A machine-checked theory of floating point arithmetic

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway
Hillsboro, OR 97124, USA

Abstract. Intel is applying formal verification to various pieces of mathematical software used in Merced, the first implementation of the new IA-64 architecture. This paper discusses the development of a generic floating point library giving definitions of the fundamental terms and containing formal proofs of important lemmas. We also briefly describe how this has been used in the verification effort so far.

1 Introduction

IA-64 is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, and the forthcoming Merced chip from Intel will be its first silicon implementation. To avoid some of the limitations of traditional architectures, IA-64 incorporates a unique combination of features, including an instruction format encoding parallelism explicitly, instruction predication, and speculative/advanced loads [4]. Nevertheless, it also offers full upwards-compatibility with IA-32 (x86) code.[1]

IA-64 incorporates a number of floating point operations, the centerpiece of which is the `fma` (floating point multiply-add or fused multiply-accumulate). This computes $xy + z$ from inputs x, y and z with a single rounding error. Floating point addition and multiplication are just degenerate cases of `fma`, $1y + z$ and $xy + 0$.[2] On top of the primitives provided by hardware, there is a substantial suite of associated software, e.g. C library functions to approximate transcendental functions.

Intel has embarked on a project to formally verify all Merced's basic mathematical software. The formal verification is being performed in HOL Light, a version of the HOL theorem prover [6]. HOL is an interactive theorem prover in the 'LCF' style, meaning that it encapsulates a small trusted logical core and implements all higher-level inference by (usually automatic) decomposition to these primitives, using arbitrary user programming if necessary.

A common component in all the correctness proofs is a library containing formal definitions of all the main concepts used, and machine-checked proofs of

[1] The worst-case accuracy of the floating-point transcendental functions has actually improved over the current IA-32 chips.

[2] As we will explain later, this is a slight oversimplification.

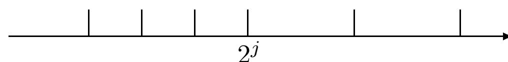
a number of key lemmas. Correctness of the mathematical software starts from the assumption that the underlying hardware floating point operations behave according to the IEEE standard 754 [9] for binary floating point arithmetic. Actually, IEEE-754 doesn't explicitly address `fma` operations, and it leaves underspecified certain significant questions, e.g. NaN propagation and underflow detection. Thus, we not only need to specify the key IEEE concepts but also some details specific to IA-64. Then we need to prove important lemmas. How this was done is the main subject of this paper.

Floating point numbers can be stored either in floating point registers or in memory, and in each case we cannot always assume the encoding is irredundant (i.e. there may be several different encodings of the same real value, even apart from IEEE signed zeros). Thus, we need to take particular care over the distinction between values and their floating point encodings.[3] Systematically making this separation nicely divides our formalization into two parts: those that are concerned only with real numbers, and those where the floating point encodings with the associated plethora of special cases (infinities, NaNs, signed zeros etc.) come into play.

2 Floating point formats

Floating point numbers, at least in conventional binary formats, are those of the form $\pm 2^e k$ with the *exponent* e subject to a certain bound, and the *fraction* (also called significand or mantissa) k expressible in a binary positional representation using a certain number p of bits. The bound on the exponent range together with the allowed *precision* p determines a particular floating point *format*.

Floating point numbers cover a wide range of values from the very small to the very large. They are evenly spaced except that at the points 2^j the interval between adjacent numbers doubles. The intervals $2^j \leq x \leq 2^{j+1}$, possibly excluding one or both endpoints, are often called *binades*, and the numbers 2^j *binade boundaries*. In a decimal analogy, the gap between 1.00 and 1.01 is ten times the gap between 0.999 and 1.00, where all numbers are constrained to three significant digits. The following diagram illustrates this.



Our formalization of the encoding-free parts of the standard is highly generic, covering an infinite collection of possible floating point formats, even including absurd formats with zero precision (no fraction bits). It is a matter of taste whether the pathological cases should be excluded at the outset. We sometimes

[3] In the actual standard (p7) 'a bit-string is not always distinguished from a number it may represent'.

need to exclude them from particular theorems, but many of the theorems turn out to be degenerately true even for extreme values.

