On The Design of IEEE Compliant Floating Point Units *

Guy Even† Wolfgang Paul‡

April 6, 2000

Abstract

Engineering design methodology recommends designing a system as follows: Start with an unambiguous specification, partition the system into blocks, specify the functionality of each block, design each block separately, and glue the blocks together. Verifying the correctness of an implementation reduces then to a local verification procedure.

We apply this methodology for designing a provably correct IEEE rounding unit that can be used for various operations, such as: addition and multiplication. First, we provide a mathematical and hopefully unambiguous definition of the IEEE Standard which specifies the functionality. We give explicit and concise rules for gluing the rounding unit with a floating-point adder and multiplier. We then present floating-point addition and multiplication algorithms that use the rounding unit.

To the best of our knowledge, our design is the first publication that deals with detecting exceptions and trapped overflow and underflow exceptions as an integral part of the rounding unit in a floating point unit. Our abstraction level avoids bit-level representations and arguments to help clarify the functionality of the algorithm.

Keywords: IEEE 754 Standard, floating-point arithmetic, floating-point rounding, floating-point addition.

1 Introduction

Background. The IEEE Standard for floating-point Arithmetic [6] defines the functionality of floating point arithmetic. Since its approval more than 10 years ago, the Standard has been an immense success, and all major floating-point units (FPU's) comply with it. Unfortunately, most designs are obtained by first designing an FPU that handles correctly most of the cases, and subsequent modifications are made to handle the remaining cases. The drawbacks of this approach are: (a) hard to predict performance and cost due to the modifications; (b) longer design process and complications in designs; and (c) hard to verify designs.

*A preliminary version of this work appeared in the Proceedings of the 13th IEEE Symposium on Computer Arithmetic, pp. 54-63, Asilomar, California, USA, July 6 - July 9, 1997.

†Dept. of Electrical Engineering-Systems, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail:guy@eng.tau.ac.il. This work was done while visiting the University of the Saarland. Supported in part by Graduiertenkolleg "Effizienz und Komplexität von Algorithmen und Rechenanlagen", Universität des Saarlandes, and in part by the North Atlantic Treaty Organization under a grant awarded in 1996.

‡Dept. of Computer Science, Univ. des Saarlandes, 66123 Saarbrücken, Germany. E-mail:wjp@cs.uni-sb.de.

Previous work. There is a vast amount of literature on designing FPU's. The literature falls mainly into two categories: (a) Fundamental work that describes how FPU's are designed but are either not fully IEEE compliant or ignore details involved with the Standard such as exceptions. Among such works are [1, 3, 4, 5, 8, 14, 15]. (b) Implementation reports that describe existing FPU's while emphasizing features that enable improved performance. Among the works that address design and optimization issues in floating-point adders and multipliers are [2, 11, 12, 13, 16].
Goals. We set forth the following goals concerning the design of an IEEE compliant FPU. (a) Provide an abstraction level that can bridge between a high level description and detailed issues involved in complying with the Standard. (b) Identify and characterize essential properties that explain the correctness of the circuit. (c) Formalize mathematical relations that can serve as "glue" for composing the building blocks together (e.g., a floating-point adder and a rounding unit).
Overview and Contributions. This paper deals systematically with the design of a rounding unit which is a major module in IEEE compliant FPU's. A key issue is the choice of an abstraction level. We use an abstraction level suggested by Matula [7] which enables ignoring representation issues and simplifies greatly the description since one can avoid bit-level arguments. The Standard specifies the functionality but is hard to interpret. We believe that it is imperative to have a clear and structured specification. In particular, we define representable values, rounding, and exceptions.

A block diagram of a rounding unit is described, and the functionality of the modules is clearly defined. We develop a terminology and formulate simple properties that enable us to provide a rigorous proof of the correctness of the rounding unit.

The IEEE standard requires that every operation be performed as if the exact result is computed and only then rounded. A key issue which we address is the fact that, to save cost and delay, a rounding unit is fed a value that may differ from the exact result. We quantify how close the value fed to the rounding unit should be so that the rounding unit will round correctly and detect exceptions correctly even though it is not fed the exact result. This enables us to formulate a succinct property that an output of a functional unit (e.g. a floating-point adder or a multiplier) must satisfy so that its output can be input to a rounding unit. The rounding unit then produces an output which is identical to its output had the exact result been fed to it. The advantage of this approach is modularity, since we provide a simple rule for gluing a functional unit to a rounding unit.

A floating-point addition algorithm is described using the language and tools that were developed for the rounding unit. The correctness of the floating-point addition algorithm is proved.
Organization. In Sec 2, representable numbers are defined by factorings. In Sec. 3, rounding is defined. In Sec. 4, exceptions are defined, and methods for detecting exceptions and dealing with trapped exceptions are proved. In Sec. 5, addresses the issue that the rounding unit is fed a number which might not equal the exact result that is supposed to be rounded. A formal definition of how close the input value must be to the exact value is given so that correct rounding and detection of exceptions can be achieved. In Sec. 6, a block diagram of a rounding unit is presented and its correctness is proved. In Sec. 7, a simple rule for gluing the rounding unit with an adder or multiplier is presented. In addition, mathematical tools that are used in proving the correctness of the addition and multiplication algorithms are presented. In Sec. 8, a high-level floating-point addition algorithm is presented and its correctness is proved. In Sec. 8, a floating point multiplication algorithm is briefly discussed. In Sec. 9, we conclude the paper. In Appendix A, the Standard's roles regarding representation are overviewed. In Appendix B, the Standard's

rules regarding trap handling are discussed.

2 IEEE Standard - Representable Numbers

In this section we discuss which numbers are representable according to the IEEE Standard. Our discussion uses an abstraction level that ignores representation issues (e.g. hidden bits and biased encoding of the exponent) focuses on the values that can be represented. We use factorings as suggested by Matula [7]. A factoring factors a real number into three components: a sign factor, a scale factor, and a significand. We define the set of representable floating point numbers by the set of representable factorings, namely, the combinations of sign, scale, and significand factors that are allowed according to the IEEE Standard.

2.1 Factorings

We begin by defining factorings of real numbers.

Definition 1 *A factoring is a triple* (s, e, f) *where:*

- $s \in \{0, 1, \}$ *is a sign bit;*
- *The exponent* e *is an integer; and*
- *The significand* f *is a non-negative real number.*

The value represented by a factoring, denoted by $\mathrm{val}(s, e, f)$*, is defined by*

$$\mathrm{val}(s, e, f) = (-1)^s \cdot 2^e \cdot f$$

Usually the range of the significand is limited to the half open interval $[0, 2)$, but intermediate results may have significands that are larger than 2. A natural issue is that of unique representation. We postpone this issue until Sec. 3.1 in which we define two unique factorings.

Factorings of $\pm\infty$. We need to introduce a factoring of $\pm\infty$ as well. We do that by introducing a special exponent symbol e_∞ and a special significand symbol f_∞ that must appear together in a factoring (namely, an integer exponent cannot appear with f_∞ and a real significand cannot appear with e_∞). Thus, the factoring of ∞ is $(0, e_\infty, f_\infty)$, and the factoring of $-\infty$ is $(1, e_\infty, f_\infty)$.

2.2 Standard's representable real numbers

In this section we define which real number are representable according to the IEEE Standard. The representable numbers are defined by the representable factorings, and hence, we discuss which exponent and significand values are representable, and which combinations of exponents and significands are allowed.

Parameters of a format. Every format (single, double, etc.) has two parameters attached to it:

1. n - the length of the exponent string. The set of representable exponent values is the set of all integers between $e_{\max}$ and $e_{\min}$. The values of $e_{\max}$ and $e_{\min}$ are determined by the parameter n as follows:

   minimum exponent: $e_{\min} \stackrel{\triangle}{=} 2 - 2^{n-1}$.
   maximum exponent: $e_{\max} \stackrel{\triangle}{=} 2^{n-1} - 1$.

   This range of exponent values results from the biased binary representation of exponents (see Appendix A).

2. p - the length of the significand string (including the hidden bit). The set of representable significand values is the set of all integral multiples of $2^{-(p-1)}$ in the half open interval $[0, 2)$.

The representable significand values are partitioned into two ranges, as follows:

1. Representable significand values in the half open interval $[0, 1)$ are called *denormalized*. (We find the term "tiny" more informative than "denormalized", and it is used later in the definition of the underflow for non-zero denormalized numbers. However, we decided to use the terminology of the IEEE Standard to avoid confusion.)

2. Representable significand values in the half open interval $[1, 2)$ are called *normalized*.

Not all combinations of representable exponent values and significand values are allowed. Specifically, the denormalized significands can only appear with the exponent value $e_{\min}$. However, the normalized significands may appear with all the representable exponent values. Table 1 summarizes the representable numbers for a given format (parameterized by n and p).

range	exponent	significand
denormalized	$e = e_{\min}$	$f \in \{i \cdot 2^{-(p-1)} : 0 \leq i < 2^{p-1}\} \subset [0, 1)$
normalized	$e_{\min} \leq e \leq e_{\max}$	$f \in \{1 + i \cdot 2^{-(p-1)} : 0 \leq i < 2^{p-1}\} \subset [1, 2)$

Table 1: Allowed factoring combinations for representable numbers

In Appendix A we briefly discuss how the representable numbers are represented as binary strings according to the IEEE Standard.

2.3 The Geometry of representable numbers

We depict the non-negative representable real numbers in Fig. 1; the picture for the negative representable numbers is symmetric.

The following properties characterize the representable numbers:

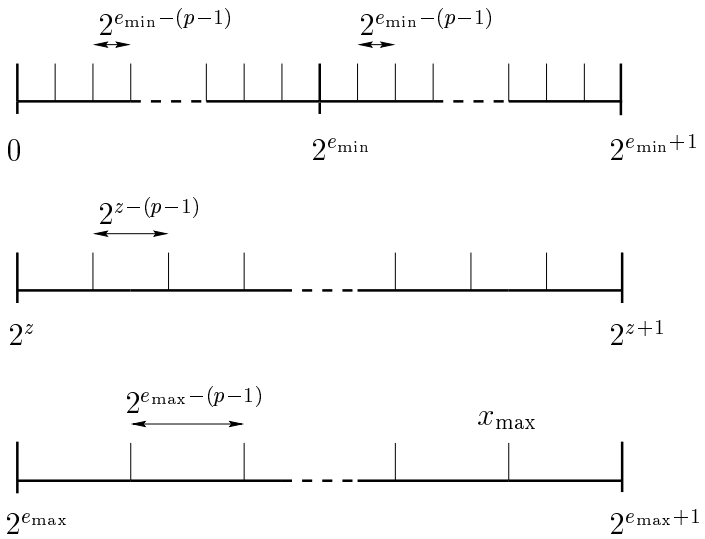

Figure 1: Geometry of Representable Numbers

1. For every exponent value z between $e_{\min}$ and $e_{\max}$ there are two intervals of representable numbers: $[2^z, 2^{z+1})$ and $(-2^{z+1}, -2^z]$. The gaps between consecutive representable numbers in these intervals are $2^{z-(p-1)}$.

