On The Design of IEEE Compliant Floating Point Units *

Guy Even† and Wolfgang Paul
Univ. des Saarlandes
66123 Saarbrücken, Germany
E-mail:guy,wjp@cs.uni-sb.de

June 1, 1997

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 modular IEEE compliant Floating Point Unit. First, we provide a mathematical and hopefully unambiguous definition of the IEEE Standard which specifies the functionality. The design consists of: an adder, a multiplier, and a rounding unit, each of which is further partitioned. 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 arguments while still enabling addressing crucial implementation issues such as delay and cost.

1 Introduction

Background. The IEEE Standard for Floating Point Arithmetic [1] 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.

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 [2, 3, 5, 6, 7]. (b) Implementation reports that describe existing FPU's while emphasizing features that enable improved performance.

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

†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.

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.

Overview and Contributions. This paper deals systematically with the design of IEEE compliant FPU's. A key issue is the choice of an abstraction level. We use an abstraction level suggested by Matula [4] 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. Modularity is obtained by defining a succinct property that the output of an adder and the input to the rounding unit should satisfy so that correctness is guaranteed. A rounding unit design is presented together with a correctness proof that it detects exceptions correctly and deals correctly with trapped overflows and underflows. An addition algorithm is presented, and its correctness is discussed.

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, a rounding unit is presented and its correctness is proved. In Sec. 6, 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. 7, an high-level adder algorithm is presented and its correctness is proved. In Sec. 8, a multiplication algorithm is briefly discussed. In sec. 9, we conclude the paper. In Appendix A, the Standard's roles regarding representation are discussed. In Appendix B, the Standard's rules regarding trap handling are discussed.

2 IEEE Standard - Representable Numbers

In this section we review which numbers are representable according to the IEEE Standard. The abstraction level ignores representation and uses factorings as suggested by Matula [4]. 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 and describe their geometry.

Factorings

Every real number x can be factored into a sign factor (determined by a sign-bit), a scale factor (determined by an exponent), and a significand as follows:

$$x = (-1)^s \cdot 2^e \cdot f$$

The sign bit s is in $\{0, 1\}$, the exponent e is an integer, and the significand f is a non-negative real number. Usually the range of the significand is limited to the half open interval $[0, 2)$, but for example, intermediate results may have significands that are larger than 2. We henceforth refer to a triple (s, e, f) as a *factoring*, and $val(s, e, f)$ is defined to be $(-1)^s \cdot 2^e \cdot f$. A natural issue is that of unique representation. We postpone this issue until Sec. 3.1 in which we define a unique factoring called a *normalized factoring.*

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 $(1, e_\infty, f_\infty)$.

Standard's representable real numbers

In this section we define which numbers 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.

Every format (single, double, etc.) has two parameters attached to it: (a) n - the length of the exponent string; and (b) p - the length of the significand string (including the hidden bit).

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: $e_{\min} \triangleq 1 - bias.$

maximum exponent: $e_{\max} \triangleq 2^n - 2 - bias.$

where $bias \triangleq 2^{n-1} - 1$. This range of exponent values results from the biased binary representation of exponents.

The set of representable significand values is the set of all integral multiples of $2^{-(p-1)}$ in the half open interval $[0, 2)$. We distinguish between two ranges: representable significand values in the half open interval $[0, 1)$ are called *denormalized*, and representable significand values in the half open interval $[1, 2)$ are called *normalized*.

Not all combinations of representable exponent values and significand values are representable. 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.

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

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:

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

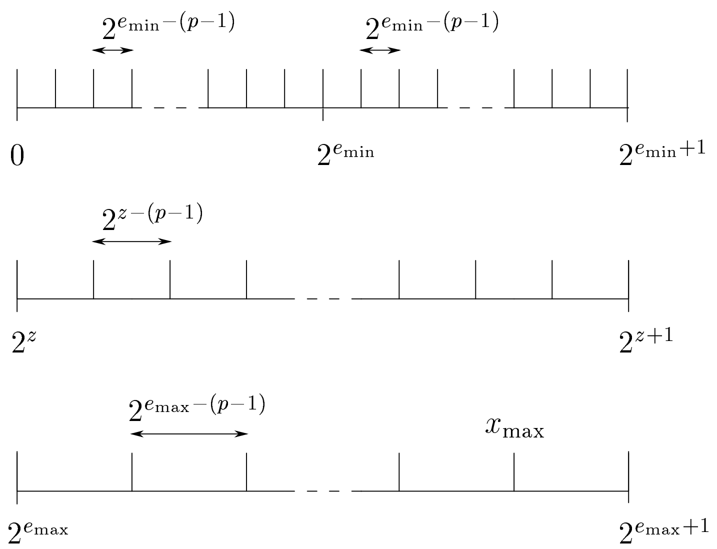

Figure 1: Geometry of Representable Numbers

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 in round-to-nearest (even) mode.

3.1 Normalized Factoring

Every real number has infinitely many factorings. We would like to obtain a unique factoring by requiring that $1 \leq f < 2$. However, there are two problems with this requirement. First, zero cannot be represented, and second, we do not want to have very small exponents. (We deal with large exponents later, during exponent-rounding). A normalized factoring is a unique factoring which avoids very small exponents and enables representing zero.

