_inv (&(b EXP (SND (REP_integer Exp))))) |
                          (&(b EXP (FST (REP_integer Exp)))))) real_mul
 (frac_sum (\dn.& (dig dn) real_mul (real_inv (&(b EXP dn)))) p)");;
```

The value function "`fp_value`" depends in turn on the summation function

"`frac_sum Fn m`" $= \sum_{n=0}^{m-1} Fn(n)$,

```
new_prim_rec_definition(`frac_sum`,
"(frac_sum Fn 0 = & 0)/\
(frac_sum Fn (SUC n) =
(Fn n) real_add (frac_sum Fn n))");;
```

Round to positive infinity returns the smallest floating-point number greater than r:

```
new_definition(`round2pinf`,
"round2pinf r p precis =
@fp1.(r real_le (fp_value fp1 p))/\
      (precis fp1)/\
      (!fp.r real_le (fp_value fp p) ==>
                        (fp_value fp1 p) real_le (fp_value fp p))");;
```

Round to negative infinity returns the largest floating-point number less than r:

```
new_definition(`round2ninf`,
"round2ninf r p precis =
@fp1.((fp_value fp1 p) real_le r)/\
      (precis fp1)/\
      (!fp. (fp_value fp p) real_le r ==>
                          (fp_value fp p) real_le (fp_value fp1 p))");;
```

Round to zero returns the largest magnitude floating-point number with magnitude less than the magnitude of r:

```
new_definition(`round2zero`,
"round2zero r p precis =
@fp1.(precis fp1)/\
      (!fp. abs(fp_value fp p) real_le (abs r) ==>
                    abs(fp_value fp p) real_le abs(fp_value fp1 p))");;
```

The next set of rounding functions is defined for real number arguments with unbound values. The real number value may be outside the representable range of finite floating-point numbers. When rounding is performed on unbounded real arguments, the rounding function must check for overflow and return an overflow exception flag when overflow is detected. The rounding functions will also check for underflow and inexact, and return the appropriate flag when an exception is detected.

The functions take as arguments a real number, rounding precision, traps status, rounding mode, tininess detection flag, accuracy detection flag, and destination maximum and minimum exponent. They return a floating point number and an exception flag.

Underflow and inexact detection are handled by separate functions outside the rounding operation. Overflow is detected inside the rounding function. Also, when the real number to be rounded has magnitude less than $b^{E_{min}}$ the rounding is handled by a separate function "denormal".

The functions "tininess", "accuracy", and "underfl" are used for underflow detection. The function "inex" is used both for underflow and inexact detection:

```
new_definition(`tininess`,
"tininess r p mode tiny emax emin =
let round = (  (mode = to_near)    => round2near  |
               (mode = to_pos_inf) => round2pinf  |
               (mode = to_neg_inf) => round2ninf  |
                                      round2zero   ) in
~tiny => (~(&0 = r) /\
         abs(r) real_lt (real_inv (&(b EXP (SND (REP_integer emin)))))  |
         (~(&0 = r) /\
         fp_value (round r p (precis_c emax emin)) p real_lt
                      (real_inv (&(b EXP (SND (REP_integer emin))))))" );;

new_definition(`accuracy`,
"accuracy r p mode acc emax emin =
let round = (  (mode = to_near)    => round2near  |
               (mode = to_pos_inf) => round2pinf  |
               (mode = to_neg_inf) => round2ninf  |
                                      round2zero   ) in
~acc => ~(fp_value (round r p exp_unbound) p = r) |
        ~(fp_value (round r p (precis_c emax emin)) p = r)" );;

new_definition(`underfl`,
"underfl r p traps mode tiny acc emax emin =
let u = (~(underflow_t traps) =>
   (tininess r p mode tiny emax emin /\ accuracy r p mode acc emax emin) |
    tininess r p mode tiny emax emin) in
 u => underflow | no_excep" );;
```

```
new_definition(`inex`,
"inex r p mode emax emin =
let round = (  (mode = to_near)     => round2near  |
               (mode = to_pos_inf) => round2pinf  |
               (mode = to_neg_inf) => round2ninf  |
                                      round2zero   ) in
(fp_value (round r p (precis_c emax emin)) p = r) => no_excep |
                                                     inexact" );;
```

When both underflow and inexact are detected the exception flag becomes `underflow_w_inex`:

```
new_definition(`underflow_inexact`,
"underflow_inexact r p traps mode tiny acc emax emin =
((underfl r p traps mode tiny acc emax emin = underflow) /\
(inex r p mode emax emin = inexact))                  =>  underflow_w_inex |
(underfl r p traps mode tiny acc emax emin = underflow)=>  underflow        |
(inex r p mode emax emin = inexact)                   =>  inexact          |
                                                           no_excep");;
```

If overflow is detected during rounding and the overflow trap handler is enabled the result of the operation will be the infinitely precise result of the operation divided by b^{α} and then rounded. The exponent adjustment α is chosen to be approximately $3((E_{max} - E_{min})/4)$ and should be divisible by twelve[3]:

```
new_definition(`alpha`,
"alpha emax emin =
let app = (3*(FST (REP_integer (emax minus emin)))) in
let q = @n. (48*n) < app /\ app < (48*(n+1)) in
(app - (48*q)) < ((48*(q+1)) - app) => 12*q | 12*(q+1)");;

new_definition(`r_to_near`,
"r_to_near r p traps mode tiny acc emax emin =
let thr = (&(b EXP (FST (REP_integer emax)))) real_mul
(&b real_sub (real_inv(&(b EXP (p-1)))/&2)) in
let bemin = (real_inv (&(b EXP (SND (REP_integer emin))))) in
  (r = &0)             => (finite (0,INT 0,\n.0)), no_excep             |
  abs(r) real_lt bemin => denormal r p traps mode tiny acc emax emin     |
  abs(r) real_lt thr   => (finite (round2near r p (precis_c emax emin))),
                            inex r p mode emax emin                      |
 (overflow_t traps)   => finite (round2near (r/(&(b EXP (alpha emax emin))))
                            p (precis_c emax emin)), overflow_w_inex     |
                         infinite (rsign r), overflow_w_inex  ");;
```

3. The definition of "`alpha`" is overly restrictive since it gives the exponent adjustment the nearest value to $3((E_{max} - E_{min})/4)$ which is divisible by 12. If an implementation description uses a different value for the exponent adjustment and a proof of compliance is to be performed, a new value for "`alpha`" should be defined consistent with the intended implementation.

The case for $|r| < b^{E_{min}}$ is handled by the "`denormal`" function. If underflow is detected and the underflow trap is enabled, denormal will return the infinitely precise result multiplied by b^{α} and then rounded with the selected rounding mode:

```
new_definition(`denormal`,
"denormal r p traps mode tiny acc emax emin =
let round = (  (mode = to_near)    => round2near  |
               (mode = to_pos_inf) => round2pinf  |
               (mode = to_neg_inf) => round2ninf  |
                                      round2zero   ) in
(~(underflow_t traps)\/
((underflow_t traps)/\
(underfl r p traps mode tiny acc emax emin = no_excep))    =>
 (    (is_zero (round r p (precis_c emax emin)))     =>
      (finite (rsign r,INT 0,\n.0)), underflow_w_inex |
      (finite (round r p (precis_c emax emin))),
         underflow_inexact r p traps mode tiny acc emax emin
 )  |
 (finite (round (r real_mul (&(b EXP (alpha emax emin)))) p
  (precis_c emax emin)),
          underflow_inexact r p traps mode tiny acc emax emin)");;
```

Round to positive infinity:

```
new_definition(`r_to_pinf`,
"r_to_pinf r p traps mode tiny acc emax emin =
let thr = real_neg (&(b EXP (FST (REP_integer emax) +1))) in
let bemin = (real_inv (&(b EXP (SND (REP_integer emin))))) in
  (r = &0)             => (finite (0,INT 0,\n.0)), no_excep                |
  abs(r) real_lt bemin => denormal r p traps mode tiny acc emax emin       |
  ((thr real_lt r) /\ (r real_le (fp_value (Gpos emax) p)))
                       => (finite (round2pinf r p (precis_c emax emin))),
                             inex r p mode emax emin                       |
  (overflow_t traps)   =>finite (round2pinf (r/(&(b EXP (alpha emax emin))))
                                 p (precis_c emax emin)), overflow_w_inex  |
   ( (r real_le thr)   => finite (Gneg emax), overflow_w_inex              |
                            infinite 0, overflow_w_inex  )");;
```

Round to negative infinity:

```
new_definition(`r_to_ninf`,
"r_to_ninf r p traps mode tiny acc emax emin =
let thr = (&(b EXP (FST (REP_integer emax) +1))) in
let bemin = (real_inv (&(b EXP (SND (REP_integer emin))))) in
  (r = &0)             => (finite (0,INT 0,\n.0)), no_excep                |
  abs(r) real_lt bemin => denormal r p traps mode tiny acc emax emin       |
  (((fp_value (Gneg emax) p) real_le r) /\ (r real_lt thr))
                       => (finite (round2ninf r p (precis_c emax emin))),
                                inex r p mode emax emin                    |
  (overflow_t traps)  => finite (round2ninf (r/(&(b EXP (alpha emax emin))))
```

```
                              p (precis_c emax emin)), overflow_w_inex   |
(  (thr real_le r)  => finite (Gpos emax), overflow_w_inex               |
                          infinite 1, overflow_w_inex  )");;
```

Round to zero:

```
new_definition(`r_to_zero`,
"r_to_zero r p traps mode tiny acc emax emin =
let thr = (&(b EXP (FST (REP_integer emax) +1))) in
let bemin = (real_inv (&(b EXP (SND (REP_integer emin))))) in
  (r = &0)             => (finite (0,INT 0,\n.0)), no_excep              |
  abs(r) real_lt bemin => denormal r p traps mode tiny acc emax emin     |
  abs(r) real_lt thr   => (finite (round2zero r p (precis_c emax emin))),
                            inex r p mode emax emin                      |
  (overflow_t traps)  => finite (round2zero (r/(&(b EXP (alpha emax emin))))
                               p (precis_c emax emin)), overflow_w_inex  |
  r real_lt &0         => finite (Gneg emax), overflow_w_inex            |
                          finite (Gpos emax), overflow_w_inex  ");;
```

The function "`round`" is the main function defining rounding. It uses the previous functions to define the rounding operation for all rounding modes and value ranges:

```
new_definition(`round`,
"round r p traps mode tiny acc emax emin =
   (mode = to_near)    => r_to_near r p traps mode tiny acc emax emin |
   (mode = to_pos_inf) => r_to_pinf r p traps mode tiny acc emax emin |
   (mode = to_neg_inf) => r_to_ninf r p traps mode tiny acc emax emin |
                          r_to_zero r p traps mode tiny acc emax emin ");;
```

5 Operations

In accordance with the IEEE-854 standard,

> *... each operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbound range, and then coerced this intermediate result to fit in the destination's precision.* [6, section 5, page 10]

Infinite precision for a floating-point operation is represented in the HOL system by the real numbers. When an operation is to be performed where the argument or arguments are finite floating-point numbers, the arguments are converted to real numbers, the operation is performed in real number arithmetic and the result is rounded according to the selected rounding mode. When operations are performed on arguments of different precisions, the lower precision argument is converted to the higher precision. The function "`p_conv`" define such conversion:

```
new_definition (`p_conv`,
"p_conv fp ps pl = is_infinite fp     => fp        |
                   is_NaN fp          => fp        |
finite(fp_sign_d fp,fp_exponent fp,\n.n < ps => fp_digits fp n |
                                      n < pl => 0              |
                                             @n.n < b)");;
```

“p_conv” converts a fp from a lower precision with “ps” significant digits to a higher precision with “pl” significant digits.

5.1 Arithmetic

Five operations are defined: addition, subtraction, multiplication, division, and remainder. “fp_arith” is then defined as an executive function which checks for NaN operands, normalizes the operands, and call an arithmetic operation based on the argument “op”. The operations are declared as a new type*:*

```
define_type `arith_op` `arith_op = fpadd | fpsub | fpmul | fpdiv |
fprem `;;
```

Some arguments to an operation might not be valid arguments depending on the operation. Division by zero is an example. If arguments to an operation are invalid the operation will return a NaN with an exception flag. Floating-point addition is defined by the function “fp_add”. “fp_add” takes as arguments floating-point operands “fp1” and “fp2”, operands’ precsion “p”, rounding precision “pr”, quiet NaN argument “cn”, trap status “traps”, rounding mode “mode”, tininess detection flag “tiny”, accuracy detection flag “acc”, and maximum and minimum exponents “emax” and “emin”:

```
new_definition (`fp_add`,
"fp_add fp1 fp2 p pr cn traps mode tiny acc emax emin =
(is_infinite fp1 /\ is_infinite fp2 /\ ~(fp_sign fp1 = fp_sign fp2))
                                   => (NaN(quiet,cn),invalid)        |
(is_infinite fp1)                  => (fp1,no_excep)                 |
(is_infinite fp2)                  => (fp2,no_excep)                 |
((fp_is_zero fp1)/\(fp_is_zero fp2)/\(fp_sign fp1 = fp_sign fp2))
                                   => (fp1,no_excep)                 |
round ((fp_value (i_finite fp1) p) real_add
          (fp_value (i_finite fp2) p)) pr traps mode tiny acc emax emin");;
```

Floating-point negation changes the arithmetic sign of a floating-point number which is not a NaN:

```
new_definition (`fp_neg`,
"fp_neg fp =
is_NaN fp    => fp                                                  |
is_finite fp => (fp_is_pos fp => (finite (1,(SND (i_finite fp))))    |
                                 finite (0,(SND (i_finite fp))))    |
                (fp_is_pos fp => infinite 1 | infinite 0)");;
```

Subtraction is defined in terms of negation and addition:

```
new_definition (`fp_sub`,
"fp_sub fp1 fp2 p pr cn traps mode tiny acc emax emin =
fp_add fp1 (fp_neg fp2) p pr traps mode tiny acc emax emin");;
```

Floating-point multiplication:

```
new_definition (`fp_mul`,
"fp_mul fp1 fp2 p pr cn traps mode tiny acc emax emin =
((is_infinite fp1 /\ fp_is_zero fp2)\/(fp_is_zero fp1 /\ is_infinite fp2))
```

```
                                       => (NaN(quiet,cn),invalid)          |
((is_infinite fp1 \/ is_infinite fp2)/\
 (fp_sign fp1 = fp_sign fp2))          => (infinite 0,no_excep)            |
((is_infinite fp1 \/ is_infinite fp2)/\
 ~(fp_sign fp1 = fp_sign fp2))         => (infinite 1,no_excep)            |
(((fp_is_zero fp1)\/(fp_is_zero fp2))/\
(fp_sign fp1 = fp_sign fp2))           => (finite (0,INT 0,\n.0)),no_excep |
(((fp_is_zero fp1)\/(fp_is_zero fp2))/\
~(fp_sign fp1 = fp_sign fp2))          => (finite (1,INT 0,\n.0)),no_excep |
round ((fp_value (i_finite fp1) p) real_mul
          (fp_value (i_finite fp2) p)) pr traps mode tiny acc emax emin");;
```

Floating-point division:

```
new_definition (`fp_div`,
"fp_div fp1 fp2 p pr cn traps mode tiny acc emax emin =
((fp_is_zero fp1 /\ fp_is_zero fp2)\/(is_infinite fp1 /\ is_infinite fp2))
                                       => (NaN(quiet,cn),invalid)         |
((fp_is_zero fp2)/\
(fp_sign fp1 = fp_sign fp2))           => (infinite 0,div_by_zero)        |
((fp_is_zero fp2)/\
~(fp_sign fp1 = fp_sign fp2))          => (infinite 1,div_by_zero)       |
round ((fp_value (i_finite fp1) p) / (fp_value (i_finite fp2) p)) pr
 traps mode tiny acc emax emin");;
```

Floating-point remainder is defined by IEEE-854 as follows:

> *When $y \neq 0$, the reminder $r = x\ REM\ y$ 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 x/y; whenever $|n - x/y| = 1/2$, then n is even. If $r = 0$, its sign shall be that of x.* [6, section 5.1, page 10]

The remainder function is defined in HOL by:

```
new_definition (`fp_rem`,
"fp_rem fpx fpy p (pr:num) cn traps (mode:round_m) tiny acc emax emin =
(is_infinite fpx \/ fp_is_zero fpy)   => (NaN(quiet,cn),invalid)          |
(is_infinite fpy)                     => (fpx,no_excep)                   |
 (let r = (fp_value (i_finite fpx) p real_sub
  ((fp_value (i_finite fpy) p real_mul
 (real_to_int_real
         ((fp_value (i_finite fpx) p) / (fp_value (i_finite fpy) p))))))
  in
  (r = &0)  =>
  (finite (fp_sign_d fpx,INT 0,\n.0)),no_excep |
  round r p traps to_near tiny acc emax emin)");;
```

The integer n nearest the exact value x/y is obtained using the function "`real_to_int_real`". The function "`real_to_int_real`", defined in section 5.8, takes a real number and returns the nearest real number with integer value.

"`fp_arith`" is the executive arithmetic function which filters NaN operands, normalizes the operands, and selects the arithmetic operation. When both operands are quiet NaNs the argument "`sel`" determines which of the operands will be returned by the arithmetic operaion. The arguments "p1" and "p2" determine the operands' precision and if normalization is needed. "pr" is the number of significant digits used in the rounding operation[4]. The destination precision is the largest of the two precisions, in accordance with IEEE-854 standard requirements.

```
   new_definition (`fp_arith`,
   "fp_arith op fp1 fp2 p1 p2 pr cn sel traps mode tiny acc emax1 emax2 emin1
emin2 =
   (\fp1c,fp2c,p,emax,emin.
   ((is_s_NaN fp1) \/ (is_s_NaN fp2))   => (NaN(quiet,cn),invalid)        |
   ((is_q_NaN fp1) /\ (is_q_NaN fp2))   => ((sel => fp1 | fp2),no_excep)  |
   (is_q_NaN fp1)                       => (fp1,no_excep)                 |
   (is_q_NaN fp2)                       => (fp2,no_excep)                 |
      (op = fpadd) => fp_add fp1c fp2c p pr cn traps mode tiny acc emax emin  |
      (op = fpsub) => fp_sub fp1c fp2c p pr cn traps mode tiny acc emax emin  |
      (op = fpmul) => fp_mul fp1c fp2c p pr cn traps mode tiny acc emax emin  |
      (op = fpdiv) => fp_div fp1c fp2c p pr cn traps mode tiny acc emax emin  |
                       fp_rem fp1c fp2c p pr cn traps mode tiny acc emax emin)
    ((p1 = p2) =>  (fp1,fp2,p1,emax1,emin1)                  |
     (p1 < p2) =>  ((p_conv fp1 p1 p2),fp2,p2,emax2,emin2)   |
                   (fp1,(p_conv fp2 p2 p1),p1,emax1,emin1))");;
```

5.2 Square root

The result is defined and is positive for all operands greater than zero, except that sqr of -0 is -0.

```
   new_definition (`fp_sqr`,
   "fp_sqr fp p pr cn traps mode tiny acc emax emin =
   (is_s_NaN fp)                         => (NaN(quiet,cn),invalid)    |
   (is_q_NaN fp)                         => (fp,no_excep)              |
   ((fp_is_neg fp) /\ ~(fp_is_zero fp))  => (NaN(quiet,cn),invalid)    |
   (is_infinite fp)                      => (infinite 0,no_excep)      |
   (fp_is_zero fp)                       => fp, no_excep               |
   (round (sqrt (fp_value (i_finite fp) p))
                                      pr traps mode tiny acc emax emin)");;
```

5.3 Precision conversions

Conversion between floating-point numbers of all precisions shall be possible. When converting from a lower to a higher precision the result will be exact. Conversion from a higher to lower precision may signal inexact.

```
   new_definition (`fp_p_conv`,
   "fp_p_conv fp p1 p2 cn traps mode tiny acc emax2 emin2 =
```

4. In most cases rounding precision is the precision of its destination. However, in systems where the result is always delivered to double or extended destinations, the user has the option of specifying a lower rounding precision than the destination precision. The result will be stored with the exponent range of the higher precision.