2. As the exponent value increases by one, the length of the interval $[2^z, 2^{z+1})$ doubles, and the gaps between the representable numbers double as well. Thus, the number of representable numbers per interval is fixed and it equals 2^{p-1}.

3. The denormalized numbers, namely, the representable numbers in the interval $(-2^{e_{\min}}, 2^{e_{\min}})$, have an exponent value of $e_{\min}$ and a significand value in the interval $[0, 1)$. The gaps between consecutive representable denormalized numbers are $2^{e_{\min}-(p-1)}$. Thus, the gaps in the interval $[0, 2^{e_{\min}})$ equal the gaps in the interval $[2^{e_{\min}}, 2^{e_{\min}+1})$. This property is called in the literature *gradual underflow* since the large gap between zero and $2^{e_{\min}}$ is filled with denormalized numbers.

3 Rounding - Definition

In this section we define rounding as a mapping from factorings to factorings. We postpone dealing with trapped underflow and overflow exceptions to Sec. 4. The rounding mapping is obtained by a composition of four functions: normalization shift, significand rounding, post-normalization, and exponent rounding. The rounding mode effects only two of these functions: significand rounding and exponent rounding. These functions are later used as building blocks in the rounding unit. To simplify the exposition, we focus on round-to-nearest (even) mode, which is the default rounding mode according to the IEEE Standard.

3.1 Normalized Factoring

Every real number has infinitely many factorings. We would like to obtain a unique factoring by requiring that $1 \leq f < 2$ (i.e. f is normalized). However, there are two problems with this requirement: (a) zero cannot be represented; and (b) very small exponents (i.e. exponents less than $e_{\min}$) are undesirable since, if p bits are used to represent significands, then the least significant bits of the significands must be computed even they are not required for rounding. For example, as we show later, only 3 bits of precision suffice for rounding $2^{e_{\min}-(p-1)} + \varepsilon$ if $\varepsilon < 2^{e_{\min}-(p-1)}$. However, representing this number with a factoring in which $1 \leq f < 2$ would require p bits.

We define below a normalization shift and normalized factorings which define a unique factoring while avoiding the two problems mentioned above.

Definition 2 Normalization shift *is the mapping* norm_fac *from factorings that represent real numbers to factorings which satisfies:*

1. *Value is preserved:* $\text{val}(\text{norm_fac}(s, e, f)) = \text{val}(s, e, f)$.
2. *Let* $(s, en, fn) = \text{norm_fac}(s, e, f)$. *If* $2^e \cdot f \geq 2^{e_{\min}}$, *then* $1 \leq fn < 2$. *If* $2^e \cdot f < 2^{e_{\min}}$, *then* $en = e_{\min}$ *and* $0 \leq fn < 1$.

The following remarks apply to the normalization shift mapping:

1. An explicit definition of normalization shift is easy to write (and harder to read): $fn = 2^{e-en} \cdot f$ to preserve the value, and en is defined as follows: If $2^e \cdot f \geq 2^{e_{\min}}$, then $en = \lfloor \log_2(2^e \cdot f) \rfloor$. If $2^e \cdot f < 2^{e_{\min}}$, then $en = e_{\min}$.
2. The sign-bit in the normalization shift of zero is not well defined. However, the Standard sets rules regarding the sign bit when the exact result equals zero. For example, $(+5) \cdot (-0) = -0$ and $5 - 5 = +0$. Therefore, we assume that the rounding unit is input the correct sign bit and the rounding unit does not change the sign bit.

We also consider normalized factorings.

Definition 3 *A factoring* (s, en, fn) *is the* normalized factoring *of* (s, e, f) *if* $(s, en, fn) = \text{norm_fac}(s, e, f)$. *A factoring* (s, e, f) *is* normalized *if* $(s, e, f) = \text{norm_fac}(s, e, f)$.

We often say that (s, en, fn) is the normalized factoring of a number x, and mean that $x = val(s, en, fn)$ and the factoring is normalized.

We use the term "normalized" for two different purposes: a normalized significand is a significand in the range $[1, 2)$, whereas a normalized factoring is the unique factoring representing a real number. The disadvantage of this terminology is that a normalized factoring can have a denormalized significand. The reason for this is poor choice of terminology is that practically all the papers on floating-point arithmetic refer to the initial shifting of the input as a normalization shift.

3.2 Significand Rounding

Consider a significand $f \geq 0$. Suppose we would like to round f so that its binary representation has at most $p - 1$ bits to the right of the binary point (this implies p bits of precision when $f < 2$). Significand rounding in round-to-nearest (even) mode is defined as follows.

Definition 4 *Significand rounding, denoted by* $\mathrm{sig_rnd}(f)$*, is the real function defined as follows. Given a real number* f*, sandwich* f *by two consecutive integral multiples of* $2^{-(p-1)}$ *denoted by:* $a = q \cdot 2^{-(p-1)}$ *and* $b = (1+q) \cdot 2^{-(p-1)}$*, where* q *is an integer. Such that:*

$$a \leq f < b$$

(Note that f *may equal* a *but is strictly less than* b*.)* $\mathrm{sig_rnd}(f)$ *is defined according to the following three cases:*

1. *If* f *is strictly closer to* a *than to* b *(i.e.* $f - a < b - f$*), then* $\mathrm{sig_rnd}(f) = a$*.*
2. *If* f *is strictly closer to* b *then to* a*, (i.e.* $f - a > b - f$*), then* $\mathrm{sig_rnd}(f) = b$*.*
3. *If* f *is the midpoint between* a *and* b *(i.e.* $f - a = b - f$*), then* $\mathrm{sig_rnd}(f) = q' \cdot 2^{-(p-1)}$*, where* q' *is the even integer in* $\{q, q+1\}$*.*

Note that the rounding modes: round to $+\infty$ and round to $-\infty$, as defined in the IEEE Standard, are non-symmetric rounding modes. Namely, in these rounding modes, significand rounding depends also on the sign bit. For simplicity, we omit the sign bit as an argument of significand rounding. However, in our rounding unit depicted in Fig. 2, the sign bit is input to the significand rounding box.

3.3 Post-Normalization

It is important to note that if $f \in [0, 2)$, then the result of significand-rounding, *sig_rnd*(f), is in the range $[0, 2]$. The case that *sig_rnd*$(f) = 2$ is called *significand overflow*. When significand overflow occurs, an additional normalization shift is required, which is called *post-normalization*. Post-normalization, in the case of significand overflow sets the significand is set to 1 and increments the exponent, as defined below.

Definition 5 *Post-normalization, denoted by* $\mathrm{post_norm}(s, e, f)$*, is defined for a significand* $f \in [0, 2]$ *as follows:*

$$\mathrm{post_norm}(s, e, f) \triangleq \begin{cases} (s, e+1, 1) & \text{if } f = 2 \\ (s, e, f) & \text{otherwise} \end{cases}$$

3.4 Exponent Rounding

Exponent rounding maps factorings that represent real numbers into factorings (including factorings that represent infinities) and deals with the case that the absolute value of a factoring is too large (i.e. greater or equal to $2^{e_{\max}+1}$).

Definition 6 *Let* (s, e, f) *denote a normalized factoring. Exponent rounding, denoted by* $\mathrm{exp_rnd}$*, is defined in round-to-nearest (even) mode by:*

$$\mathrm{exp_rnd}(s, e, f) \triangleq \begin{cases} (s, e_\infty, f_\infty) & \text{if } e > e_{\max} \\ (s, e, f) & \text{otherwise} \end{cases}$$

The following remarks apply to the exponent rounding mapping:

1. Exponent rounding is defined differently for each rounding mode. The reader can easily define it for the other three rounding modes.

2. We assume that normalization shifting and significand rounding already take place before exponent rounding. Therefore, we may assume that (s, e, f) is a normalized factoring. Moreover, $val(s, e, f)$ is a representable number, unless its absolute value is too big. Hence, if $e \leq e_{\max}$, then $2^e \cdot f < 2^{e_{\max}} \cdot (2 - 2^{-(p-1)})$.

3.5 Rounding

Definition 7 *Rounding is a mapping of factorings that represent real numbers into factorings. The rounding of a factoring (s, e, f), denoted by $r(s, e, f)$, is defined as follows: Let (s, en, fn) denote the normalized factoring of (s, e, f), i.e. $(s, en, fn) = \text{norm_fac}(s, e, f)$, then*

$$r(s, e, f) \stackrel{\triangle}{=} \text{exp_rnd}\left(\text{post_norm}\left(s, en, \text{sig_rnd}(fn)\right)\right) \tag{1}$$

Rounding is the composition of four functions: (a) normalization shift - to obtain the right factoring; (b) significand rounding - to obtain the right precision (namely, limit the length of the significand); (c) post-normalization - to correct the factoring in case significand overflow occurs; and (d) exponent rounding - to limit the range of the value of the factoring. Note that if $val(r(s, e, f))$ is a number, then $r(s, e, f)$ is a normalized factoring. We sometimes abuse notation and refer to the rounding of a real number x, and denote it by $r(x)$. One should regard $r(x)$ as shorthand for "let $val(s, e, f) = x$ and consider $r(s, e, f)$".

Remark. The Standard defines rounding using a set of rules that define how a real number is rounded to a representable number (including the cases of $\pm\infty$). In [9, 10], a proof is given that the definitions of rounding in the Standard and in this paper are equivalent.

4 Standard's Exceptions

Five exceptions are defined by the Standard: invalid operation, division by zero, overflow, underflow, and inexact. The first two exceptions have to do only with the strings given as input to an operation rather than with the outcome of a computation [1]. Hence, we focus on the last three exceptions. The main advantage is that we can use our notation for short and precise definitions of the exceptions.

Exception handling consists of two parts: signaling the occurrence of an exception by setting a status flag and invoking a trap handler (if it is enabled). We start by defining each exception, namely, we define the conditions that cause an exception to occur.

Assumptions: If one of the operands is infinite or not-a-number, then the overflow, underflow and inexact exceptions do not occur. Hence, we assume both operands represent finite values. Moreover, we assume that the exact result is also finite. (An infinite exact result generates a division by zero exception, an invalid exception, or does not signal any exception).

[1] The only exception to this statement is when a conversion cannot convert a floating-point number to an integer or decimal format due to overflow, infinity, or NaN.

4.1 Preliminary Definitions

Defining exceptions requires two additional definitions. The first definition deals with normalization shifting when there is no lower limit $e_{\min}$ on the exponent range.

Definition 8 *A normalization shift with unlimited exponent range is the mapping* ext_fac *from factorings that represent non-zero real numbers to factorings which satisfies:*

1. *Value is preserved:* $val(\text{ext_fac}(s, e, f)) = \text{val}(s, e, f)$.
2. *Let* $(s, \widehat{en}, \widehat{fn})$ *denote* $\text{ext_fac}(s, e, f)$. *Then* $\widehat{fn} \in [1, 2)$.