Definition 1 Normalization shift *is the mapping η from the reals to factorings which maps every real x into $\eta(x) = (s, e, f)$ so that $x = \mathrm{val}(\eta(x)) = \mathrm{val}(s, e, f)$, i.e. value is preserved, and the following conditions hold:*

1. *If $\mathrm{abs}(x) \geq 2^{e_{\min}}$, then $1 \leq f < 2$. In this case, we say that the significand f a normalized significand.*

2. *If $\mathrm{abs}(x) < 2^{e_{\min}}$, then $e = e_{\min}$ and $0 \leq f < 1$. In this case, we say that the significand f is a denormalized significand.*

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. When $x = \pm\infty$, the normalization shift of x is (s, e_∞, f_∞), where $s = sign(x)$, and e_∞, f_∞ denote the exponent and significand symbols used for factoring $\pm\infty$.

We also consider normalized factorings.

Definition 2 *A factoring (s, e, f) is the* normalized factoring *of (s', e', f') (respectively x) if it satisfies $(s, e, f) = \eta(\mathrm{val}(s', e', f'))$ (respectively $(s, e, f) = \eta(x)$). A factoring (s, e, f) is* normalized *if $(s, e, f) = \eta(\mathrm{val}(s, e, f))$.*

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 a unique factoring representing a real number. The disadvantage of this terminology is that a normalized factoring can have a denormalized significand.

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 $f < 2$). We deal with rounding in round-to-nearest (even) mode.

First, we sandwich f by two consecutive integral multiples of $2^{-(p-1)}$ as follows:

$$q \cdot 2^{-(p-1)} \leq f < (1 + q) \cdot 2^{-(p-1)}, \text{ where } q \text{ is an integer.} \tag{1}$$

Let q' be the even integer in $\{q, q + 1\}$. The significand rounding of f, denoted by $sig\_rnd(f)$ is defined as follows:

$$sig\_rnd(f) \triangleq \begin{cases} q \cdot 2^{-(p-1)} & \text{if } q \cdot 2^{-(p-1)} \leq f < (q + 0.5) \cdot 2^{-(p-1)} \\ q' \cdot 2^{-(p-1)} & \text{if } f = (q + 0.5) \cdot 2^{-(p-1)} \\ (q + 1) \cdot 2^{-(p-1)} & \text{if } (q + 0.5) \cdot 2^{-(p-1)} < f < (q + 1) \cdot 2^{-(p-1)} \end{cases} \tag{2}$$

Definition 3 *The signal* SIG_INEXACT *is defined as follows:*

$$\text{SIG\_INEXACT} = 1 \quad \textit{if} \quad \mathrm{sig\_rnd}(f) \neq f$$

The SIG_INEXACT signal is required for detecting the loss of accuracy.

Note that the rounding modes: round to $+\infty$ and round to $-\infty$, as defined in the 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, the significand is set to 1 and the exponent is incremented. This normalization shift takes place only when a significand overflow occurs. Therefore, post-normalization, denoted by $post\_norm(s, e, f)$, is defined as follows:

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

3.4 Exponent Rounding

Exponent rounding maps factorings into factorings and deals with the case that the absolute value of a factoring is too large. According to the Standard, exponent rounding is bypassed when trapped overflow exception occurs. We define only exponent rounding in round-to-nearest (even) mode.

Recall that e_∞ and f_∞ denote the exponent and significand symbols used for factoring $\pm\infty$. Let $x^*_{\max} = 2^{e_{\max}} \cdot (2 - 2^{-p})$. The exponent-rounding of (s, e, f) is defined as follows:

$$exp\_rnd(s, e, f) \triangleq \begin{cases} (s, e_\infty, f_\infty) & \text{if } 2^e \cdot f \geq x^*_{\max} \\ \eta(val(s, e, f)) & \text{otherwise} \end{cases}$$

The motivation for the definition of the threshold of $x^*_{\max}$ is as follows. Let $x_{\max}$ denote the largest representable number, namely, $x_{\max} = 2^{e_{\max}} \cdot (2 - 2^{-(p-1)})$. If we were not limited by $e_{\max}$, then the next representable number would be $2^{e_{\max}} \cdot 2$. The midpoint between these numbers is exactly $x^*_{\max}$. The binary representation of the significand of $x_{\max}$ is $1.11 \cdots 1$. Therefore, the significand is an odd integral multiple of $2^{-(p-1)}$. Hence, $x^*_{\max}$ would be rounded up in round-to-nearest (even) mode. This is why numbers greater than or equal to $x^*_{\max}$ are rounded to infinity. A similar argument holds for rounding of values not greater than $-x^*_{\max}$ to minus infinity.

Note that if $1 \leq f < 2$ is an integral multiple of $2^{-(p-1)}$, then we can simplify the definition of exponent rounding, and round to infinity if $e > e_{\max}$.

3.5 Rounding

Rounding is a mapping of reals into factorings. The rounding of x, denoted by $r(x)$, is defined as follows: Let (s, e, f) denote the normalized factoring of x, i.e. $(s, e, f) = \eta(x)$, then

$$r(x) \triangleq exp\_rnd\,(post\_norm\,(s, e, sig\_rnd(f))) \tag{3}$$

Therefore, 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 $r(x)$ is a normalized factoring.

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. 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).

4.1 Definitions

Defining exceptions requires two additional definitions.

Definition 4 *A normalization shift with unbounded exponent range is the mapping $\widehat{\eta}$ from the* non-zero *reals to factorings which maps every real x into a factoring $(s, \widehat{e}, \widehat{f})$ so that $x = \mathrm{val}(\widehat{\eta}(x))$, i.e. value is preserved, and $\widehat{f} \in [1, 2)$.*

In other words, $\widehat{\eta}(x)$ is the normalized factoring of $x \neq 0$ if there were no lower bound on the exponent range.