```
(is_s_NaN fp)                         => (NaN(quiet,cn),invalid)      |
(is_q_NaN fp)                         => (fp,no_excep)                |
(is_infinite fp)                      => (fp,no_excep)                |
     (p1 < p2)    =>   ((p_conv fp p1 p2),no_excep)  |
     (p1 = p2)    =>   (fp,no_excep)                 |
   round (fp_value (i_finite fp) p1) p2 traps mode tiny acc emax2 emin2");;
```

5.4 Conversion between Floating-point and Integer

Floating-point to integer conversion is defined in HOL by converting finite floating-point numbers to an integer number and converting infinities and NaNs to an unspecified integer number:

```
new_definition (`fp_int_conv`,
"fp_int_conv fp p mode =
(is_s_NaN fp)       => ran_int, invalid            |
(is_q_NaN fp)       => ran_int, no_excep           |
(is_infinite fp)    => ran_int, no_excep           |
(fp_is_zero fp)     => INT 0, no_excep             |
                       finite2int fp p mode, exc3 fp p mode");;
```

The unspecified integer number is:

```
new_definition (`ran_int`,
"ran_int = @N:integer.T");;
```

The conversion of non-zero finite floating-point numbers (of type ":fp_num") to integer numbers (of type "`:integer`") is defined by:

```
new_definition (`finite2int`,
"finite2int fp p mode =
let r = abs(fp_value (i_finite fp) p) in
let n =
(  (mode = to_near)   =>
             ( (EVEN (floor r)) =>
               floor (&(ceiling (&2 real_mul r))/&2) |
               ceiling (&(floor (&2 real_mul r))/&2)   )     |
   (mode = to_pos_inf) =>
             ( (fp_is_neg fp)  =>
               floor r  |
               ceiling r           )                          |
   (mode = to_neg_inf) =>
             ( (fp_is_neg fp)  =>
               ceiling r  |
               floor r             )                          |
               floor r  ) in
fp_is_neg fp => neg (INT n) |
                     INT n   ");;
```

The function "`exc3`" defines the inexact exception flag for the floating-point to integer conversion:

```
new_definition (`exc3`,
"exc3 fp p mode =
let r = abs(fp_value (i_finite fp) p) in
```

```
let N = finite2int fp p mode in
fp_is_neg fp =>
  ( (abs r = &(SND (REP_integer N))) => no_excep |
                                        inexact    )    |
  ( (    r = &(FST (REP_integer N))) => no_excep |
                                        inexact    ) ");;
```

Integer to floating point is accomplished by converting the integer number to a real number and using the round function to obtain a floating-point number.

```
new_definition(`int_fp_conv`,
"int_fp_conv N p traps mode tiny acc emax emin =
let r = ( NEG N => real_neg (&(SND (REP_integer N))) |
                  &(FST (REP_integer N)) ) in
FST (round r p traps mode tiny acc emax emin)");;
```

5.5 Conversion of Floating-point to Integral valued floating-point

Conversion of floating-point to an integer valued floating-point is defined by conversion of floating-point to an integer and from integer back to floating point. This conversion leaves infinities and quiet NaNs unchanged and generates an invalid exception for signaling NaNs.

```
new_definition (`fp_fp_int_conv`,
"fp_fp_int_conv fp p cn traps mode tiny acc emax emin =
(is_s_NaN fp)       => (NaN(quiet,cn)), invalid     |
(is_q_NaN fp)       => fp, no_excep                 |
(is_infinite fp)    => fp, no_excep                 |
(fp_is_zero fp)     => fp, no_excep                 |
 ((exc3 fp p mode = no_excep) =>
   fp, no_excep                                                      |
   (int_fp_conv (finite2int fp p mode) p traps mode tiny acc emax emin),
                 inexact)");;
```

When the conversion from floating-point to integer is exact (exc3 = no_excep) the floating-point number already has an integral value and no conversion is necessary. When the conversion from floating-point to integer is inexact conversion takes place and the inexact exception is raised.

5.6 Conversion between floating-point and decimal string

Decimal strings are strings of characters representing decimal numbers or a string of characters representing non-valued entities. A partial characterization of a decimal string is performed in HOL by defining a new type:

```
define_type `decimal_string`  `decimal_string = quiet_nan | signaling_nan |
nan | unrecognizable | nzero | pzero | zero | ninf | pinf | inf |
format real `;;
```

The elements "`nzero`", "`pzero`", "`zero`", and "`format real`" represent values. The elements "`quiet_nan`", "`signaling_nan`", "`nan`", "`unrecognizable`", "`ninf`", "`pinf`", and "`inf`" represent non-valued decimal strings. Note that there exists an overlap in the representation of the value 0. The value of a decimal string "`format real`" is the argument "real" to the type constructor "`format`".