Note that $\widehat{en} = \lfloor \log_2(2^e \cdot f) \rfloor$ and that $\widehat{fn} = 2^{e-\widehat{en}} \cdot f$.

The second definition deals with rounding when the exponent range is unlimited (both from below and from above).

Definition 9 *The rounding with an unbounded exponent range is the mapping* ext_r *of non-zero reals into factorings defined by:*

$$\text{ext_}r(s, e, f) \triangleq \text{post_norm}((s, \widehat{en}, \text{sig_rnd}(\widehat{fn}))), \text{ where } (s, \widehat{en}, \widehat{fn}) = \text{ext_fac}(s, e, f)$$

Notation: Throughout this section we use the following notation:

- x denotes the exact (finite) value of the result of a given operation.
- (s, en, fn) is a normalized factoring of x (i.e. $(s, en, fn) = norm_fac(s, en, fn)$ and $x = val(s, en, fn)$).
- $(s, \widehat{en}, \widehat{fn})$ is a normalization shift with an unlimited exponent range of (s, en, fn) (i.e. $(s, \widehat{en}, \widehat{fn}) = ext_fac(s, en, fn)$).

We often abuse notation and refer to the rounding with an unlimited exponent range of a real number x, and denote it by $ext_r(x)$. One should regard $ext_r(x)$ as shorthand for $ext_r(s, \widehat{en}, \widehat{fn})$.

4.2 Overflow

Overflow is defined by the IEEE Standard as follows.

Definition 10 *Let x denote the exact finite result, and $x_{\max}$ the largest representable number (i.e. $x_{\max} = 2^{e_{\max}} \cdot (2 - 2^{-(p-1)})$). An overflow exception occurs if*

$$\text{abs}(\text{val}(\text{ext_}r(x))) > x_{\max}$$

Note that the definition of overflow is with respect to rounding with an unbounded exponent range. The importance of this is emphasized by the following example. In round toward 0 mode, exponent rounding causes values greater than or equal to $2^{1+e_{\max}}$ to be rounded down to $x_{\max}$. So if regular rounding were used in the definition of the overflow exception, then the overflow exception would not occur in round toward 0 mode at all.

The following claim facilitates the detection of overflow, since we do not compute $ext_r(x)$.

Claim 1 *Let (s, en, fn) denote the normalized factoring of the exact result. An overflow exception occurs iff $2^{en} \cdot \text{sig_rnd}(fn) > x_{\max}$ or, equivalently, iff $en > e_{\max}$ or ($en = e_{\max}$ and $\text{sig_rnd}(fn) = 2$).*

Proof: Observe that $(s, en, fn) = (s, \widehat{en}, \widehat{fn})$ if $abs(x)$ is large enough, which is the case with overflow. Since $val(ext_r(x)) = 2^{\widehat{en}} \cdot sig_rnd(\widehat{fn})$, the claim follows. □

Notation. We refer to the case that $en > e_{\max}$ as *overflow before rounding*, and sometimes use the equivalent condition that $2^{en} \cdot fn \geq 2^{e_{\max}+1}$. We refer to the case that $(en = e_{\max})$ and $(sig_rnd(fn) = 2)$ as *overflow after rounding.*

4.3 Underflow

The definition of underflow in the Standard is extremely complicated. In the definitions, we follow the language of the Standard using our terminology and notation. Informally, underflow occurs when two conditions occur: (a)*tininess* - The magnitude of the result is below $2^{e_{\min}}$; and (b) *loss-of-accuracy* - Accuracy is lost when the tiny result is represented with a denormalized significand. The Standard gives two definitions for each of these conditions, and thus, the Standard gives four definitions of underflow, each of which conforms with the Standard. Note that the same definition must be used by an implementation for all operations. However, the Standard does not state that the same definition should be used for all precisions.

The two definitions of tininess, *tiny-after-rounding* and *tiny-before-rounding*, are defined below.

Definition 11 *Let x denote the exact result.* tiny-before-rounding *occurs if*

$$0 < \text{abs}(x) < 2^{e_{\min}}$$

tiny-after-rounding *occurs if $x \neq 0$ and* [2]

$$0 < \text{abs}(\text{val}(\text{ext_}r(x))) < 2^{e_{\min}}$$

In round-to-nearest (even) mode, *tiny-after-rounding* occurs iff $0 < abs(x) < 2^{e_{\min}} \cdot (1 - 2^{-(p+1)})$. In general, *tiny-after-rounding* implies *tiny-before-rounding.* [3]

The two definitions of loss of accuracy are defined below.

Definition 12 loss-of-accuracy-a *(also called denormalization loss) occurs if $x \neq 0$ and*

$$r(x) \neq \text{ext_}r(x)$$

loss-of-accuracy-b *(also called inexact result) occurs if*

$$\text{val}(r(x)) \neq x$$

Assume that tininess (of either kind) occurs. *loss-of-accuracy-b* simply means that significand rounding introduces an error. The meaning of *loss-of-accuracy-a* is more complicated. If $fn = \widehat{fn}$ (namely, $fn \in [1, 2)$), then $r(x) = ext_r(x)$, and *loss-of-accuracy-a* does not occur. However, when fn is denormalized, then $en = e_{\min}$ and the binary representation of fn has $en - \widehat{en}$ leading zeros. Therefore, $sig_rnd(\widehat{fn})$ is in general different from $2^{en-\widehat{en}} \cdot sig_rnd(fn)$, because extra nonzero digits are "shifted in", and *loss-of-accuracy-a* may occur. Note that *loss-of-accuracy-a* implies *loss-of-accuracy-b.*

The definition of underflow depends on whether the underflow trap handler is enabled or disabled.

[2] Note that $ext_r(x)$ is not defined if $x = 0$.

[3] Note that $0 < 2^{en} \cdot sig_rnd(fn) < 2^{e_{\min}}$ in round-to-nearest (even) mode is equivalent to $2^{e_{\min}} \cdot 2^{-p} < abs(x) < 2^{e_{\min}} \cdot (1 - 2^{-p})$.

Definition 13 *If the underflow trap handler is disabled, then underflow occurs if both tininess and loss-of-accuracy occur. If the underflow trap handler is enabled, then underflow occurs if tininess occurs.*

Note that tininess in Def. 13 means either *tiny-after-rounding* or *tiny-before-rounding*. Similarly, loss-of-accuracy means either *loss-of-accuracy-a* or *loss-of-accuracy-b*.

4.4 Trapped Overflows and Underflows

The Standard sets special rules for the delivered result when an overflow or an underflow exception occurs and the corresponding trap handler is enabled. We call these cases *trapped overflows* and *trapped underflows*, respectively. In these cases, the exponent is "wrapped" so that the delivered result is brought back into the range of representable normalized numbers. We formalize the Standard's requirements below for trapped overflows and underflows for all operations except conversions. In Appendix B we describe the way trap handlers are invoked according to the Standard.

Definition 14 *Exponent wrapping in trapped overflow and underflow exceptions:*

1. *If a trapped overflow occurs, then the delivered result is* $r(x \cdot 2^{-\alpha})$, *where* $\alpha = 3 \cdot 2^{n-2}$.
2. *If a trapped underflow occurs, then the delivered result is* $r(x \cdot 2^{\alpha})$.

The difficulty that this "wrapped exponent" requirement causes is that we do not have the exact result x, since representing the exact result may require too many bits. We show that, in case of a trapped overflow, $r(x \cdot 2^{-\alpha})$ can be easily computed from $(s, en, sig_rnd(fn))$, and that in case of a trapped underflow, $r(x \cdot 2^{\alpha})$ can be easily computed from $(s, \widehat{en}, sig_rnd(\widehat{fn}))$. Hence, one can easily integrate the computation of the result when a trapped overflow or underflow occurs with the computation of rounding.

How big can a result be when an overflow occurs? Note, that the absolute value of a result of addition, subtraction, multiplication, division, or square root is less than $2^{e_{\max}+2}$. Moreover, α is approximately $1.5 \cdot e_{\max}$. This implies that when overflow occurs, then $x \cdot 2^{-\alpha}$ is well within the range of representable numbers with a normalized significand. A similar argument applies to the delivered result when a trapped underflow exception occurs.

The following claim shows that that a trapped overflow can be easily handled by subtracting α from the exponent.

Claim 2 *Let* $(s, en', fn') = \text{post_norm}(s, en, \text{sig_rnd}(fn))$. *If a trapped overflow occurs, then*

$$r(2^{-\alpha} \cdot x) = (s, en' - \alpha, fn')$$

Proof: Note that for all the operations described in the IEEE Standard, if a trapped overflow occurs, then $abs(2^{-\alpha} \cdot x)$ is approximately $2^{-e_{\max}/2}$, and hence, $abs(2^{-\alpha} \cdot x) \in (2^{e_{\min}}, 2^{e_{\max}})$. This implies that exponent rounding preserves value in this case. Since $(s, en - \alpha, fn)$ is a normalized factoring of $x \cdot 2^{-\alpha}$ we conclude that:

$$\begin{aligned} r(s, en - \alpha, fn) &= exp_rnd(post_norm(s, en - \alpha, sig_rnd(fn))) \\ &= post_norm(s, en - \alpha, sig_rnd(fn)) \\ &= (s, en' - \alpha, fn') \end{aligned}$$

and the claim follows. □

The analogous claim for the case of underflow does not hold. The reason being that $(s, en + \alpha, fn)$ is not the normalized factoring of $2^\alpha \cdot x$ when an underflow exception occurs. Recall, that fn is denormalized, and hence, its binary representation has leading zeros. However, the normalized factoring of $2^\alpha \cdot x$ has a normalized significand. Therefore, multiplying x by 2^α effects both the exponent and the significand in the normalized factoring.

Dealing with a trapped underflow exception is simple in addition and subtraction operations since the rounded result is exact (see Sec. 7.3). Therefore, $val(r(2^\alpha \cdot x)) = val(s, e + \alpha, f)$, and hence, one only needs to normalize f and update the exponent by adding α minus the number of leading zeros in f.

Dealing with a trapped underflow exception in other operations (such as multiplication) relies on the availability of normalized factoring with an unlimited exponent range $(s, \widehat{en}, \widehat{fn})$ representing the value x. Note that since underflow occurs, it follows that $\widehat{en} < e_{\min}$. We formalize an analogous claim for trapped underflow exceptions using the factoring $(s, \widehat{en}, \widehat{fn})$. The proof follows the proof of Claim 2.

Claim 3 *Let* $(s, en', fn') = \text{post_norm}(s, \widehat{en}, \text{sig_rnd}(\widehat{fn}))$. *If a trapped underflow occurs, then*

$$r(2^\alpha \cdot x) = (s, en' + \alpha, fn')$$

4.5 Inexact

The inexact exception occurs when the rounded result differs from the exact result.