Definition 5 *The rounding with an unbounded exponent range is the mapping $\widehat{r}$ of non-zero real into factorings defined by:*

$$\widehat{r}(x) \triangleq \mathrm{post\_norm}((s, \widehat{e}, \mathrm{sig\_rnd}(\widehat{f}))), \;\; \textit{where } \widehat{\eta}(x) = (s, \widehat{e}, \widehat{f})$$

Notation: Throughout this section we use x to denote the exact (finite) value of the result of an operation. Let $\eta(x) = (s, e, f)$, and $\widehat{\eta}(x) = (s, \widehat{e}, \widehat{f})$.

Overflow. Informally, overflow occurs when the magnitude of the result is larger than $x_{\max}$. However, this is not precise, because significand rounding can lower the magnitude back into the range of representable numbers. The precise definition is given below.

Definition 6 *Let x denote the exact result, and $x_{\max}$ the largest representable number. An overflow exception occurs if*

$$\mathrm{val}(\widehat{r}(x)) > x_{\max} \quad \textit{or} \quad \mathrm{val}(\widehat{r}(x)) < -x_{\max}$$

Note that the definition of overflow is with respect to rounding with an unbounded exponent range.

The following claim facilitates the detection of overflow, since we do not compute $\widehat{r}(x)$.

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

Proof: Observe that $\eta(x) = \widehat{\eta}(x)$ if $abs(x)$ is large enough, which is the case with overflow. Since $\widehat{r}(x) = 2^{\widehat{e}} \cdot sig\_rnd(\widehat{f})$, the claim follows. $\square$

Underflow. The definition of underflow in the Standard is extremely complicated. In the definitions, we follow the language of the Standard, although some of the definitions include irrelevant cases. 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 7 tiny-after-rounding *occurs if* $x \neq 0$ *and*

$$0 < \text{abs}(\widehat{r}(x)) = 2^{\widehat{e}} \cdot \text{sig\_rnd}(\widehat{f}) < 2^{e_{\min}}$$

tiny-before-rounding *occurs if*

$$0 < \text{abs}(x) = 2^{e} \cdot f < 2^{e_{\min}}$$

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

The two definitions of loss of accuracy are defined below.

Definition 8 loss-of-accuracy-a *(also called denormalization loss) occurs if* $x \neq 0$ *and*

$$r(x) \neq \widehat{r}(x)$$

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

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

The Standard is unclear about whether loss-of-accuracy should also occur when overflow occurs. We have chosen to define loss-of-accuracy so that the two types of loss-of-accuracy occur when overflow occurs. However, we are interested in detecting loss-of-accuracy when tininess occurs, and not when overflow occurs. In this context, *loss-of-accuracy-b* simply means that significand rounding introduces an error. The meaning of *loss-of-accuracy-a* is more complicated: When f is normalized (namely, in the range $[1, 2)$), then $\eta(x) = \widehat{\eta}(x)$, and if no overflow occurs, then *loss-of-accuracy-a* does not occur. However, when f is denormalized, then $e = e_{\text{min}}$ and the binary representation of f has $e - \widehat{e}$ leading zeros. Therefore, $sig\_rnd(\widehat{f})$ is in general different from $2^{e-\widehat{e}} \cdot sig\_rnd(f)$, because extra nonzero digits are "shifted in". 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.

Definition 9 *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.*

[1] Note that $0 < 2^{e} \cdot sig\_rnd(f) < 2^{e_{\min}}$ in round-to-nearest (even) mode is equivalent to $2^{e_{\min}} \cdot 2^{-p} < abs(x) < 2^{e_{\min}} \cdot (1 - 2^{-p})$.

Note that tininess in Def. 9 means either *tiny-after-rounding* or *tiny-before-rounding*. Similarly, loss-of-accuracy means either *loss-of-accuracy-a* or *loss-of-accuracy-b*.

Inexact. The Standard defines the inexact exception twice, and the two definitions turn out to be equivalent. The first definition says that an inexact exception occurs when the value of the delivered result does not equal the exact result. This has already been defined in Def. 8 as *loss-of-accuracy-b*. The second definition is as follows.

Definition 10 *An inexact exception occurs if* $\text{val}(r(x)) \neq x$ *or if an overflow occurs with the trap handler disabled.*

Since overflow implies *loss-of-accuracy-b*, it follows that both definitions are equivalent.

The following claim facilitates the detection of an inexact exception.

Claim 2 $\text{val}(r(x)) = x$ *iff* $\text{sig\_rnd}(f) = f$ *and* $\text{exp\_rnd}(s, e, \text{sig\_rnd}(f)) = (s, e, \text{sig\_rnd}(f))$.

Proof: In the definition of $r(x)$ there are only two functions that do not preserve value. These are *sig_rnd*(f) and *exp_rnd*$(s, e,$ *sig_rnd*$(f))$. A change in value by one of these functions cannot be "corrected" by the other, and hence, the claim follows. □

Note that the overflow and underflow trap handlers have precedence over the inexact trap handler.

4.2 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. In Appendix B we describe the way trap handlers are invoked according to the Standard.

Definition 11 *Exponent wrapping in trapped overflow and underflow exceptions:*

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

The difficulty that this "wrapped exponent" requirement causes is that we do not have the exact result x, but only the factoring $(s, e,$ *sig_rnd*$(f))$. Our goal is to integrate these requirements 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 $4 \cdot 2^{e_{\max}}$. 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.

The following claim shows that that exponent overflow this can be easily done in the case of overflow.