Section 3.1 of the standard parametrizes floating point formats by precision p and maximum and minimum exponents E_{max} and E_{min}. We follow this closely, except we represent the fraction by an integer rather than a value $1 \leq f < 2$, and the exponent range by two nonnegative numbers N and E. The allowable floating point numbers are then of the form $\pm 2^{e-N} k$ with $k < 2^p$ and $0 \leq e < E$. This was not done because of the use of biasing in actual floating point encodings (as we have stressed before, we avoid such issues at this stage), but rather to use nonnegative integers everywhere and carry around fewer side-conditions. The cost of this is that one needs to remember the bias when considering the exponents of floating point numbers. We name the fields of a triple as follows:

```
|- exprange (E,p,N) = E

|- precision (E,p,N) = p

|- ulpscale (E,p,N) = N
```

and the definition of the set of real numbers corresponding to a triple is:[4]

```
|- format (E,p,N) =
    { x | ∃s e k. s < 2 ∧ e < E ∧ k < 2 EXP p ∧
                  (x = --(&1) pow s * &2 pow e * &k / &2 pow N)}
```

This says exactly that the format is the set of real numbers representable in the form $(-1)^s 2^{e-N} k$ with $e < E$ and $k < 2^p$ (the additional restriction $s < 2$ is just a convenience). For many purposes, including floating point rounding, we also consider an analogous format with an exponent range unbounded above. This is defined by simply dropping the exponent restriction $e < E$. Note that the exponent is still bounded *below*, i.e. N is unchanged.

```
|- iformat (E,p,N) =
    { x | ∃s e k. s < 2 ∧ k < 2 EXP p ∧
                  (x = --(&1) pow s * &2 pow e * &k / &2 pow N)}
```

We then prove various easy lemmas, e.g.

```
|- &0 IN iformat fmt

|- --x IN iformat fmt = x IN iformat fmt

|- x IN iformat fmt ⟹ (&2 pow n * x) IN iformat fmt
```

[4] The ampersand denotes the injection from $\mathbb{N}$ to $\mathbb{R}$, which HOL's type system distinguishes. The function `EXP` denotes exponentiation on naturals, and `pow` the analogous function on reals.

The above definitions consider the mere existence of triples `(s,e,k)` that yield the desired value. In general there can be many such triples that give the same value. However there is the possibility of a *canonical* representation:

```
|- iformat (E,p,N) =
     { x | ∃s e k. (2 EXP (p - 1) <= k ∨ (e = 0)) ∧
                   s < 2 ∧ k < 2 EXP p ∧
                   (x = --(&1) pow s * &2 pow e * &k / &2 pow N)}
```

This justifies our defining a 'decoding' of a representable real number into a standard choice of sign, exponent and fraction. This is defined using the Hilbert ε operator and from the definition we derive:

```
|- x IN iformat(E,p,N)
   ⟹ (2 EXP (p - 1) <= decode_fraction (E,p,N) x ∨
       (decode_exponent (E,p,N) x = 0)) ∧
      decode_sign (E,p,N) x < 2 ∧
      decode_fraction (E,p,N) x < 2 EXP p ∧
      (x = --(&1) pow (decode_sign (E,p,N) x) *
           &2 pow (decode_exponent (E,p,N) x) *
           &(decode_fraction (E,p,N) x) / &2 pow N)
```

Note that it is these canonical notions, not the fields of any encodings, that we later discuss when we consider, say, whether the fraction of a number is even in rounding to nearest.

We prove that there can only be one restricted triple `(s,e,k)` for a given value, except for differently signed zeros, and these coincide with the canonical decodings defined above. For example:

```
|- s < 2 ∧ k < 2 EXP p ∧
   (2 EXP (p - 1) <= k ∨ (e = 0)) ∧
   (x = --(&1) pow s * &2 pow e * &k / &2 pow N)
   ⟹ (decode_fraction(E,p,N) x = k)
```

Nonzero numbers represented by a canonical triple such that $k < 2^{p-1}$ (and hence with $e = 0$) are often said to be *denormal* or *unnormal*. Other representable values are said to be *normal*. We do not define these terms formally in HOL at this stage, reserving them for properties of actual floating point register encodings, where a subtle terminological distinction is made between 'denormal' and 'unnormal' numbers. But we do now define criteria for an arbitrary real to be in the 'normalized' or 'tiny' range and these are used quite extensively later:

```
|- normalizes fmt x =
     (x = &0) ∨
     &2 pow (precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(x)

|- tiny fmt x = ¬(normalizes fmt x)
```

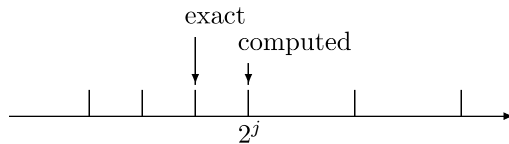
3 Units in the last place

The term 'unit in the last place' is only mentioned in passing by the standard on p. 12 when discussing binary to decimal conversion. Nevertheless, it is of great importance for later proofs because the error bounds for transcendental functions need to be expressed in terms of ulps. Doing so is quite standard, yet there is widespread confusion about what an ulp is, and a variety of incompatible definitions appear in the literature.