Definition 15 *Let* x *denote the exact result. An* inexact exception *occurs if* $\text{val}(r(x)) \neq x$.

Note that the definition of *loss-of-accuracy-b* and the occurrence of an inexact exception are identical.

The Standard says that an inexact exception occurs "if the rounded result of an operation is not exact or it overflows without an overflow trap". The second part of this statement is redundant since an overflow always implies that the rounded result is not exact.

The precedence of trap handlers, namely, the fact that the overflow and underflow trap handlers have precedence over the inexact trap handler, seems to imply that the occurrence of an inexact exception need not be signaled when a trapped overflow or trapped underflow occurs. However, this is not explicitly said in the Standard. We choose the following interpretation which we believe provides the most information about loss of accuracy: If a trapped overflow or underflow exception occurs, then an inexact exception is signaled iff significand rounding introduced a loss of accuracy. Hence, the occurrence of a trapped underflow or overflow does not always imply an inexact exception, even thought the delivered result is $r(x \cdot 2^{\pm\alpha})$ and $val(r(x)) \neq x$. Note that our interpretation leads in Claim 4 to a wording which is similar to that of the Standard.

There are two possible sources of loss of accuracy in the computation of $r(x)$: significand rounding and exponent rounding. Note that a loss of accuracy introduced by significand rounding cannot be "corrected" exponent rounding. The SIG_INEXACT signal is raised when significand rounding introduces a loss of accuracy. Note that a trapped underflow causes a complication in this definition.

Definition 16 *The binary signal* SIG_INEXACT *is defined as follows: If a trapped underflow does not occur, then*

$$\text{SIG_INEXACT} = 1 \quad \textit{iff} \quad \text{sig_rnd}(fn) \neq fn$$

If a trapped underflow occurs, then

$$\text{SIG_INEXACT} = 1 \quad \textit{iff} \quad \text{sig_rnd}(\widehat{fn}) \neq \widehat{fn}$$

According to our interpretation, exponent rounding causes a loss of accuracy which should signal an inexact exception only if an untrapped overflow exception occurs. Therefore, an inexact exception is detected as follows:

Claim 4 *An inexact exception occurs iff* SIG_INEXACT $= 1$ *or an untrapped overflow exception occurs.*

5 Sticky-bits As Representatives of Equivalence Classes

Consider the task of significand rounding: the input consists of a significand f which might have very many bits to the left of the binary point (e.g. $e_{\max} - e_{\min}$ bits in floating-point addition), and the output has only $p-1$ bits to the left of the binary point. Obviously, it is not feasible to hold the exact significand, and hence f is substituted with a representative f' of f so that: (a) $sig_rnd(f) = sig_rnd(f')$; and (b)f' has only $p+1$ bits to the left of the binary point. Thus, hardware cost and delay are saved without introducing an error.

An important issue that we address is how to detect the occurrence of overflow, underflow, and inexact exceptions correctly even though the "exact" significand is substituted with a representative. We show that if the representative is close enough to the exact result, then the occurrence of exceptions can be inferred from the representative as if it were the exact result.

5.1 Sticky-bits

The common way of "shortening" a significand f whose binary representation requires more than $p+1$ bits to the right of the binary point is called a *sticky-bit computation*. In a sticky-bit computation, one replaces the tail of the binary representation of f starting in position $p+1$ to the right of the binary point with the OR of the bits in this tail.

Definition 17 *Let f denote a significand and let* $\text{bin}(f) = f_0.f_1f_2\ldots$ *denote the binary representation of f, namely, $f = \sum_i f_i \cdot 2^{-i}$. Define:*

$$\begin{aligned} \text{sticky_bit}(f,p) &\triangleq \text{OR}(f_{p+1}, f_{p+2}, \ldots) \\ \text{sticky}(f,p) &\triangleq \sum_{i \leq p} f_i \cdot 2^{-i} + \text{sticky_bit}(f,p) \cdot 2^{-p-1} \end{aligned}$$

The following claim follows from the definition of significand rounding.

Claim 5 *Let* $f' = \text{sticky}(f,p)$.

$$\begin{aligned} \text{sig_rnd}(f) &= \text{sig_rnd}(f') \\ \text{sig_rnd}(f) = f \quad &\textit{iff} \quad \text{sig_rnd}(f') = f' \\ \text{sig_rnd}(f') = f' \quad &\textit{iff} \quad f' = q \cdot 2^{-(p-1)} \textit{ for an integer } q \end{aligned}$$

Claim 5 shows that one can substitute f with $f' = \mathit{sticky}(f, p)$ for the purpose of computing $\mathit{sig_rnd}(f)$. The last two parts of the claim deal with detecting the occurrence of inexactness due to significand rounding, denoted by the signal SIG_INEXACT, which is used in the detection of the inexact exception (see Claim 4). The second part shows that SIG_INEXACT $= 1$ iff inexactness is introduced by significand rounding of f'. Hence, significand rounding does not distinguish at all between f and f'. The third part shows that SIG_INEXACT $= 1$ iff f' is not an integral multiple of $2^{-(p-1)}$. This implies that SIG_INEXACT can be computed as follows: Since f' can be represented with $p+1$ bits to the left of the binary point, SIG_INEXACT equals the OR of the two least significant bits of the binary representation of f'.

5.2 Equivalence Classes

Our goal is to define a condition for two numbers (or factorings) to have the same rounding (see Claim 7). Moreover, we would like the two factoring to cause the same exceptions. For this purpose we consider a partitioning of the real numbers into equivalence classes so that two numbers that belong to the same equivalence class have the same rounding, regardless of the rounding mode.

Definition 18 *Suppose α is an integer. Two real numbers x_1 and x_2 are α-equivalent if there exists an integer q such that $x_1, x_2 \in (q2^{-\alpha}, q2^{-\alpha} + 2^{-\alpha})$ or $x_1 = x_2 = q2^{-\alpha}$.*

The binary representations of α-equivalent reals must agree in the first α positions to the right of the binary point. Note that in case a number has two binary representations, we choose the finite representation (for example, 0.1 rather than $0.0111\ldots$).

One can easily verify that $\mathit{sticky}(f, \alpha)$ is a representative of the α-equivalence class of f, as summarized by the following claim.

Claim 6 *Let f denote a significand and α denote a positive integer. Then, f and* $\mathrm{sticky}(f, \alpha)$ *are α-equivalent.*

Note that the least significant bit of the binary representation of $\mathit{sticky}(f, \alpha)$ can be regarded as a flag indicating whether the corresponding equivalence class is a single point (i.e. $q2^{-\alpha}$) or an open interval (i.e. $(q2^{-\alpha}, (q+1)2^{-\alpha})$).

The following claim shows a simple and sufficient condition for two reals to have the same rounded factoring. We heavily rely on this claim for gluing the rounding unit with the adder.

Claim 7 *Let (s_x, e_x, f_x) denote a normalized factoring of x, and let (s_y, e_y, f_y) denote a normalized factoring of y. If x and y are $(p - e_x)$-equivalent, then:*

1. *$s_x = s_y$, $e_x = e_y$, and f_x and f_y are p-equivalent; and*
2. *$r(x) = r(y)$.*

Proof: The assumption that x and y are $(p - e_x)$-equivalent means that either $x = y$ or $x, y \in I$, where $I = (q \cdot 2^{e_x - p}, (q+1) \cdot 2^{e_x - p})$, for some integer q. If $x = y$ then the claim is obvious. If $x \neq y$, then since the interval I contains either only positive numbers or only negative numbers, it follows that $s_x = s_y$. For simplicity, we henceforth assume that $x, y > 0$.

Moreover, the interval I either consists only of denormalized values or of normalized values. The reason is that $2^{e_{\min}}$ can not belong to the interval I. Since both factorings are normalized, it follows that either $f_x, f_y \in [0, 1)$ or $f_x, f_y \in [1, 2)$.

1. If $f_x, f_y \in [0,1)$, then it follows that $e_x = e_y = e_{\min}$.

2. If $f_x, f_y \in [1,2)$, let us assume by contradiction that $e_x > e_y$. This would imply that

$$y = 2^{e_y} f_y < 2^{1+e_y} \leq 2^{e_x} \leq 2^{e_x} f_x = x$$

   Since $x, y \in I$, it follows that $2^{e_x} \in I$. But 2^{e_x} is an integral multiple of $2^{e_x - p}$, and therefore, the interval I cannot contain 2^{e_x}, a contradiction. Therefore, $e_x = e_y$.

Since $e_x = e_y$, it follows that $f_x, f_y \in (q2^{-p}, (q+1)2^{-p})$, and f_x and f_y are p-equivalent, as required in Part (1).

Since f_x and f_y are p-equivalent, it follows that $sticky(f_x, p) = sticky(f_y, p)$. Apply Claim 5, and Part (2) follows. □

5.3 Detecting Exceptions With Representatives

Consider an operation the exact result of which is x, and suppose that the computed result which is fed to the rounding unit equals y. How close should y be to x so that one can signal correctly whether the occurrence of an overflow, underflow, or inexact exception?

The following discussion assumes that x and y satisfy the premises of Claim 7. The correct detection of an inexact exception (or *loss-of-accuracy-b* for an untrapped underflow) requires a strengthening of the assumption, as described below.

Overflow. According to Claim 1, an overflow exception occurs iff $2^{e_x} \cdot sig_rnd(f_x) \geq 2^{e_{\max}+1}$. The following claim shows that the occurrence of an overflow can be inferred from y, as if y were the exact result.

Claim 8

$$2^{e_x} \cdot \text{sig_rnd}(f_x) \geq 2^{e_{\max}+1} \quad \textit{iff} \quad 2^{e_y} \cdot \text{sig_rnd}(f_y) \geq 2^{e_{\max}+1}$$

Proof: Since the premises of Claim 7 is satisfied, $e_x = e_y$ and f_x and f_y are p-equivalent. By Claim 5 it follows that $sig_rnd(f_x) = sig_rnd(f_y)$, and hence,

$$2^{e_x} \cdot sig_rnd(f_x) = 2^{e_y} \cdot sig_rnd(f_y)$$

and the claim follows. □

Underflow (tininess). The occurrence of an underflow exceptions depends on two conditions: tininess and loss-of-accuracy. The following claim implies that *tiny-before-rounding* occurs for y iff it occurs for x. A similar proof holds for *tiny-after-rounding*, and in the next paragraph we deal with *loss-of-accuracy-a* which is equivalent to an inexact exception.