Claim 3 *Let* $(s, e', f') = \text{post\_norm}(s, e, \text{sig\_rnd}(f))$*, where* $\eta(x) = (s, e, f)$*. If* $\text{abs}(\widehat{r}(x)) < 2^{e_{\max}+1+\alpha}$ *and a trapped overflow occurs, then*

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

Proof: The condition $abs(\widehat{r}(x)) < 2^{e_{\max}+1+\alpha}$ is satisfied for all the operations described in the IEEE Standard, and implies that $e_{\min} < e - \alpha < e_{\max} - 1$ and $\eta(2^{-\alpha} \cdot x) = (s, e - \alpha, f)$. Hence, $exp\_rnd(post\_norm(s, e-\alpha, sig\_rnd(f))) = post\_norm(s, e-\alpha, sig\_rnd(f))$, and the claim follows. □

The implication of Claim 3 is that when a trapped overflow occurs, the only fix required is to subtract α from the exponent.

The analogous claim for the case of underflow does not hold. The reason being that $(s, e+\alpha, f)$ is not the normalized factoring of $2^{\alpha} \cdot x$. Recall, that f 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 multiplication relies on the availability of $\widehat{\eta}(x)$. Consider the factoring $(s, \widehat{e}, \widehat{f}) = \widehat{\eta}(x)$. Note that since underflow occurs, it follows that $\widehat{e} < e_{\min}$. We formalize an analogous claim for trapped underflow exceptions using $\widehat{\eta}(x)$. The proof follows the proof of Claim 3.

Claim 4 *Let* $(s, \widehat{e}, \widehat{f}) = \widehat{\eta}(x)$*, and* $(s, e', f') = \text{post\_norm}(s, \widehat{e}, \text{sig\_rnd}(\widehat{f}))$*. If* $2^{e_{\min}-\alpha} \leq \text{abs}(\widehat{r}(x))$ *and a trapped underflow occurs, then*

$$r(2^{\alpha} \cdot x) = (s, e' + \alpha, f')$$

5 Rounding - Computation

In this section we deal with the issue of how to design a rounding unit. Our goal is to design a rounding unit that also detects the exceptions: overflow, underflow, and inexact. Moreover, the rounding unit wraps the exponent in case of a trapped overflow or underflow exception. Significand rounding is presented using representatives that require less bits of precision while guaranteeing correct rounding. We relate representatives with sticky bits, and present a block diagram of a rounding unit capable of detecting and dealing with exceptions.

5.1 Representatives

Definition 12 *Suppose α is an integer. Two real numbers x_1 and x_2 are α-equivalent, denoted by* $x_1 \stackrel{\alpha}{=} x_2$ *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...).

We choose α-representatives of the equivalence classes as follows.

Definition 13 *Let x denote a real number and α an integer. Let q denote the integer satisfying: $q2^{-\alpha} \leq x < (q+1)2^{-\alpha}$. The* α-representative *of x, denoted by $\text{rep}_\alpha(x)$, is defined as follows:*

$$\text{rep}_\alpha(x) \triangleq \begin{cases} q2^{-\alpha} & \textit{if } x = q2^{-\alpha} \\ (q+0.5)\cdot 2^{-\alpha} & \textit{if } x \in (q2^{-\alpha}, (q+1)2^{-\alpha}) \end{cases}$$

Note that for a fixed α, the set of all α-representatives equals the set of all integral multiples of $2^{-\alpha-1}$. Thus, the α-representatives have $\alpha+1$ bits of precision beyond the binary point. Moreover, the least significant bit of the binary representation of an α-representative can be regarded as a flag indicating whether the corresponding equivalence class is a single point or an open interval.

The following claim summarizes the properties of representatives that we use in significand rounding. The proof follows immediately from the definition of significand rounding and $rep_p(f)$.

Claim 5 *Let $f' = \text{rep}_p(f)$, then*

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

Claim 5 shows that one can substitute f with $rep_p(f)$ for the purpose of computing *sig_rnd*(f). Not only do they generate the same significand rounding, but they also generate the same SIG_INEXACT signal. Moreover, the SIG_INEXACT signal can be easily computed from $rep_p(f)$ because SIG_INEXACT $= 1$ iff $rep_p(f)$ is not an integral multiple of $2^{-(p-1)}$. Since $rep_p(f)$ is an integral multiple of $2^{-(p+1)}$, it follows that SIG_INEXACT is the OR of the two least significant bit of the binary representation of $rep_p(f)$.

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 6 *If $\eta(x) = (s_x, e_x, f_x)$, $\eta(y) = (s_y, e_y, f_y)$ and $x \stackrel{p-e_x}{=} y$, then (a) $s_x = s_y$, $e_x = e_y$ and $f_x \stackrel{p}{=} f_y$; and (b) $r(x) = r(y)$.*

Proof: The assumption that $x \stackrel{p-e_x}{=} y$ 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. In the second case, since the interval I cannot contain both negative and positive numbers, let us assume that $x > 0$, and hence $y > 0$ as well.

Note also that 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 $f_x, f_y \in [1,2)$. If $f_x, f_y \in [0,1)$, then it follows that $e_x = e_y = e_{\min}$. Therefore, $f_x, f_y \in (q2^{-p}, (q+1)2^{-p})$, and $f_x \stackrel{p}{=} f_y$, as required in Part (a).

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

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

But the interval I cannot contain 2^{e_x}, a contradiction. Therefore, $e_x = e_y$, and as before, this implies $f_x \stackrel{p}{=} f_y$, as required.

Part (b) follows from Claim 5, and the claim follows. □

Consider an exact result x and a computed result y. Claims 5 and 6 imply that the inexact exception is detected correctly even if we use y rather than x.

Based on Claims 2 and Claim 5, we can detect an inexact exception as follows:

Corollary 7 *An inexact exception occurs iff an untrapped overflow exception occurs or* $\text{rep}_p(f)$ *is not an integral multiple of* $2^{-(p-1)}$.