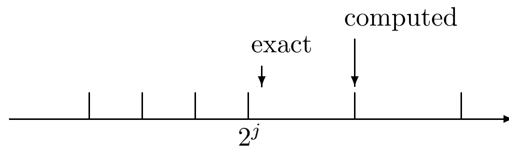
Suppose $x \in \mathbb{R}$ is approximated by $a \in \mathbb{R}$, the latter being representable by a floating point number. For example, x might be the true result of a mathematical function, and a the approximation returned by a floating point operation. What do we mean by saying that the error $|x - a|$ is within $n\ \ ulp$? In the context of a finite binary (or decimal) string, a unit in the last place is naturally understood as the magnitude of its least significant digit, or in other words, the distance between the floating point number a and the next floating point number of greater magnitude. Indeed, if we examine two standard references on the subject, we see the definition framed in both ways:

> In general, if the floating-point number $d.d \cdots d \times \beta^e$ is used to represent z, it is in error by $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}$ units in the last place.[5] (Goldberg [5].)
>
> The term *ulp(x) (*for *unit in the last place*) denotes the distance between the two floating point numbers that are closest to x. (Müller [13].)

Both these definitions have some counterintuitive properties. For example, the following approximation is in error by $0.5ulp$ according to Goldberg, but intuitively, and according to Müller, $1ulp$:



But Müller's definition has the somewhat curious property that its discontinuities occur away from the binade boundaries 2^j, because the closest floating point numbers to a real number x may not be the straddling ones. For example, the following (a valid rounding up) has an error of about $1.4ulp$ according to Müller, but intuitively and according to the Goldberg definition, it is less than 1.



[5] where p is the precision and β the base of the floating point format. For us $\beta = 2$.

Arguably no simple function either of the exact or computed result can avoid counterintuitive properties completely. However, it is very convenient to have such a simple definition. We adopt a definition more like Müller's in that it (a) is a function of the exact value, and (b) it includes a point 2^j in the interval immediately below it. However we effectively insist that an ulp in x is the distance between the two closest *straddling* floating point numbers a and b, i.e. those with $a \leq x \leq b$ and $a \neq b$ assuming an unbounded exponent range.

This seems to convey the natural intuition of units in the last place, and preserves the important mathematical properties that rounding to nearest corresponds to an error of $0.5ulp$ and directed roundings imply a maximum error of $1ulp$. The actual HOL definition is explicitly in terms of binades, and defined using the Hilbert choice operator ε:[6]

```
|- binade(E,p,N) x =
      εe. abs(x) <= &2 pow (e + p) / &2 pow N ∧
          ∀e'. abs(x) <= &2 pow (e' + p) / &2 pow N ⟹ e <= e'

|- ulp(E,p,N) x = &2 pow (binade(E,p,N) x) / &2 pow N
```

After a fairly tedious series of proofs, we eventually derive the theorem that an ulp does indeed yield the distance between two straddling floating point numbers:

```
|- ¬(p = 0)
   ⟹ ∃a b. a IN iformat(E,p,N) ∧ b IN iformat(E,p,N) ∧
              a <= x ∧ x <= b ∧ (b = a + ulp(E,p,N) x) ∧
              ¬(∃c. c IN iformat(E,p,N) ∧ a < c ∧ c < b)
```

4 Rounding

Floating point rounding takes an arbitrary real number and chooses a floating point approximation. Rounding is regarded in the Standard as an operation mapping a real to a member of the extended real line $\mathbb{R} \cup \{+\infty, -\infty\}$, not the space of floating point numbers itself. Thus, encoding and representational issues (e.g. zero signs) are not relevant to rounding. The Standard defines four rounding modes, which we formalize as the members of an enumerated type:

```
roundmode = Nearest | Down | Up | Zero
```

Our formalization defines rounding into a given format as an operation that maps into the corresponding format *with an exponent range unbounded above*. That is, we do not take any special measures like coercing overflows back into the format or to additional 'infinite' elements; this is defined separately when we consider operations. While this separation is not quite faithful to the letter of the

[6] Read '`εe. ...`' as 'the `e` such that ...'.