Claim 9

$$\text{abs}(x) \in (0, 2^{e_{\min}}) \quad \textit{iff} \quad \text{abs}(y) \in (0, 2^{e_{\min}})$$

Proof: From Claim 7, it follows that x and y have the same signs, so we assume for simplicity that they are both non-negative. Moreover, x and y are $(p - e_x)$-equivalent. The $(p - e_x)$-equivalence classes in the neighborhood of 0 are: $\{0\}, (0, 2^{e_{\min}-p}), \ldots$. Similarly, the equivalence classes in the neighborhood of $2^{e_{\min}}$ are: $(2^{e_{\min}} - 2^{e_{\min}-p}, 2^{e_{\min}}), \{2^{e_{\min}}\}, (2^{e_{\min}}, 2^{e_{\min}} + 2^{e_{\min}-p})$. Since x and y belong to the same equivalence class, the claim follows. □

Inexact. Suppose a trapped underflow exception does not occur. If x and y satisfy the premises of Claim 7, then Claims 5 and 7 imply that the inexact exception is detected correctly even if we use y rather than x. If a trapped underflow exception occurs, then x and y must be $(p-\widehat{e_x})$-equivalent, where $\widehat{e_x} = \lfloor \log_2(2^{e_x} \cdot f_x) \rfloor$. Based on Claims 4 and 5, we can detect an inexact exception as follows:

Claim 10 *Let x denote the exact result. Suppose that x and y are $(p - \widehat{e_x})$-equivalent, where $\widehat{e_x} = \lfloor \log_2 \mathrm{abs}(x) \rfloor$. Let (s_y, e_y, f_y) denote a normalized factoring of y, and let $(s_y, \widehat{e_y}, \widehat{f_y}) = \mathrm{ext_fac}(s_y, e_y, f_y)$.*

1. *If a trapped underflow exception occurs, then* SIG_INEXACT $= 1$ *iff* $\mathrm{sticky}(\widehat{f_y}, p)$ *is not an integral multiple of* $2^{-(p-1)}$.
2. *If a trapped underflow exception does not occur, then* SIG_INEXACT $= 1$ *iff* $\mathrm{sticky}(f_y, p)$ *is not an integral multiple of* $2^{-(p-1)}$.

5.4 The Significand-Rounding Decision

Given the representative $sticky(fn, p)$, significand rounding reduces to a decision whether the p most significand bits of the binary representation of $sticky(fn, p)$ should be incremented or not. The decision depends on: the three least significant bits of the binary representation of $sticky(fn, p)$, the sign bit, and the rounding mode. The three least significant bits of the binary representation of $sticky(fn, p)$ are called in the literature: the LSB (which is $p-1$ positions to the right of the binary point), the round-bit, and the sticky-bit.

5.5 Block Diagram of Rounding Unit

In this section we describe a block-diagram of a rounding unit. Obviously other more efficient implementations are possible and have been implemented. The advantages of our design are: (a) Well defined requirements from the inputs to facilitate the usage of the rounding unit for different operations. (b) Correct and full detection of exceptions. (c) Correct handling of trapped underflow and overflow exceptions. Whenever no confusion is caused, we refer to a representation of a number (a significand, exponent, etc.) as the number itself. Fig. 2 depicts a block diagram of a rounding unit. Note that intermediate exponents and significands are denoted by $e^n, e^1, e^2, e^3, f^n, f^1, f^2, f^3$.
Input. The inputs consist of:

1. a factoring (s_{in}, e_{in}, f_{in});
2. the rounding mode; and
3. flags called UNF_EN and OVF_EN indicating whether the underflow trap and overflow trap handlers respectively are enabled.

Assumptions. We assume that the inputs satisfy the following assumptions:

1. Loosely speaking, $val(s_{in}, e_{in}, f_{in})$ is close enough to x. Formally, let x denote the exact result to be rounded and let $\widehat{en} = \lfloor \log_2 abs(x) \rfloor$. We assume that $val(s_{in}, e_{in}, f_{in})$ and x are $(p-\widehat{en})$-equivalent. In the correctness proof we show that the rounding unit computes $r(s_{in}, e_{in}, f_{in})$ and that this value also equals $r(x)$. Moreover, the rounding unit handles trapped exceptions correctly even though $val(s_{in}, e_{in}, f_{in})$ may not be equal to x.

2. $abs(x)$ is not too large or too small so that wrapping the exponent when a trapped underflow or overflow occurs produces a number within the range of numbers representable with normalized significands. Namely,
$$2^{e_{\min}-\alpha} < abs(x) < 2^{e_{\max}+\alpha} \tag{2}$$

3. If $x = 0$, then the sign-bit s_{in} is the sign-bit that should be output according to the IEEE Standard. Hence, the sign-bit of the rounded result equals s_{in}.

Output. The rounding unit outputs a factoring $(s_{in}, e_{out}, f_{out})$ that equals:

$$(s_{in}, e_{out}, f_{out}) = \begin{cases} r(x) & \text{if no trapped overflow or underflow occurs} \\ r(x \cdot 2^{-\alpha}) & \text{if a trapped overflow occurs} \\ r(x \cdot 2^{\alpha}) & \text{if a trapped underflow occurs} \end{cases} \tag{3}$$

According to Claims 2, 3, and 7, based on the first two assumptions, this specification can be reformulated as follows:

$$(s_{in}, e_{out}, f_{out}) = \begin{cases} r(s_{in}, e_{in}, f_{in}) & \text{if no trapped overflow or underflow occurs} \\ r(s_{in}, e_{in} - \alpha, f_{in}) & \text{if a trapped overflow occurs} \\ r(s_{in}, e_{in} + \alpha, f_{in}) & \text{if a trapped underflow occurs} \end{cases} \tag{4}$$

Additional flags are output: OVERFLOW, SIG_INEXACT, and TINY.

1. The OVERFLOW flag signals whether an overflow exception occurred.

2. The SIG_INEXACT signals whether significand rounding introduces a rounding error as defined in Def. 16. According to Claim 4 an inexact exception is detected as follows:
$$\text{INEXACT} = \text{SIG_INEXACT OR (OVERFLOW AND } not(\text{OVF_EN}))$$

3. The TINY flag signals whether *tiny-before-rounding* occurred. We detect an underflow exception using the definitions of *tiny-before-rounding* and *loss-of-accuracy-b* as follows:
$$\text{UNDERFLOW} = \textit{tiny-before-rounding} \text{ AND } (\textit{loss-of-accuracy-b} \text{ OR } not(\text{UNF_EN}))$$

Functionality. We describe the functionality of each block in Fig.2:

1. The normalization shift box deals primarily with computing the normalization shift of (s_{in}, e_{in}, f_{in}). However, it also deals with trapped underflows, overflow before rounding, and trapped overflows before rounding. Defining the functionality of the normalization shift box requires the following notation: Let $(s_{in}, en, fn) = norm_fac(s_{in}, e_{in}, f_{in})$ and let $(s_{in}, \widehat{en}, \widehat{fn}) = ext_fac(s_{in}, f_{in}, e_{in})$. The outputs of the normalization shift box are:

   - TINY - a flag indicating *tiny-before-rounding*, namely:
   $$\text{TINY} = 1 \quad \text{if} \quad 0 < 2^{e_{in}} \cdot f_{in} < 2^{e_{\min}}$$

- OVF_1 - a flag indicating overflow before rounding, namely:

$$\text{OVF}_1 = 1 \quad \text{if} \quad 2^{e_{in}} \cdot f_{in} \geq 2^{e_{\max}+1}$$

- an exponent e^n and a significand f^n defined by

$$(e^n, f^n) = \begin{cases} (en - \alpha, fn) & \text{if OVF_EN and } \text{OVF}_1 \\ (\widehat{en} + \alpha, \widehat{fn}) & \text{if UNF_EN and TINY} \\ (en, fn) & \text{otherwise} \end{cases}$$

Note that when an untrapped overflow exception occurs (i.e. OVF_1 and $\overline{\text{OVF_EN}}$), the values of e^n and f^n are not important because e_{out} and f_{out} are fully determined in the exponent rounding box according to the rounding mode and the sign bit.

2. The $sticky(f^n, p)$ box computes $f^1 = sticky(f^n, p)$. Typically, a "massive" sticky-bit computation takes place before the rounding unit to reduce the number of bits required for representing f_{in}, and here, if the guard bit turns out to be unnecessary, it becomes the round bit and the two least-significant bits are ORed to generate the sticky-bit.

3. The significand rounding outputs $f^2 = sig_rnd(f^1)$ and two flags: SIG-OVF and SIG_INEXACT. The SIG-OVF flag signals the case that $f^2 = 2$, known as significand overflow. The SIG_INEXACT flag signals whether $f^2 \neq f^1$.

4. The post-normalization box uses the SIG-OVF flag as follows: If $f^2 = 2$, then: (a) The exponent needs to be incremented. This can be implemented by incrementing e^n while the sticky bit and the significand rounding are computed, and then selecting either the exponent e^n or the incremented exponent. (b) The significand needs to be replaced by 1. This can be obtained by ORing the two most-significant bits of the binary representation of f^2, which is depicted in the figure by an OR gate that is input these two most-significant bits. Note, that since $f^2 \in [0, 2]$, the binary representation of f^2 has two bits to the left of the binary point.

5. The exponent adjust box deals with two issues caused by significand overflow: (a) a significand overflow might cause the rounded result to overflow; and (b) a significand overflow might cause a denormalized significand f^n to be replaced with a normalized significand $f^3 = 1$. (We later show that case (b) does not occur in addition/subtraction.)

   Formally, the outputs OVF_2 (which signals an overflow after rounding) and e^3 are defined as follows:

$$\text{OVF}_2 = 1 \quad \text{if} \quad e^2 = e_{\max} + 1$$

$$e^3 = \begin{cases} e_{\max} + 1 - \alpha & \text{if OVF_EN and } \text{OVF}_2 \\ e_{\min} & \text{if } msb(f^3) = 1 \text{ and TINY and } \overline{\text{UNF_EN}} \\ e^2 & \text{otherwise} \end{cases}$$

   The explanation for cases in the definition of e^3 are as follows.

- The case that OVF_EN and OVF_2 corresponds to the case in which significand rounding caused a trapped overflow.
- The case that $msb(f^3) = 1$ and TINY corresponds to the case in which significand rounding causes a tiny result to have a normalized significand. Then e^3 is set to the "normalized" representation of $e_{\min}$. Note that this case deals with the representation of the exponent, namely, whether the representation reserved for denormalized significands, 0^n, should be used or the representation reserved for normalized significands, $0^{n-1} \cdot 1$.

Note that if an untrapped overflow exception occurs, then the value of e^3 is not important because it is overridden by the exponent rounding.

6. The exponent rounding box outputs an exponent and significand (e_{out}, f_{out}) as follows.

$$(e_{out}, f_{out}) = \begin{cases} (e^3, f^3) & \text{if } \overline{\text{OVERFLOW}} \text{ or OVF_EN} \\ (e_{max}, f_{max}) & \text{if OVERFLOW and } \overline{\text{OVF_EN}} \text{ and round to } x_{\max} \\ (e_\infty, f_\infty) & \text{if OVERFLOW and } \overline{\text{OVF_EN}} \text{ and round to } \infty \end{cases}$$

The decision of whether an overflowed result is rounded to $x_{\max}$ or to ∞ is determined by the rounding mode and the sign bit.