5.2 Sticky-bits and Representatives

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. We show that the value of the binary string obtained by this substitution equals the p-representative of f. Thus our use of representatives captures the common practice of using a sticky-bit.

Claim 8 *Let* f *denote a significand and let* $\text{bin}(f) = f_0.f_1 f_2 \ldots$ *denote the binary representation of* f*, namely,* $f = \sum_i f_i \cdot 2^{-i}$*. Define:*

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

Then,

$$\text{rep}_p(f) = \text{sticky}(f,p)$$

Proof: The binary representation $rep_p(f)$ is identical to the binary representation of f up to position p to the right of the binary point. If f is an integral multiple of 2^{-p}, then $f = rep_p(f)$ and the rest of the binary representation of f is all zeros, and so is $sticky\_bit(f,p)$. If f is not an integral multiple of 2^{-p}, then the rest of the binary representation of f is not all zeros, and therefore, $sticky\_bit(f,p) = 1$, and $rep_p(f) = sticky(f,p)$, as required. □

5.3 The Significand-Rounding Decision

Given the representative of $f \in [0,2)$, significand rounding reduces to a decision whether the p most significand bits of the binary representation of $rep_p(f)$ should be incremented or not. The decision depends on: the three least significant bits of the binary representation of $rep_p(f)$, the sign bit, and the rounding mode.

The three least significant bits of the binary representation of $rep_p(f)$ are called: the LSB (which is $p-1$ positions to the right of the binary point), the round-bit, and the sticky-bit. Using the notation used in defining significand rounding: the LSB is needed to determine the parity of q, whereas the round-bit and sticky-bit determine which of the three cases is selected.

5.4 Design - Block Diagram

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.

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 two assumptions:

1. Let x denote the exact result. For example, x is the exact sum of two values that are added by the adder and rounded by the rounding unit. Let $(s, \widehat{e}_x, \widehat{f}_x) = \widehat{\eta}(x)$. Then, we assume that

$$val(s_{in}, e_{in}, f_{in}) \stackrel{p-\widehat{e}_x}{=} x \tag{4}$$

2. $abs(x)$ is not too large or 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,

$$2^{e_{\min}-\alpha} \leq abs(\widehat{r}(x)) < 2^{e_{\max}+1+\alpha} \tag{5}$$

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

$$(s_{in}, e_{out}, f_{out}) = \begin{cases} r(x) & \text{if no trapped overflow or underflow occurs} \\ r(x \cdot 2^{-\alpha}) & \text{if a trapped overflow occurs} \\ r(x \cdot 2^{\alpha}) & \text{if a trapped underflow occurs} \end{cases} \tag{6}$$

According to Claims 3, 4, and 6, based on the assumptions stated in Eq. 4 and 5, this specification can be reformulated as follows:

$$(s_{in}, e_{out}, f_{out}) = \begin{cases} r(val(s_{in}, e_{in}, f_{in})) & \text{if no trapped overflow or underflow occurs} \\ r(val(s_{in}, e_{in} - \alpha, f_{in})) & \text{if a trapped overflow occurs} \\ r(val(s_{in}, e_{in} + \alpha, f_{in})) & \text{if a trapped underflow occurs} \end{cases} \tag{7}$$

Additional flags are output: OVERFLOW, TINY and SIG_INEXACT.

1. The OVERFLOW flag signals whether an overflow exception occurred.
2. The TINY flag signals whether *tiny-before-rounding* occurred.
3. The SIG_INEXACT signals whether significand rounding introduces a rounding error as defined in Def. 3. SIG_INEXACT is used for detecting an inexact exception occurred (according to Coro. 7, inexact occurs iff SIG_INEXACT OR (OVERFLOW AND $\overline{\text{OVF\_EN}}$)).

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

1. The normalization shift box deals primarily with computing $\eta(val(s_{in}, e_{in}, f_{in}))$. However, it also deals with trapped underflows and partially deals with trapped overflows. Defining the functionality of the normalization shift box requires some notation: Let $(s_{in}, e'_{in}, f'_{in}) = \eta(val(s_{in}, e_{in}, f_{in}))$ and let $(s_{in}, \widehat{e}_{in}, \widehat{f}_{in}) = \widehat{\eta}(val(s_{in}, f_{in}, e_{in}))$. Then $\text{TINY}, \text{OVF}_1, e^n, f^n$ are defined as follows:

$$\text{TINY} = 1 \quad \text{if} \quad 0 < 2^{e_{in}} \cdot f_{in} < 2^{e_{\min}}$$

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

$$(e^n, f^n) = \begin{cases} (e'_{in} - \alpha, f'_{in}) & \text{if OVF\_EN and OVF}_1 \\ (\widehat{e}_{in} + \alpha, \widehat{f}_{in}) & \text{if UNF\_EN and TINY} \\ (e'_{in}, f'_{in}) & \text{otherwise} \end{cases}$$

Note that when 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 $rep_p()$ box computes $f^1 = rep_p(f^n)$, meaning that the final sticky-bit is computed. Typically, a "massive" sticky-bit computation takes place before the rounding unit, and here, only the round-bit and sticky-bit are ORed, if the guard-bit turns out to be the LSB.

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 the exponent needs to be incremented, and 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 and e^3 are defined as follows:

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

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

   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.

5.5 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 in the case of trapped underflow and overflow exceptions is performed.

We assume first that $x = val(s_{in}, e_{in}, f_{in})$, namely that the value of the input equals the exact result. We then show that the proof still holds even if $val(s_{in}, e_{in}, f_{in})$ is only $(p - \widehat{e})$-equivalent to the exact result x, where $\widehat{e}$ denotes the exponent in $\widehat{\eta}(x)$.