Standard, we consider our approach preferable. It has obvious technical convenience, avoiding the formally laborious adjunction of infinite elements to the real line and messy side-conditions in some theorems about rounding. Moreover, it avoids duplication of closely related issues in different parts of the Standard. For example, the rather involved criterion for rounding to $\pm\infty$ in round-to-nearest mode in sec. 4.1 of the Standard ('an infinitely precise result with magnitude at least $E_{max}(2 - 2^{-p})$ shall round to ∞ with no change of sign') is not needed. In our setup we later consider numbers that round to values outside the range-restricted format as overflowing, so the exact same condition is implied. This approach in any case *is* used later in the Standard 7.3 when discussing the raising of the overflow exception ('. . . were the exponent range unbounded').

Rounding is defined in HOL as a direct transcription of the Standard's definition. There is one clause for each of the four rounding modes:

```
|- (round fmt Nearest x =
       closest_such (iformat fmt) (EVEN o decode_fraction fmt) x) ∧
   (round fmt Down x = closest a | a IN iformat fmt ∧ a <= x x) ∧
   (round fmt Up x = closest a | a IN iformat fmt ∧ a >= x x) ∧
   (round fmt Zero x =
       closest a | a IN iformat fmt ∧ abs a <= abs x x)
```

For example, the result of rounding `x` down is defined to be the closest to `x` of the set of real numbers `a` representable in the format concerned (`a IN iformat fmt`) and no larger than `x` (`a <= x`). The subsidiary notion of 'the closest member of a set of real numbers' is defined using the Hilbert ε operator. As can be seen from the definition, rounding to nearest uses a slightly elaborated notion of closeness where the result with an even fraction is preferred.[7]

```
|- is_closest s x a =
       a IN s ∧ ∀b. b IN s ⟹ abs(b - x) >= abs(a - x)

|- closest s x = εa. is_closest s x a

|- closest_such s p x =
      εa. is_closest s x a ∧ (∀b. is_closest s x b ∧ p b ⟹ p a)
```

In order to derive useful consequences from the definition, we then need to show that the postulated closest elements always exist. Actually, this depends on the format being nontrivial. For example, if the format has nonzero precision, then rounding up behaves as expected:

[7] Note again the important distinction between real values and encodings. The canonical fraction is used; the question of whether the actual floating point value has an even fraction is irrelevant.

```
|- ¬(precision fmt = 0)
   ==> round fmt Up x IN iformat fmt ∧
       x <= round fmt Up x ∧
       abs(x - round fmt Up x) < ulp fmt x ∧
       ∀c. c IN iformat fmt ∧ x <= c
           ==> abs(x - round fmt Up x) <= abs(x - c)
```

The strongest results for rounding to nearest depend on the precision being at least 2. This is because in a format with $p = 1$ nonzero normalized numbers all have fraction 1, so 'rounding to even' no longer discriminates between adjacent floating point numbers in the same way.

4.1 Lemmas about rounding

While these results are the key to all properties of rounding, there are lots of other important consequences of the definitions that we sometimes use in proofs. For example, rounding is monotonic in all modes:

```
|- ¬(precision fmt = 0) ∧ x <= y ==> round fmt rc x <= round fmt rc y
```

and has various properties like the following:

```
|- ¬(precision fmt = 0) ∧ a IN iformat fmt ∧ a <= x
   ==> a <= round fmt rc x

|- ¬(precision fmt = 0) ∧ a IN iformat fmt ∧ abs(x) <= abs(a)
   ==> abs(round fmt rc x) <= abs(a)
```

Something already representable rounds to itself, and conversely:

```
|- a IN iformat fmt ==> (round fmt rc a = a)

|- ¬(precision fmt = 0)
   ==> ((round fmt rc x = x) = x IN iformat fmt)
```

An important case where a result of a calculation *is* representable is subtraction of nearby quantities.

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧ a / &2 <= b ∧ b <= &2 * a
   ==> (b - a) IN iformat fmt
```

This well-known result [5] can be generalized to subtraction of nearby quantities in formats with more precision (as effectively occur in the intermediate step of an `fma` operation):

```
|- ¬(p = 0) ∧
   a IN iformat (E1,p+k,N) ∧
   b IN iformat (E1,p+k,N) ∧
   abs(b - a) <= abs(b) / &2 pow (k + 1)
   ==> (b - a) IN iformat (E2,p,N)
```

A little thought shows that the first version is easily derivable by linear arithmetic reasoning (automatic in HOL) from this more general version with $k = 1$. Both the general and special case can be strengthened if both inputs are known to be in the same binade, e.g.