5.6 Correctness

In this section we prove that the proposed rounding unit functions correctly. Specifically, the correct rounded factoring is output, exceptions are detected correctly, and exponent wrapping is performed in the case of trapped underflow and overflow exceptions.

We first make the simplifying assumption that $x = val(s_{in}, e_{in}, f_{ain})$, and discard this assumption at the end of the proof.

Correctness of the output flags.

1. Overflow is detected in two stages according to Claim 1: OVF_1 signals an overflow before rounding and OVF_2 signals an overflow after rounding.
2. If $x = val(s_{in}, e_{in}, f_{in})$, the TINY flag signals *tiny-before-rounding* by definition.
3. The SIG_INEXACT flag is correct because $2^{e_{in}} \cdot f_{in}$ and $2^{e^n} \cdot f_1$ are $(p - \widehat{en})$-equivalent (see Claims 5 and 10).

Correctness of f_{out} and e_{out}. We consider four cases:

1. Overflow does not occur and the underflow trap handler is disabled, namely:

$$-x_{\max} \leq \hat{r}(x) \leq x_{\max} \text{ AND } not(\text{UNF_EN})$$

The normalization shift satisfies $(e^n, f^n) = (en, fn)$, and therefore, $(s_{in}, e^2, f^3) = post_norm(s_{in}, en, sig_rnd(fn))$. Exponent adjustment in this case does change the value, although it may modify the representation of e^2 from 0^n to $0^{n-1}1$. Therefore, $(s_{in}, e_{out}, f_{out}) = r(x)$.

2. Overflow does not occur and the underflow trap handler is enabled, namely:

$$-x_{\max} \leq \widehat{r}(x) \leq x_{\max} \text{ AND UNF_EN}$$

This case differs from the first case only if an underflow exception occurs. When an underflow occurs, it is trapped, and the output should satisfy $(s_{in}, e_{out}, f_{out}) = r(x \cdot 2^{\alpha})$. Normalization shift in this case satisfies $(e^n, f^n) = (\widehat{en} + \alpha, \widehat{fn})$, and the rest of the proof follows the first case.

3. Overflow occurs and the overflow trap handler is disabled, namely:

$$abs(\widehat{r}(x)) > x_{\max} \text{ AND } not(\text{OVF_EN})$$

Since overflow is detected properly, the exponent rounding box sets e_{out} and f_{out} according to the rounding mode and sign.

4. Overflow occurs and the overflow trap handler is enabled, namely:

$$abs(\widehat{r}(x)) > x_{\max} \text{ AND OVF_EN}$$

Since overflow is detected properly, exponent wrapping takes place either during normalization shift or during exponent adjustment. Either way, Claim 2 is satisfied.

Now, $y = val(s_{in}, e_{in}, f_{in})$ is not the exact result, it is only $(p - \widehat{en})$-equivalent to the exact result x. In Claims 8-10, we have already established that this suffices for the correct detection of overflow, *tiny-before-rounding*, and SIG_INEXACT. As for e_{out} and f_{out}, we consider two cases:

1. If a trapped underflow does not occur, then from Claim 7, it suffices that x and $val(s_{in}, e_{in}, f_{in})$ be $(p - en)$-equivalent. Since $en \geq \widehat{en}$, it follows from Claim 11 part 2, that x and $val(s_{in}, e_{in}, f_{in})$ are $(p - en)$-equivalent.

2. If a trapped underflow occurs, then the output should equal $r(x \cdot 2^{\alpha})$. Using Claim 7 again, with respect to $x \cdot 2^{\alpha}$ and $val(s_{in}, e_{in} + \alpha, f_{in})$, we conclude that $r(x \cdot 2^{\alpha}) = r(s_{in}, e_{in} + \alpha, f_{in})$. We already proved that when a trapped underflow occurs the output satisfies: $(s_{in}, e_{out}, f_{out}) = r(s_{in}, e_{in} + \alpha, f_{in})$, and the correctness of the rounding unit follows.

6 Gluing Rounding With Other Operations

In this section we discuss how the rounding unit is glued with an adder or a multiplier (or other functional units). We provide some mathematical tools that facilitate this task.

6.1 General Framework

How does one design a circuit that meets the Standard's requirement that *"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 the destination's format"*? Obviously, one does not perform exact computations because that would be too expensive and slow. Fig. 3 depicts how infinite precision calculation is avoided. The functional unit is typically

a fixed point unit, for example, an adder or a multiplier. Cost and delay is saved by minimizing the precision of the functional unit. Given two operands, rather than computing $op(a, b)$ with infinite precision, we pre-process the operands to bound the precision, and operate on a' and b'. The bounded precision result, denoted by $op(a', b')$, satisfies $r(op(a, b)) = r(op(a', b'))$, and thus, correctness is obtained. We guarantee that $r(op(a, b)) = r(op(a', b'))$ by using Claim 7, namely, by insuring that $op(a', b')$ and $op(a, b)$ are $(p-e)$-equivalent, where e denotes the normalized exponent of $op(a, b)$. (When a trapped underflow occurs, we need more precision, namely, $(p - \widehat{e})$-equivalence, where $\widehat{e} = \lfloor log_2 abs(op(a, b)) \rfloor$). This succinct condition greatly simplifies the task of correctly gluing together the functional unit and the rounding unit.

6.2 Mathematical Tools

The following simple claim is used in proving the correctness of rounding, addition, detection of exceptions, and handling of trapped overflows and underflows. It enables a symbolic treatment of the manipulations performed during various operations without having to deal with the peculiarities of representation.

Claim 11 *Let α and α' denote integers such that $\alpha' \leq \alpha$. Let x, y denote two α-equivalent real numbers. Then the following properties hold:*

1. $\text{sticky}(-x, \alpha) = -\text{sticky}(x, \alpha)$ *and hence, $-x$ and $-y$ are α-equivalent.*
2. *x and y are α'-equivalent.*
3. *For every integer i : $x2^{-i}$ and $y2^{-i}$ are $(\alpha + i)$-equivalent.*
4. *$x + q2^{-\alpha'}$ and $y + q2^{-\alpha'}$ are α-equivalent, for every integer q.*

Proof: Part 1 follows from the fact that negation and the computation of the sticky bit are commutative.

Part 2 follows from the fact that the lattice defined by integral multiples of $2^{-\alpha}$ is finer than that defined by integral multiples of $2^{-\alpha'}$.

If $x = y$, then part 3 is trivial. Otherwise, $x, y \in (q \cdot 2^{-\alpha}, (q+1) \cdot 2^{-\alpha})$. Therefore, $x2^{-i}, y2^{-i} \in (q \cdot 2^{-(\alpha+i)}, (q+1) \cdot 2^{-(\alpha+i)})$, as required.

Part 4 follows from the fact that adding the same multiple of $2^{-\alpha}$ to x and y keeps them within the same interval, the endpoints of which are consecutive multiples of $2^{-\alpha}$. □

Claim 11 should be interpreted as follows: (1) $sticky(-x, \alpha)$ can be computed by computing $sticky(x, \alpha)$ and multiplying it by $sign(x)$. (2) Having too many bits of precision does not disturb correct rounding. (3) A shift to the right increases the number of bits of precision beyond the binary point, whereas a shift to the left reduces the the number of bits of precision beyond the binary point. (4) Adding an integral multiple of $2^{-\alpha}$ to two reals whose binary representations agree up to bit position α beyond the binary point, does not create a disagreement in these bit positions.

7 Floating-Point Addition

In this section we present an addition algorithm and prove its correctness. By correctness we refer to delivering the correct result and detecting the exceptions. In particular, we provide a proof that

3 additional bits of precision suffice for correct rounding. We deal here only with adding finite valued factorings and ignore the Standard's requirements concerning the sign-bit when the exact result equals zero. The special cases, such as NAN's, infinite valued factorings, and the sign of zero results are handled by additional logic which we do not specify. Whenever no confusion is caused, we refer to a representation of a number (a significand, exponent, etc.) as the number itself.

7.1 Floating-Point Addition Algorithm

The input to the adder consists of two factorings (s_1, e_1, f_1) and (s_2, e_2, f_2) which are normalized factorings. The adder outputs the factoring (s', e_1, g'), and this factoring is rounded by the rounding unit which is described in Sec. 5.5. The algorithm is divided into three stages: Preprocessing, Adding, and Rounding as follows:

Preprocessing

1. Swap. Swap the operands, if necessary, so that $e_1 \geq e_2$. This step is computed by comparing the exponents and selecting the operands accordingly. For simplicity, we assume that $e_1 \geq e_2$.

2. Alignment Shift. Replaces f_2 with f_2', where $f_2' \stackrel{\triangle}{=} f_2 \cdot 2^{-\delta}$ and $\delta = \min\{e_1 - e_2, p+2\}$. This step is computed by shifting the binary representation of f_2 by δ positions to the right.

3. Compute Representative. Compute $f_2'' \stackrel{\triangle}{=} \mathit{sticky}(f_2', p+1)$. Note that the binary representation of f_2'' has $p+2$ bits to the right of the binary point (namely, exactly 3 bits of precision are added to the significands).

Functional Unit

Add Significands. Output the factoring is (s', e_1, g'), where s' and g' are the sign and magnitude of $(-1)^{s_1} \cdot f_1 + (-1)^{s_2} \cdot f_2''$. Note that g' has $p+2$ bits to the right of the binary point and that $g' \in [0, 4)$.

Rounding

The Rounding unit receives the factoring (s', e_1, g') and additional inputs as described in Sec. 5.5.

Note, that the alignment shift and the representation computation can be performed in parallel since the shift amount $\delta = e_1 - e_2$ can be computed during the swap stage. Moreover, the width of the shifter equals $p+2$, namely, the bits that are shifted beyond position $p+1$ to the right of the binary point are discarded by the shifter. These discarded bits participate in the sticky-bit computation.

7.2 Correctness

In this section we prove the correctness of the addition algorithm.

Let $x = \mathit{val}(s_1, e_1, f_1)$, $y = \mathit{val}(s_2, e_2, f_2)$, and $\mathit{ext_fac}(x+y) = (s, \widehat{e}, \widehat{f})$. Let $y'' = \mathit{val}(s_2, e_1, f_2'')$. Since the factoring (s', e_1, g') is obtained by adding factorings that represent x and y'', it follows that $\mathit{val}(s', e_1, g') = x + y''$. Based on the correctness of the rounding-unit, proving the correctness of the floating-point addition algorithm reduces to proving the following claim.

Claim 12 *$x + y$ and $x + y''$ are $(p - \widehat{e})$-equivalent.*

Proof: We consider two cases: (a) $e_1 - e_2 \leq 1$; and (b) $e_1 - e_2 \geq 2$. The first case captures the event that the alignment shift is by at most one position whereas normalization shift might be large. The second case (as we show later) captures the event that the normalization shift is by at most one position (either to the left or to the right) whereas the alignment shift is small.