The first case we consider is: no overflow and underflow trap handler disabled, namely, $-x_{\max} \leq \widehat{r}(x) \leq x_{\max}$ and $\overline{\text{UNF\_EN}}$. The normalization shift satisfies $(s_{in}, e^n, f^n) = \eta(x)$. Also, $f^1 = rep_p(f^n)$ and $f^2 = sig\_rnd(f^1)$. Post-normalization guarantees that $(s_{in}, e^2, f^3) = \eta(val(s_{in}, e^n, sig\_rnd(f^n)))$. Exponent adjustment in this case does not modify the value of e^2, but may modify its representation from 0^n to $0^{n-1}1$. Both overflow flags, OVF_1 and OVF_2, are set to zero. Therefore, $(s_{in}, e_{out}, f_{out}) = r(x)$.

The detection of the inexact exception is based on Coro. 7 and Claim 5. Recall that f^n equals the significand of $\eta(x)$, and SIG_INEXACT $= 1$ iff $sig\_rnd(rep_p(f^n)) \neq rep_p(f^n)$. Therefore, an inexact exception occurs iff SIG_INEXACT or (OVERFLOW AND $\overline{\text{OVF\_EN}}$).

The TINY flag in this case signals whether tiny-before-rounding occurred. Therefore, an underflow exception occurs iff TINY and an inexact exception occurs.

The second case we consider is $-x_{\max} \leq \widehat{r}(x) \leq x_{\max}$ and UNF_EN. The difference between this case and the first case is when a trapped underflow occurs, namely, $0 < abs(x) < 2^{e_{\min}}$. In this case $val(s_{in}, e_{out}, f_{out})$ should equal $r(x \cdot 2^{\alpha})$. Since $2^{e_{\min}-\alpha} \leq abs(\widehat{r}(x))$ and $x = val(s_{in}, e_{in}, f_{in})$, it follows that $\eta(x \cdot 2^{\alpha}) = (s_{in}, \widehat{e}_{in} + \alpha, \widehat{f}_{in})$. Therefore, normalization shift functions properly, and the rest of the proof for this case follows the first case.

The third case we consider is $abs(\widehat{r}(x)) > x_{\max}$ and $\overline{\text{OVF\_EN}}$. The detection of overflow is based on Claim 1: $\text{OVF}_1 = 1$ if $abs(x) \geq 2^{e_{\max}+1}$, whereas $\text{OVF}_2 = 1$ if $e^n = e_{\max}$ and $sig\_rnd(f^n) = 2$. Since OVERFLOW $= 1$ and $\overline{\text{OVF\_EN}}$, exponent rounding overrides previous computations, and e_{out}, f_{out} are determined by the rounding mode and sign.

The fourth case we consider is $abs(\widehat{r}(x)) > x_{\max}$ and OVF_EN. As in the third case, overflow is detected, however, exponent wrapping takes place either during normalization shift or during exponent adjustment. Either way, Claim 3 is satisfied.

Now, $val(s_{in}, e_{in}, f_{in})$ is not the exact result, it is only $(p - \widehat{e})$-equivalent to the exact result. Unless a trapped underflow occurs, Claim 6 implies that for the purpose of computing $r(x)$ and detecting exceptions, using $val(s_{in}, e_{in}, f_{in})$ yields the same results. In other words, the output of the rounding unit does not distinguish between two inputs that are $(p - e)$-equivalent. In case a trapped underflow occurs, the same holds because $val(s_{in}, e_{in} + \alpha, f_{in})$ and $x \cdot 2^{\alpha}$ are $p - (\widehat{e} + \alpha)$-equivalent (see part 3 of Claim 9). This completes all the possible cases, and thus, 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 6, namely, by insuring that $op(a', b')$ and $op(a, b)$ are $(p - e)$-equivalent. (When a trapped underflow occurs, we need more precision, namely, $op(a', b') \overset{p-\widehat{e}}{=} op(a, b)$, where $\widehat{e}$ is the exponent in $\widehat{\eta}(x)$). This succinct condition greatly simplifies the task of correctly gluing together the functional unit and the rounding unit.

6.2 Mathematical Tools

The following 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 9 *Let x, y denote real numbers such that $x \overset{\alpha}{=} y$, for an integer α. Let $\alpha' \leq \alpha$ denote an integer. Then the following properties hold:*

1. $\mathrm{rep}_{\alpha}(-x) = -\mathrm{rep}_{\alpha}(x)$, *and hence,* $-x \overset{\alpha}{=} -y$.

2. $x \overset{\alpha'}{=} y$.

3. *For every integer* i : $x2^{-i} \overset{\alpha+i}{=} y2^{-i}$.

4. $x + q2^{-\alpha'} \overset{\alpha}{=} y + q2^{-\alpha'}$, *for every integer* q.

Proof: If x is an integral multiple of $2^{-\alpha}$, then $rep_\alpha(x) = x$. If x is not an integral multiple of $2^{-\alpha}$, then $rep_{-\alpha}(x)$ equals the midpoint between the two integral multiples of $2^{-\alpha}$ that sandwich x. Note that $-x$ is an integral multiple of $2^{-\alpha}$ iff x is, and that the absolute value of the multiplies of $2^{-\alpha}$ that sandwich x are equal to those that sandwich $-x$. Therefore, Part 1 follows.