```
|- ¬(p = 0) ∧
   a IN iformat (E1,p+k,N) ∧ b IN iformat (E1,p+k,N) ∧
   abs(b - a) <= abs(b) / &2 pow k ∧
   (∀e. &2 pow e / &2 pow N <= abs(b) = &2 pow e / &2 pow N <= abs(a))
   ==> (b - a) IN iformat (E2,p,N)
```

We have also proved some direct cancellation theorems for an `fma` operation. The following embodies the useful fact that one can get an exact representation of a product in two parts by one multiplication and a subsequent `fma` to get a correction term.[8]

```
|- a IN iformat fmt ∧ b IN iformat fmt ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(a * b)
   ==> (a * b - round fmt Nearest (a * b)) IN iformat fmt
```

A few other miscellaneous theorems about rounding include simple relations between rounding modes, e.g.

```
|- ¬(precision fmt = 0) ==> (round fmt Down (--x) = --(round fmt Up x))
```

Plenty of other lemmas are proved formally too.

4.2 Rounding error

One of the central questions in floating point error analysis is the bounding of rounding error. We define rounding error as:

```
|- error fmt rc x = round fmt rc x - x
```

The rounding error is easily bounded in terms of ulps:

```
|- ¬(precision fmt = 0)
   ==> (abs(error fmt Nearest x) <= ulp fmt x / &2) ∧
       (abs(error fmt Down x) < ulp fmt x) ∧
       (abs(error fmt Up x) < ulp fmt x) ∧
       (abs(error fmt Zero x) < ulp fmt x)
```

[8] See `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps` for example; thanks to Paul Miner for pointing us at these documents.

and ulps in their turn can be bounded in terms of relative error, provided denormalization is avoided:

```
|- normalizes fmt x ∧ ¬(precision fmt = 0) ∧ ¬(x = &0)
   ⟹ ulp fmt x <= abs(x) / &2 pow (precision fmt - 1)
```

Conversely, we have a lower bound on ulps in terms of relative error:

```
|- abs(x) / &2 pow (precision fmt) <= ulp fmt x
```

A simple generic result for all rounding modes can be stated in terms of a parameter `mu`:

```
|- (mu Nearest = &1 / &2) ∧
   (mu Down = &1) ∧
   (mu Up = &1) ∧
   (mu Zero = &1)
```

namely:

```
|- normalizes fmt x ∧ ¬(precision fmt = 0)
   ⟹ abs(error fmt rc x)
       <= mu rc * abs(x) / &2 pow (precision fmt - 1)
```

5 Exceptions and flag settings

The IEEE operations not only return values, but also indicate special conditions by setting sticky flags or raising exceptions. These indications are all triggered according to fairly straightforward criteria. For example, the overflow flag is set precisely when the result rounded (as we do anyway) with unbounded upper exponent range is not a member of the actual format with bounded exponent range. Similarly, the inexact flag is set either if overflow occurs or if rounding was nontrivial, i.e. the rounded number was not already representable. It is easy to state these two criteria in terms of existing concepts, e.g.

```
let overflow_flag = ¬(round fmt rc x IN format fmt) in
let inexact_flag = overflow ∨ ¬(round fmt rc x = x) in
...
```

5.1 Underflow

Somewhat more complicated is the definition of underflow.[9] The Standard (sec. 7.4) underspecifies underflow considerably, so it is possible that different implementations of the Standard could set flags or raise exceptions differently. Underflow is said to occur when there is both *tininess* and *loss of accuracy*, and each of these may be detected in two different ways. We have already defined tininess of a number, but the number tested for tininess may either be:

[9] An additional complication is that the criteria for flag-setting and exception-raising are different. We consider only flag setting here.

– The exact result before any rounding.
– The result rounded as if the exponent range were unbounded *below*.

While we already round into a format with the exponent range unbounded *above*, we have no easy way of using our existing infrastructure to define rounding with the exponent range unbounded *below*. Instead, we consider rounding with 'sufficiently large lower exponent range':

```
|- tiny_after_rounding fmt rc x =
      ∃N. N > ulpscale fmt ∧
          tiny fmt (round(exprange fmt,precision fmt,N) rc x)
```

Since this is not a direct transcription of the Standard, we have proved a number of 'sanity check' lemmas to make it clear that this definition is equivalent to the Standard's definition. The crucial one is that if a result is tiny when rounded with a particular lower exponent range, then it will still be tiny for all larger lower exponent ranges:

```
|- 2 <= precision fmt
    ==> (tiny_after_rounding fmt rc x =
          ∃N. N > ulpscale fmt ∧
              ∀M. M >= N
                  ==> tiny fmt
                        (round(exprange fmt,precision fmt,M) rc x))
```

Loss of accuracy may also be detected in more than one way, either as simple inexactness (see above) or as a difference between the actually rounded result and the result rounded as if the exponent range were unbounded below. Again we state a 'profinite' version of this definition and again feel honor-bound to justify it by some additional lemmas.