If $e_1 - e_2 \leq 1$, then the alignment shift is small and no precision is lost at all. Formally, since f_2' is an integral multiple of $2^{-(p-1+e_1-e_2)}$, and since $p - 1 + e_1 - e_2 \leq p$, it follows that $f_2'' = f_2'$. Therefore, $y'' = y$ and the claim follows.

If $e_1 - e_2 \geq 2$, the following facts hold:

1. $e_1 - 1 \leq e \leq e_1 + 1$. Since $e_1 \geq e_2$ (after swapping), it follows that $e \leq e_1 + 1$. The lower bound on e follows from the fact that $abs(x + y) > 2^{e_1} - 2^{e_2+1}$.

2. $f_2'' = sticky(2^{-(e_1-e_2)} \cdot f_2, p+1)$. To prove this we only need to show that $2^{e_1-e_2} \cdot f_2$ and f_2' are $(p + 1)$-equivalent. We consider two cases: If $\delta = e_1 - e_2$, then $2^{e_1-e_2} \cdot f_2 = f_2'$. If $\delta = p + 2$, then $e_1 - e_2 \geq p + 2$. Combining with $0 \leq f_2 < 2$, it follows that both $2^{-(e_1-e_2)} \cdot f_2$ and f_2' are either zero or belong to the interval $(0, 2^{-(p+1)})$, and thus, they are $(p + 1)$-equivalent.

3. y and y'' are $(p + 1 - e_1)$-equivalent. This follows from the previous fact and from Claim 11 Parts 1 and 3.

4. $x + y$ and $x + y''$ are $(p + 1 - e_1)$-equivalent. This follows from the previous fact and from Claim 11 Part 4, since x is an integral multiple of $2^{-(p-1-e_1)}$.

5. $e = \widehat{e}$. Note, that $e_1 \geq e_2 + 2 \geq e_{\min} + 2$. The first fact shows that $e \geq e_1 - 1$, and hence, $e > e_{\min}$, and therefore, $\widehat{e} > e_{\min}$, and the fact follows.

The first fact implies that $p - e \leq p + 1 - e_1$, and therefore, the fourth fact implies that $x + y$ and $x + y''$ are $(p - e)$-equivalent. Since $e = \widehat{e}$, the claim follows. □

Note that we need all the 3 additional bits of precision only if $e = e_1 - 1$. In this case, the normalization shift shifts the significand by one position to the left, and only 2 additional bits of precision are left, without them correct rounding cannot be performed.

7.3 Remarks

In this section we point out properties specific to floating-point addition and subtraction.

The following claim deals with the underflow exception in the addition operation. In particular, it shows that all four definitions of the underflow exception are identical with respect to addition and that a trapped underflow never occurs.

Claim 13 *Suppose two finite valued representable numbers are added, then:*

1. *If* tiny-before-rounding *occurs, then the adder outputs the exact sum, namely, neither* loss-of-accuracy-a *nor* loss-of-accuracy-b *occur.*

2. tiny-after-rounding *occurs iff* tiny-before-rounding *occurs.*

Proof: Note that all the representable numbers are integral multiples of $2^{e_{\min}-(p-1)}$. Since the values of the inputs are representable numbers, it follows that the exact sum is also an integral multiple of $2^{e_{\min}-(p-1)}$. Let x denote the exact sum. If *tiny-before-rounding* occurs, then

$$2^{e_{\min}-(p-1)} \leq x \leq 2^{e_{\min}} \cdot (1 - 2^{-(p-1)}).$$

Since all integral multiples of $2^{e_{\min}-(p-1)}$ in the above interval are representable, it follows that the exact sum is representable, and hence, neither significand rounding nor exponent rounding will introduce a loss of accuracy, implying $x = r(x)$. Since the fixed points of $r(x)$ are a subset of the fixed points of $\widehat{r}(x)$, it follows that $\widehat{r}(x) = x$, and the claim follows. □
Therefore, during addition, if the underflow trap handler is disabled, then the underflow exception never occurs. If the underflow trap handler is enabled, the underflow exception occurs iff $0 < f^n < 1$ where f^n denotes the significand output by the normalization shift box in the rounding unit.

An interesting consequence of Claim 13 is that the exponent adjust box in the rounding unit can be simplified for rounding sums and differences. Since a denormalized result is always exact, significand rounding cannot cause a denormalized significand to become normalized (thanks to Marc Daumas for pointing this out).

8 Multiplication

Multiplication is straightforward. The input consists of two factorings (s_1, e_1, f_1) and (s_2, e_2, f_2) which are normalized factorings. The multiplier outputs the factoring (s', e', f'), and this factoring is rounded by the rounding unit which is described in Sec. 5.5. The outputs are defined by $s' = \text{XOR}(s_1, s_2)$, $e' = e_1 + e_2$, and $f' = f_1 \cdot f_2$. Note, that if $f_1, f_2 \geq 1$, then one can avoid having a double length significand f' by computing $f' = \mathit{sticky}(f_1 \cdot f_2, p)$.

9 Conclusions

In the first part of this paper (Sections 2–4) we have presented a mathematical description of the IEEE Standard for floating-point arithmetic. Our presentation follows the language of the Standard so that the reader can verify that the mathematical description matches the Standard. This rigorous description facilitates implementation and correctness proofs. For example, the definitions of exceptions are mathematical transcriptions of the definitions as they appear in the Standard. We find these definitions hard to follow and a straightforward implementation of exception detection according to these definitions is not easy. To simplify the task of exception detection, we have presented claims that reformulate *equivalently* the definitions of exceptions. To our belief, these claims present more natural definitions of exceptions that render the detection of exceptions easier, as we show in our rounding-unit.

In Section 5, we focus on the problem that, although the rounding unit is not fed the exact result, it still must produce an output as if it were fed the exact result. Most descriptions of rounding units are integrated with a functional unit, such as an adder or a multiplier [2, 12, 13, 11, 16], so this issue is usually not addressed. Section 5 plays an important role in the proof of the correctness of the rounding unit as well as in the formulation of the rule for gluing a functional unit to a rounding unit which is presented in Sec. 7.

In Section 6, a block diagram of a rounding unit is provided, accompanied with a correctness proof. We prove that our rounding unit not only rounds as specified by the Standard, but also detects exceptions correctly, and deals with trapped overflow and underflow exceptions. The issue of detecting exceptions is typically brushed aside and solved ad hoc by designers.

In Section 7, we discuss how a functional unit (e.g. an adder) is "glued" to a rounding unit. In general, glue conditions ensure correct functionality when connecting together components that were separated in early design stages. We present a succinct specification that the functional unit needs to satisfy so that feeding the rounding unit with the output of the functional unit will result with a correct result.

In Sections 8-9, we demonstrate how the rounding unit can be used in floating point addition and multiplication algorithms. In Section 8, we present a floating point addition algorithm and prove its correctness. The proof shows that feeding the rounding unit with the output of the adder results with a correct result, as specified by the Standard. Typical treatments of this issue in the literature fail to deal with all the "side-cases" involved with such a proof.

The mathematical notation that we have used is instrumental in achieving a precise and concise specification of the Standard. Our experience with students in the University of the Saarland at Saarbrücken and in Tel-Aviv University has shown us that, after this presentation, students with basic knowledge of hardware design are able to design IEEE compliant floating point units. We find this to be a smooth way to get started with researching optimization techniques in floating point units.

Acknowledgments

We would like to thank Marc Daumas, Arno Formella, John Hauser, David Matula, Silvia Müller, Asger Nielsen, and Thomas Walle for many helpful suggestions and discussions.

References

[1] J. T. Coonen, "Specification for a Proposed Standard for Floating Point Arithmetic", Memorandum ERL M78/72, University of California, Berkeley, 1978.

[2] G. Even, S.M. Mueller, and P.M. Seidel. A Dual Mode IEEE multiplier. In *Proceedings of the 2nd IEEE International Conference on Innovative Systems in Silicon*, pages 282–289. IEEE, 1997.

[3] David Goldberg, "Computer Arithmetic", Appendix A in *Computer Architecture: A Quantitative Approach*, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, 1990.

[4] James M. Feldman and Charles T. Retter, *Computer Architecture: a designer's text based on a generic RISC* McGraw-Hill, 1994.

[5] Israel Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

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

[7] David Matula, "Floating Point Representation", manuscript, May 1996.

[8] Amos R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice Hall, 1994.

[9] S. M. Müller and W. Paul, "The Decomposition Theorem for IEEE Floating Point Rounding", manuscript, available at: `http://www-wjp.CS.Uni-SB.DE/~smueller/publi.html`.

[10] S. M. Müller and W. Paul, "Complexity and Correctness of Computer Architectures". to appear by Springer.

[11] N. Quach, N. Takagi, and M. Flynn. On fast IEEE rounding. Technical Report CSL-TR-91-459, Stanford University, January 1991.

[12] M.R. Santoro, G. Bewick, and M.A. Horowitz. Rounding algorithms for IEEE multipliers. In *Proceedings 9th Symposium on Computer Arithmetic*, pages 176–183, 1989.

[13] Peter-M. Seidel, Guy Even, "How many logic levels does floating-point addition require ?", Proceedings of the 1998 IEEE International Conference on Computer Design (ICCD'98):VLSI in Computers & Processors, pp. 142-149, Oct., 1998.

[14] Uwe Sparmann, *Strukturbasierte Testmethoden für arithmetische Schaltkreise*, Ph.D. Dissertation, Universität des Saarlandes, 1991.

[15] Shlomo Waser and Michael J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Reiner & Winston, 1982.

[16] R.K. Yu and G.B. Zyner. 167 MHz Radix-4 floating point multiplier. *Proceedings 12th Symposium on Computer Arithmetic*, 12:149–154, 1995.

A Representation

Our abstraction level enables discussing the design of FPU's without dealing with the peculiarities of representation. However, for the sake of completeness, we briefly discuss how floating point numbers are represented according to the IEEE Standard.

The exponent is represented by an n-bit string, denoted by E. We consider three cases:

1. $\mathrm{E} \notin \{0^n, 1^n\}$: In this case the value represented by E is $bin(\mathrm{E}) - bias$, where $bin(\mathrm{E})$ is the binary number represented by E and $bias \stackrel{\triangle}{=} 2^{n-1} - 1$. This is a biased exponent representation.

2. $\mathrm{E} = 0^n$: The all zeros string is reserved for denormalized significands. The value represented by the 0^n exponent string is $e_{\min}$.

3. $\mathrm{E} = 1^n$: This case is used for representing infinity (e_∞) and "not a number" (NAN), depending on the significand string.