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 9 should be interpreted as follows: (1) $rep_\alpha(x)$ can be computed by computing $rep_\alpha(abs(x))$ and multiplying it by $sign(x)$. (2) Recall that an α-representative has $\alpha + 1$ bits of precision beyond the binary point. Thus, having too many bits of precision does not disturb correct rounding. (3) If two reals have identical binary representation up to bit position α beyond the binary point, then after division by 2^i (shift right by i positions) they have identical binary representations up to bit position $\alpha + i$. Conversely, a shift to the left reduces the 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 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 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. 5.4. 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 $f_2' \triangleq f_2 \cdot 2^{-\delta}$ 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 $f_2'' \triangleq rep_{p+1}(f_2')$. This computation is, in effect, a sticky-bit computation as described in Claim 8. 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 2 bits to the left of the binary point.

Rounding

The Rounding unit receives the factoring (s', e_1, g') and additional inputs as described in Sec. 5.4.

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

Claim 10 $x + y \overset{p-\widehat{e}}{=} x + y''$

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 $e_1 - e_2 \geq 2$, then we first show that the normalization shift is small. Recall that $\eta(x + y) = (s, e, f)$. We show that $e \geq e_1 - 1$ (obviously, $e \leq e_1 + 1$) as follows: Since the factorings (s_1, e_1, f_1) and (s_2, e_2, f_2) are normalized, it follows that $abs(x + y) > 2^{e_1} - 2^{e_2+1}$. The assumption is that $e_2 \leq e_1 - 2$, and hence, $abs(x + y) > 2^{e_1-1}$, and thus, $e \geq e_1 - 1$.

Next, we show that

$$f_2'' = rep_{p+1}(2^{-(e_1-e_2)} \cdot f_2) \tag{8}$$

Since $\delta = \min\{e_1 - e_2, p+2\}$, we consider two cases: If $\delta = e_1 - e_2$, then Eq. 8 follows from the definition of 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 $2^{-\delta} \cdot f_2$ belong to the interval $[0, 2^{-(p+1)})$, and thus, they have the same $(p+1)$-representative, and Eq. 8 follows.

Since $y = (-1)^{s_2} \cdot 2^{e_1} \cdot 2^{-(e_1-e_2)} \cdot f_2$ and $y'' = (-1)^{s_2} \cdot 2^{e_1} \cdot f_2''$, it follows from Claim 9 part 3 that $y \stackrel{p+1-e_1}{=} y''$. Since x is an integral multiple of $2^{-(p-1-e_1)}$, it follows from Claim 9 part 4 that $x+y \stackrel{p+1-e_1}{=} x+y''$. Since $e \geq e_1 - 1$, it follows that $p - e \leq p+1-e_1$, and from Claim 9 part 2 it follows that $x+y \stackrel{p-e}{=} x+y''$. The claim follows because if $e_1 \geq e_2 + 2$, then $e \geq e_1 - 1 \geq e_{\min} + 1$, which implies that $e = \widehat{e}$. □

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 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 11 *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, and $\eta(x) = (s, e, f)$. If *tiny-before-rounding* occurs, then

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

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, no loss of accuracy occurs. Since $r(x) = x$, it also 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 11 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. 5.4. 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 $f' = rep_p(f_1 \cdot f_2)$.

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 and then reformulates it to obtain simplicity and ease of implementation. 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, moreover, implementing these definitions is tricky. Thus, an implementation has to be validated to make sure that its detection of exceptions conforms with the definitions of the Standard. To simplify this problem, 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 the second part of the paper (Sec. 5), we have presented a rounding unit 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 the third part of the paper (Sec. 6), 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 the fourth part of the paper (Sec. 7), we have proved the correctness of a floating point addition algorithm. 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 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] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985.

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

[3] Isreal Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, 1993.

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

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

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

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

A Representation

Our abstraction level enables discussing the design of FPU's without dealing with the pecularities 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. $\text{E} \notin \{0^n, 1^n\}$: In this case the value represented by E is $bin(\text{E}) - bias$, where $bin(\text{E})$ is the binary number represented by E. This is a biased exponent representation.

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

3. $\text{E} = 1^n$: 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 occured 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 occurance 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 occured?

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.