```
|- losing fmt rc x =
      ∃N. N > ulpscale fmt ∧
          ¬(round (exprange fmt,precision fmt,N) rc x = round fmt rc x)
```

5.2 Relations between underflow conceptions

Using the definitions of the previous section, it is easy to define underflow in any of the ways the Standard allows, including the choice adopted in IA-64. It is of interest to note, however, that there are very strong correlations between the different criteria for tininess and loss of accuracy. In fact, one of them implies all the others (for reasonable formats), as we have formally proved in HOL:

```
|- 2 <= precision fmt ∧ losing fmt rc x
   ==> tiny_after_rounding fmt rc x

|- ¬(precision fmt = 0) ∧ tiny_after_rounding fmt rc x
   ==> tiny fmt x

|- ¬(precision fmt = 0) ∧ losing fmt rc x
   ==> ¬(round fmt rc x = x)
```

Thus, while an implementation may make a variety of choices, many of the combinations collapse into the same one when their meaning is considered. Since the above theorems show that `losing` is the 'weakest' criterion for underflow, it is occasionally worth strengthening some previous theorems to take it as a hypothesis, e.g:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ==> abs(error fmt rc x)
       <= mu rc * abs(x) / &2 pow (precision fmt - 1)
```

The following very useful theorem can be employed to show that the rounding of an `fma` operation does not underflow provided the argument being added is sufficiently far from the low end:

```
|- ¬(precision fmt = 0) ∧
   a IN iformat fmt ∧ b IN iformat fmt ∧ c IN iformat fmt ∧
   &2 pow (2 * precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(c)
   ==> ¬(losing fmt rc (a * b + c))
```

5.3 Flag settings for perfect rounding

Certain software algorithms, e.g. those for division suggested in [11], are designed so that they set no flags and trigger no exceptions in intermediate stages, and culminate in a single `fma` operation that is supposed to deliver the correctly rounded result and set most of the flags, including overflow, underflow and inexact. It is a useful observation in proofs that it suffices to verify only that the result is correct, in the precise sense that the ideal value and the value computed before the final rounding will round identically in all rounding modes. The correctness of these three flags then follows immediately, as does the correct sign of zero results. Properly speaking, in the case of underflow, one needs to prove perfect rounding even assuming unbounded exponent range, but this is usually a straightforward extension. The only case that requires some thought is inexactness, which comes down to the following theorem:

```
|- ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ==> ∀rc. (round fmt rc x = x) = (round fmt rc y = y)
```

Note that the theorem is symmetrical between `x` and `y`, so it suffices to prove that, in any given rounding mode, if `x` rounds to itself, so does `y`. The proof is simple: if `x` rounds to itself, then it must be representable. But by hypothesis, `y` rounds to the same thing, that is `x`, in *all rounding modes*. In particular the roundings up and down imply `x <= y` and `x >= y`, so `y = x`.

Overflow is detected after rounding, so it is immediate that if `x` and `y` round identically, they will either both overflow or both not overflow. Similarly, it is easy to see that underflow behavior is equivalent.[10] For the signs of zeros, it suffices to prove:

```
|- ¬(precision fmt = 0) ∧
   (∀rc. round fmt rc x = round fmt rc y)
   ⟹ (x > &0 = y > &0) ∧ (x < &0 = y < &0)
```

This follows easily from that fact that zero is always representable in any format. For example, if `x` is (strictly) positive, it must round to a strictly positive number in round-up mode. Thus so must `y`, so `y` must also be strictly positive. The other cases are analogous.

6 Encodings

IA-64 includes direct support for several different floating point formats including an internal 82-bit format with a 17-bit exponent and 64-bit fraction (with an explicit 1 bit). Our formalization uses a single HOL type `float`, and all the available floating point numbers can be mapped into this type.

The Standard includes a variety of special numbers such as infinities and NaNs. The subset of 'sensible' values is defined in the HOL formalization by a predicate `finite:float->bool`. It is mainly with numbers in this subset that we will be concerned. The real value of a floating point number is defined by a HOL function `Val:float->real`. Again, we will not show the definition here.

7 Operations

The operations such as addition, subtraction and multiplication are defined in the Standard by composing previously defined concepts in a straightforward way. Roughly speaking, special inputs are treated in some reasonable way, while for finite inputs, the result is generated as if the exact answer were calculated and rounded, with appropriate flag settings. Certain value coercions also happen on overflow, depending on the rounding mode. We will not show the precise definition of the IA-64 operations, but only indicate some respects in which the definitions require care.