Note that $e_{\min}$ has two representations: $0^{n-1} \cdot 1$ for normalized significands and 0^n for denormalized significands. The decision of whether the significand's value is in $[0, 1)$ or in $[1, 2)$ is implied by the representation of the exponent. We elaborate on this decision below.

The significand is represented by a $(p-1)$-bit string: $f_1 f_2 \cdots f_{p-1}$. The value represented by the significand string depends on the exponent string E as follows:

1. If $\text{E} \notin \{0^n, 1^n\}$, then the significand is normalized. We set $f_0 = 1$, and the value represented by the significand string is $\sum_{i=0}^{p-1} f_i \cdot 2^{-i}$.

2. If $\text{E} = 0^n$, then the significand is denormalized. We set $f_0 = 0$, and the value represented by the significand string is $\sum_{i=0}^{p-1} f_i \cdot 2^{-i}$.

3. If $\text{E} = 1^n$, then the 0^{p-1} significand string represents the symbol f_∞. Thus, the symbol e_∞ is represented by 1^n and the symbol f_∞ is represented by 0^{p-1}. If the significand string is not 0^{p-1}, then the floating point number does not represent any value, and it is called "not a number" (NAN).

Note that the value of f_0 (if any) is implied by the exponent string. Since this bit does not appear explicitly in the floating point number, it is called the *hidden bit*. However, this bit appears usually in the internal format of floating point units.

Note that zero has a denormalized significand, and therefore, must be represented with an all zeros exponent and an all zeros significand. Since every floating point number has a sign bit, zero has two different representations, one is called $+0$ and the other is called -0.

B Trap Handling

In this section we describe the mechanisms defined by the IEEE standard for handling exceptions when the trap handler is disabled and when the trap handler is enabled.

Figure 4 depicts the components of this mechanism: the floating-point unit, the User and the trap handler, and two registers. We now describe the components.

1. The floating-point unit. This unit may realized in any combination of hardware and software.

2. The User. This is a person, software, or hardware that uses the FPU operations. For example, this may be a user's program or the operating system. The trap handler is part of the user, and is a program (either in software or hardware) that is invoked when a trapped exception occurs, as described shortly. The user may change the trap handler.

3. Trap handler enable/disable register. This 5-bit register has one bit per exception (there are five types of exceptions). The value stored in the register is written by the user, and the FPU can only read the value.

4. Status Flags register. This 5-bit register has one bit per exception. The individual bits of the status flag can be read and reset (i.e. set to zero) by the user. Moreover, the user can save and restore all five bits of the Status Flags register at one time. The FPU, on the other hand, sets the bits of the Status Flags register whenever an untrapped exception occurs (namely, when an exception occurs with the corresponding trap handler being disabled). From the point of view of the FPU, the bits of the Status Flags register are "sticky-bits". A possible way of implementing such a Status Flags register is as follows. After each operations, the FPU prepares a temporary status register in which only the bits corresponding to untrapped exceptions that occurred during the operation are set to one. The Status Flags register is then set to the bitwise OR of its previous value and the temporary status register.

Every operation is executed by the FPU as follows. At the beginning of the operations, the FPU reads the trap handler enable/disable register. When the operation is executed, the FPU detects the occurrence of the exceptions (which may be trapped or untrapped) Whenever a trapped exception occurs, the execution of the program which is currently being executed is suspended, and the trap handler is invoked. The trap handler is supplied with the following data:

1. Which exception occurred?
2. Which operation caused the exception?
3. What is the format of the destination (e.g., single precision, double precision)? This is required because some FPU's can perform operations in various formats. The trap handler must know the format in which the result is to be given.
4. In trapped overflow, underflow, and inexact exceptions, the trap handler is supplied with the correctly rounded result (with exponent wrapping), including information that might not fit in the destination's format.
5. In trapped invalid and division by zero exceptions, the trap handler is supplied with the operand values.

The trap handler returns a value to the FPU. This value is output by the FPU as the final result of the operation. Moreover, the trap handler determines whether the bits in the Status Flags register corresponding to trapped exceptions are set or reset.

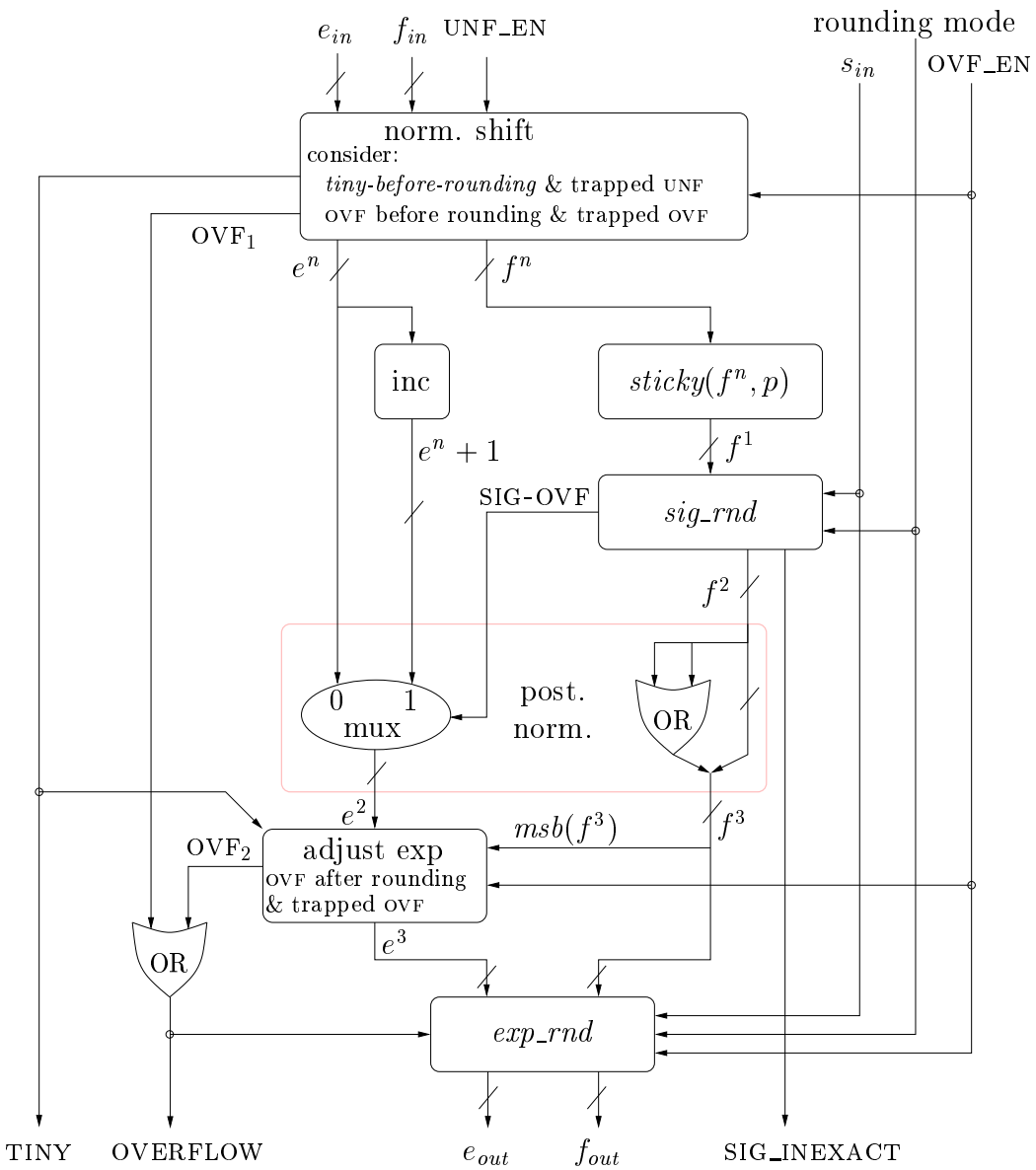

Figure 2: Rounding unit

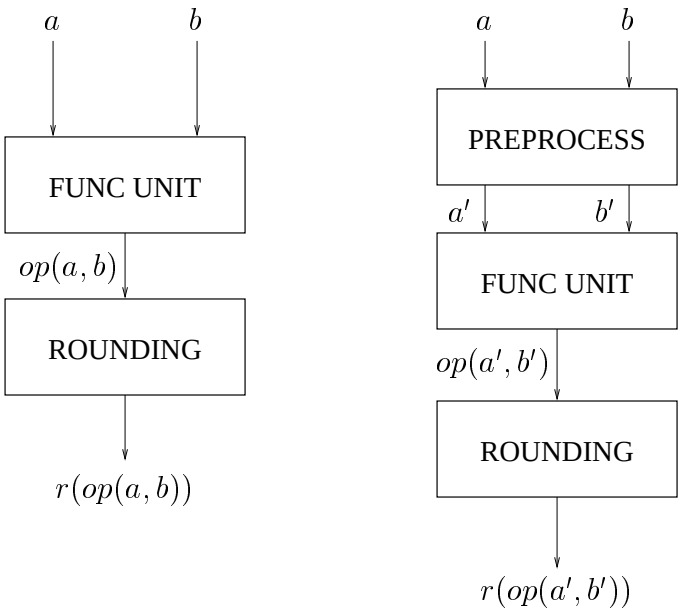

Figure 3: Avoiding Infinite Precision

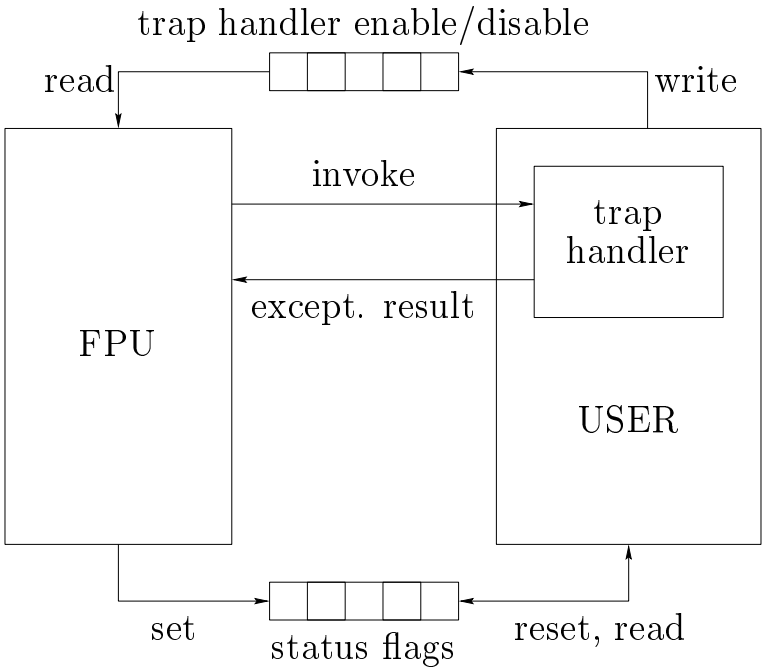

Figure 4: Trap Handling According to The IEEE Standard