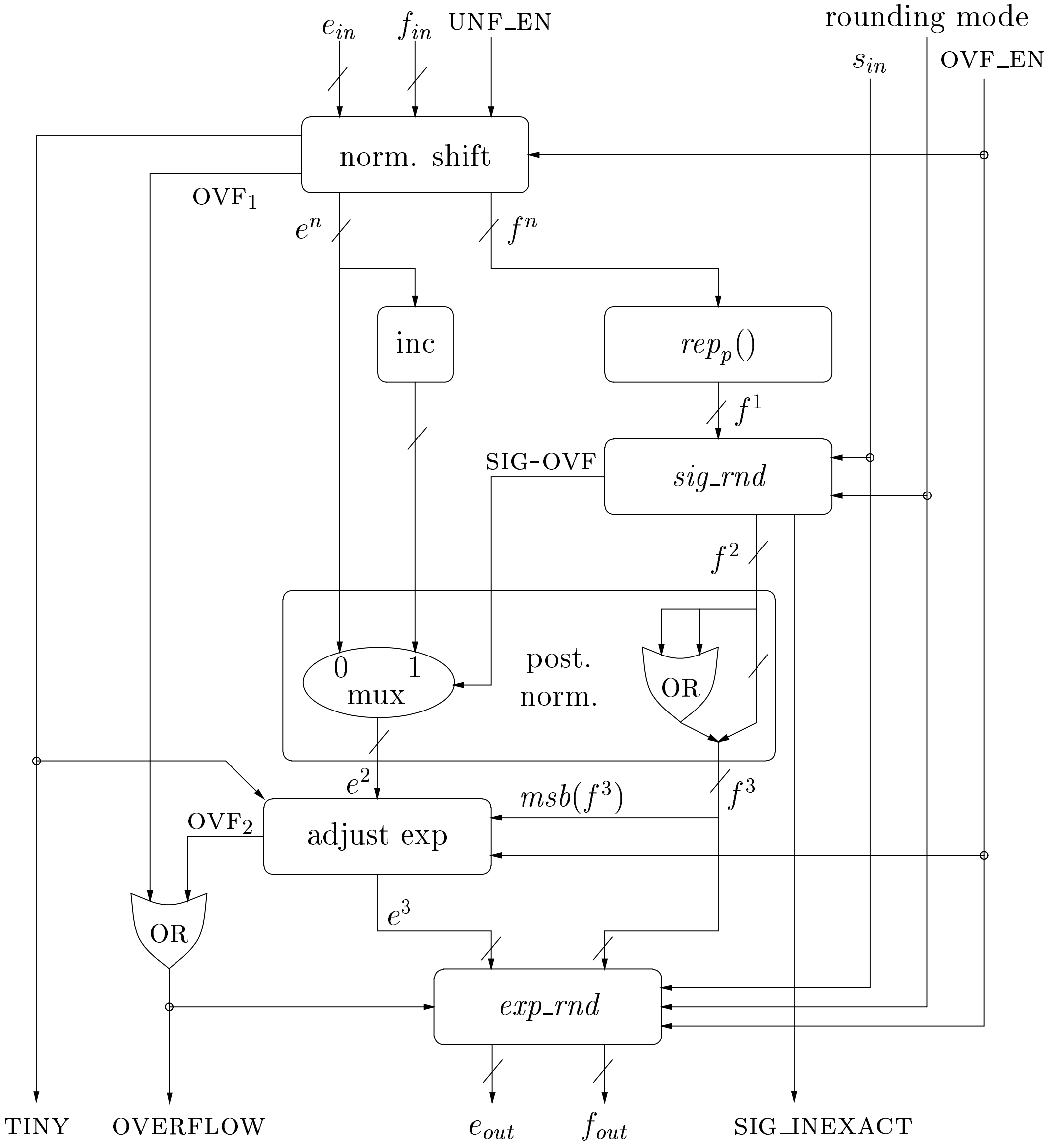

Figure 2: Rounding Unit

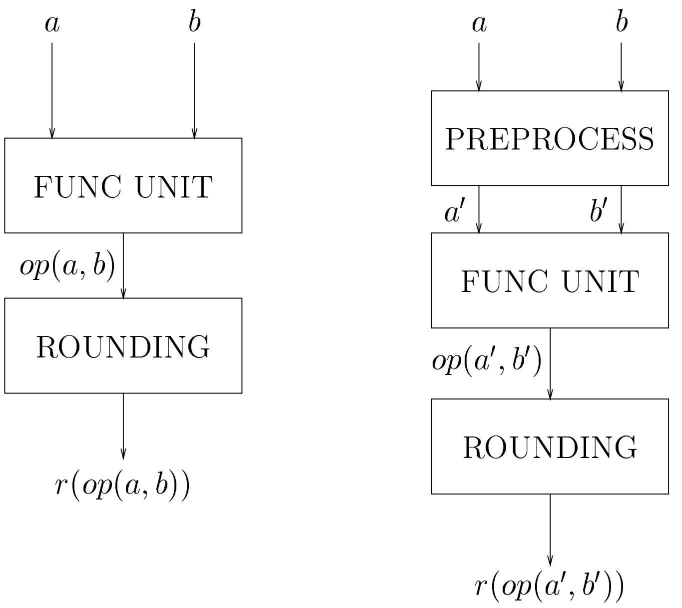

Figure 3: Avoiding Infinite Precision

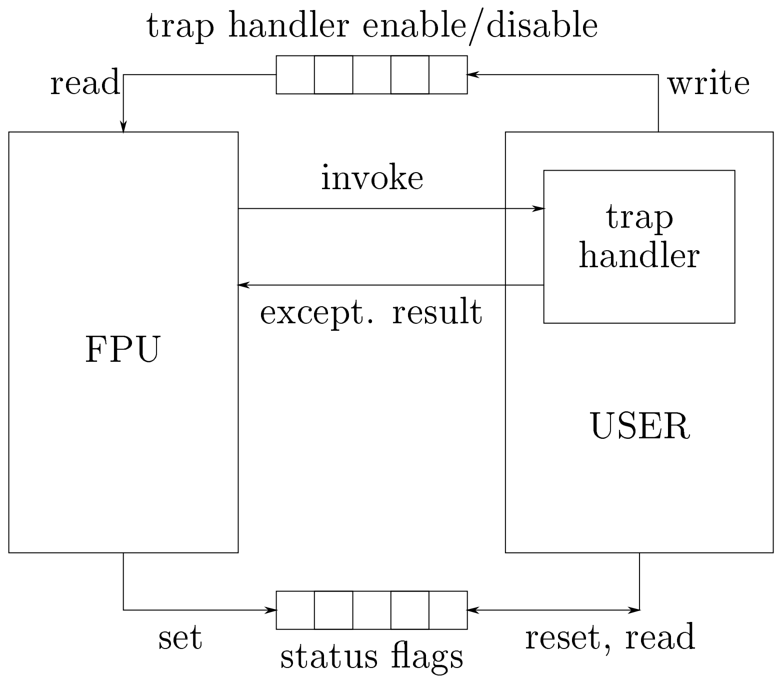

Figure 4: Trap Handling According to The IEEE Standard