[10] Actually we have only proved this for the IA-64 definition of underflow, but other variants would work too.

The IEEE standard does not explicitly address the `fma` operation. Generally speaking, one can extrapolate straightforwardly from the IEEE-754 specifications of addition and multiplication operations. There are some debatable questions, however, mostly connected with the signs of zeros. First, the interpretation of addition and multiplication as degenerate cases of `fma` requires some policy on the sign of $1 \times -0 + 0$. More significantly, the `fma` leads to a new possibility: $a \times b + c$ can round to zero even though the exact result is nonzero. Out of the operations in the standard, this can occur for multiplication or division, but in this case the rules for signs are simple and natural. A little reflection shows that this cannot happen for pure addition, so the rule in the standard that 'the sign of a sum ... differs from at most one of the addend's signs' is enough to fix the sign of zeros when the exact result is nonzero. For the `fma` this is not the case, and IA-64 guarantees that the sign correctly reflects the sign of the exact result in such cases. This is important, for example, in ensuring that certain software algorithms yield the correctly signed zeros in all cases without special measures.

8 Proof tools

The formal theorems proved above capture the main general lemmas about floating point arithmetic that have been found important in the verification undertakings to date. However, it is often important to have special theorem-proving tools based around (variants of) the lemmas, to avoid the tedium of manually applying them and proving routine side-conditions. Broadly speaking, the operations that are most important to automate involve 'symbolic execution' of various kinds.

8.1 Explicit execution

It often happens that one needs to 'evaluate' floating point operations or associated formal concepts for particular explicit values. For example, one often wants to:

- Calculate `ulp(r)` for a particular rational number `r`.
- Calculate `round fmt rc r` for a particular floating point format `fmt`, rounding mode `rc` and rational number `r`.
- Evaluate `Val(a)` for a particular floating point number `a`.
- Prove that a particular floating point value is non-exceptional, i.e. return a theorem `|- finite(a)` for a particular floating point number `a`.

We have implemented HOL *conversions* (see [15] for more on conversions) to do all these, and a few other operations too. Now, explicit details of this sort can be disposed of automatically. For example, the conversion `ROUND_CONV` takes rounding parameters and a rational number to be rounded and not only returns the 'answer', but also a formally proved theorem that the answer is correct. (Under the surface, theorems about the uniqueness of rounding are applied.)

```
#ROUND_CONV `round (10,11,12) Nearest (&22 / &7)`;;
it : thm = |- round (10,11,12) Nearest (&22 / &7) = &1609 / &512
```

HOL already includes proof tools to perform explicit calculation with rational numbers and even with computable real numbers [8]. In conjunction with the new proof tools, we now have powerful automatic assistance for goals involving all forms of explicit calculation.

8.2 Automated error analysis

Explicit computations are not always enough. Sometimes one does not know the actual floating point values involved, merely some properties such as maximum or minimum magnitudes and the maximum absolute or relative error from some 'ideal' value. We have implemented HOL tools to propagate knowledge of such properties through additional `fma` operations. Using these, it is simple to get a formally proven absolute or relative error bound for a sequence of `fma` operations, e.g. the evaluation of a polynomial approximation to a transcendental function, completely automatically, given only some assumptions on the input number(s).

Whether one wants absolute or relative error depends on the kind of proofs being undertaken. When trying to get a sharp ulp error bound for a transcendental function approximation, we find it useful to split the ideal output into binades (determining the ulp value), and evaluate the maximum absolute error on each corresponding input set. We can then see which binade yields the largest ulp error, without the loosening of the error bound that would be caused by using relative error. However, typically the errors are large only for a few binades, so we still use relative error to dispose of all the others, for efficiency reasons. (For extended precision, there could be around 2^{15} different binades.)

The proof tool for absolute errors requires theorems about each nonconstant input `x` to an `fma` operation of the following form: [11]

```
|- finite x

|- abs(Val x) <= b

|- abs(Val x - y) <= e
```

Here `b` and `e` must be expressions made up of rational constants, while `y`, the value approximated, can be any expression. From this information, the proof tool automatically derives analogous assertions for the output of the `fma` operation. At present, fairly crude maximization techniques are used to evaluate the range and error in the output. This has proved fine for verifications undertaken to date, since the intermediate inputs tend to be monotonic over the fairly narrow intervals considered. However, we are presently considering a more sophisticated mechanism to get tighter error bounds.

[11] Assumptions of this sort are only needed for variables, as they are derived automatically for explicit values as described in the previous section.