```
new_definition(`ds_value`,
"ds_value ds = @r.ds = (format r)");;
```

The floating-point number corresponding to each of the decimal string elements is given by the function:

```
new_definition(`ds_fp_conv`,
"ds_fp_conv ds p cn traps mode tiny acc emax emin =
(ds = quiet_nan)      => (NaN(quiet,cn)), no_excep      |
(ds = signaling_nan)  => (NaN(sign,cn)), no_excep       |
(ds = nan)            => (NaN(sign,cn)), no_excep       |
(ds = unrecognizable) => (NaN(quiet,cn)), invalid       |
(ds = nzero)          => finite(1,INT 0,\n.0), no_excep |
(ds = pzero)          => finite(0,INT 0,\n.0), no_excep |
(ds = zero)           => finite(0,INT 0,\n.0), no_excep |
(ds = ninf)           => infinite 1, no_excep           |
(ds = pinf)           => infinite 0, no_excep           |
(ds = inf)            => infinite 0, no_excep           |
 round (ds_value ds) p traps mode tiny acc emax emin");;
```

Floating-point to decimal string conversion maps floating-point numbers to the corresponding decimal strings. Floating-point to decimal string conversion is defined in a relational style to permit more than one decimal string for a given floating-point number. The predicate "`fp_ds_conv`" take as arguments floating-point "`fp`", floating-point precision "`p`" and decimal string "`ds`":

```
new_definition(`fp_ds_conv`,
"fp_ds_conv fp p ds =
(is_s_NaN fp)      => ((ds=(signaling_nan,invalid))\/(ds=(nan,invalid)))   |
(is_q_NaN fp)      => ((ds=(quiet_nan,no_excep))\/(ds=(nan,no_excep)))     |
(fp_is_zero fp/\fp_is_neg fp)       => (ds = (nzero, no_excep))            |
(fp_is_zero fp)    => ((ds=(pzero,no_excep))\/(ds=(zero,no_excep)))        |
(is_infinite fp/\fp_is_neg fp)      => (ds = (ninf, no_excep))             |
(is_infinite fp)   => ((ds=(pinf,no_excep))\/(ds=(inf,no_excep)))          |
                       (ds = format (fp_value (i_finite fp) p),no_excep)");;
```

5.7 Comparison

For any two arbitrary floating-point numbers, one and only one of the following relations must hold: "less than", "equal", greater than", or "unordered". The four relations between floating-point numbers are defined by the type:

```
define_type `relations` `relations = less_than | equal | greater_than |
unordered`;;
```

The comparison operation can be defined in two optional ways: 1) by returning one of the possible four relations between the arguments; 2) by returning true or false on a given predicate. The first option is specified by the function:

```
new_definition(`relation`,
"relation fp1 fp2 p1 p2 =
```

```
(is_s_NaN fp1\/ is_s_NaN fp2)       => unordered, invalid         |
(is_q_NaN fp1\/ is_q_NaN fp2)       => unordered, no_excep        |
(is_infinite fp1/\is_infinite fp2/\(fp_sign fp1 = fp_sign fp2))
                                    => equal, no_excep            |
(is_infinite fp1/\(fp_is_neg fp1)  => less_than, no_excep        |
(is_infinite fp1/\(fp_is_pos fp1)  => greater_than, no_excep     |
(is_infinite fp2/\(fp_is_neg fp2)  => greater_than, no_excep     |
(is_infinite fp2/\(fp_is_pos fp2)  => less_than, no_excep        |
(fp_value (finite fp1) p1) real_lt (fp_value (finite fp2) p2)
                                    => less_than, no_excep        |
(fp_value (finite fp1) p1) = (fp_value (finite fp2) p2)
                                    => equal, no_excep            |
                                       greater_than, no_excep ");;
```

If the comparison operation is defined in terms of predicates, the following HOL definitions list six predicates that must be provided by the implementation and a seventh predicates which is desirable. The predicates are defined in terms of the function “relation”.

```
new_definition(`EQ`,
"EQ fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => F, no_excep  |
                                                 F, no_excep");;

new_definition(`NE`,
"NE fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => T, no_excep  |
                                                 T, no_excep");;

new_definition(`GT`,
"GT fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => T, no_excep  |
                                                 F, invalid");;

new_definition(`GE`,
"GE fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => T, no_excep  |
                                                 F, invalid");;

new_definition(`LT`,
"LT fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => F, no_excep  |
                                                 F, invalid");;
```

```
new_definition(`LE`,
"LE fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => T, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => F, no_excep  |
                                                 F, invalid");;

new_definition(`UN`,
"GT fp1 fp2 p1 p2 =
(FST (relation fp1 fp2 p1 p2) = less_than)    => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = equal)        => F, no_excep  |
(FST (relation fp1 fp2 p1 p2) = greater_than) => F, no_excep  |
                                                 T, no_excep");;
```

5.8 Supporting functions

This section includes some functions that are used within the definition of the IEEE-854 standard in the HOL system, but are more of a general nature than specific to the standard.

The function "`real_to_int_real r`" delivers the integer real number nearest "r". If two such numbers exist, "`real_to_int_real`" delivers an even integer real number.

```
new_definition(`real_to_int_real`,
"real_to_int_real r = (
(r real_ge &0) =>
  (&
  (((&(ceiling r) real_sub r) real_lt (r real_sub &(floor r))) => ceiling r |
    ((r real_sub &(floor r)) real_lt (&(ceiling r) real_sub r)) => floor r |
    (@n.((n = ceiling r)\/(n = floor r))/\(EVEN n))))
               |
 let rn = abs r in
(real_neg (&
(((&(ceiling rn) real_sub rn) real_lt (rn real_sub &(floor rn))) => ceiling rn  |
   ((rn real_sub &(floor rn)) real_lt (&(ceiling rn) real_sub rn)) => floor rn |
     (@n.((n = ceiling rn)\/(n = floor rn))/\(EVEN n)))))
                )");;
```

Logarithm base n of x is defined in terms of the natural logarithm provided in the *reals* library

```
new_definition(`log`,
"log n x = (ln x) real_mul (real_inv (ln (& n)))");;
```

The ceiling function when applied to a number "`x:real`" will return the least number "`n:num`" greater or equal to *x*. This function is only valid for non-negative values of *x*. When *x* is negative ceiling of *x* is zero.

```
new_definition(`ceiling`,
"ceiling x = @n.((& n) real_ge x) /\ (!i.((& i) real_ge x) ==> n <= i)");;
```

The floor function when applied to a number "`x:real`" will return the greatest "`n:num`" less than or equal to *x*. Floor is only valid for non-negative arguments. When *x* is negative floor of *x* is an undefined natural number.

```
new_definition(`floor`,
```

```
"floor x = @n.((& n) real_le x) /\ (!i.((& i) real_le x) ==> i <= n)");;
```

The function rsign returns a 1 if its argument of type "`:real`" is negative and 0 otherwise.

```
new_definition(`rsign`,
"rsign r = r real_lt &0 => 1 | 0" );;
```

6 References

[1] Geoff Barret. Formal Methods Applied to a Floating-Point Number System, IEEE Transactions on Software Engineering, volume 15, number 5, May 1989, pages 611-621.

[2] M.J.C. Gordon. A Proof Generating System for Higher-Order Logic, in VLSI Specification, Verification and Synthesis, G. Birtwistle and P.A. Subrahmanyam editors, Kluwer International Series in Engineering and Computer Science, SECS35, 1988.

[3] Elsa Gunter. (Integer library in HOL system distribution) March 1989.

[4] John Harrison. The HOL Reals Library. University of Cambridge Computer Laboratory, July 1992.

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

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

[7] Paul Miner. Defining the IEEE-854 Floating-Point Standard in PVS, NASA Technical Memorandum 110167, June 1995.

[8] Jing Pan. Formal Specification and Verification of a Floating-Point Coprocessor, Department of Computer Science, University of California, Davis, September 1990.