For relative error, the approach is analogous, with the absolute error `e` replaced by a relative error. Moreover, an additional assumption is needed about the *minimum* (nonzero) size of the input, because to get a sharp relative error result we need to prove that underflow doesn't occur. We use the following definition:

```
|- zorbigger a x = &0 <= a ∧ ((x = &0) ∨ a <= abs(x))
```

It is straightforward to propagate such assumptions through expressions for varying threshold `a`, using theorems such as the following:

```
|- zorbigger a1 x1 ∧ zorbigger a2 x2 ⟹ zorbigger (a1 * a2) (x1 * x2)

|- x1 IN iformat fmt ∧ x2 IN iformat fmt ∧ x3 IN iformat fmt ∧
   zorbigger a3 x3 ∧ ¬(x3 = &0)
   ⟹ zorbigger (a3 / &2 pow (2 * precision fmt)) (x1 * x2 + x3)
```

and then when required we can derive normalization from the lemma:

```
|- normalizes fmt =
    zorbigger (&2 pow (precision fmt - 1) / &2 pow ulpscale fmt)
```

8.3 Intermediate levels of explicitness

There are some other possibilities that fall between the previous categories. For example, we have formally checked some correctness proofs for floating point square root algorithms using a methodology discussed in [2]. This methodology gives us a proof of correctness for all but a certain set of values, isolated using number-theoretic considerations. It is then necessary, to get an overall correctness proof, to check the remaining values explicitly.

While in principle this can be done with explicit calculation, such an approach is inefficient and unnatural, because the set of values is parametrized by a much smaller set of e_i and k_i by simply varying the exponent while preserving its even/odd parity:

$$2^{e_i+2n}k_i$$

It is much more natural to check the values only for a particular n, say $n = 0$, and then extrapolate from that. This can be justified by scaling theorems for rounding, provided overflow cannot occur for the maximum exponent. The scaling theorem for rounding also requires that loss of precision is avoided; one sufficient condition is shown in the next theorem.

```
|- 2 <= precision fmt ∧
   (&2 pow (precision fmt - 1) / &2 pow ulpscale fmt <= abs x ∨
    (round fmt rc x = x))
   ⟹ (round fmt rc (&2 pow n * x) = &2 pow n * round fmt rc x)
```

At present, we have not automated this kind of scaling analysis completely, but it has been taken far enough that the proofs were all reasonably straightforward. If we did many more proofs of the same kind, further effort would be needed.

9 Conclusions and related work

We have detailed a theory of floating point arithmetic that is generic over a wide variety of floating point formats, and has then been specialized to the particular formats used in IA-64. By contrast, an earlier formalization by ourselves [7] required duplication of results for different precisions and did not achieve the same neat separation between floating point values and their encodings. Our present formulation contains a far larger collection of medium-level lemmas than any other formalization we are aware of. In contrast to some previous IEEE-754 specifications such as one in Z [1], ours is completely formal and all results have been logically proved by machine.

Most of the definitions (excluding, perhaps, some of those connected with underflow) are a direct formal translation of the Standard, making their correctness highly intuitive. For example, our definition of floating point rounding is the same as the Standard's, whereas all related machine-checked formalizations of which we are aware [12, 14, 16] use less intuitive translations. In some cases, this is forced by the limited mathematical expressiveness of other theorem provers.

The price one pays for intuitive high-level specifications is that one cannot automatically 'execute' the formal specification in the proof process. By contrast, ACL2 specifications like Rusinoff's [16] are always executable by construction. We have ameliorated this shortcoming by providing a suite of automatic proof tools that can, effectively, execute specifications, and moreover can do more sophisticated forms of symbolic evaluation including automatic error analysis of chains of floating point computations. Since such 'execution' merely abbreviates and automates standard logical inferences, we have the advantage of generating a formal proof rather than relying on a separate execution mechanism. The only drawback of this is that using standard logical inferences is relatively slow.

The formalization described here has been used quite extensively in verification of various algorithms for division and square root [3] and some transcendental functions [17]. It is hoped that we can describe these verifications in more detail at a later date. Such verifications sometimes combine nontrivial continuous mathematics with low-level machine details and a certain amount of explicit execution. Using a general theorem prover like HOL equipped with our formalization and proof tools, all these disparate aspects can be unified in one system, and the final result verified according to the strictest standards of logical rigor. For example, one of the transcendental function verifications involves approximately 77 million primitive logical inferences. However, generating such big proofs is quite feasible as most of the more tedious parts are automatic.

References

1. M. Barratt. Formal methods applied to a floating-point system. *IEEE Transactions on Software Engineering*, 15:611–621, 1989.
2. M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.
3. M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In Koren and Kornerup [10], pages 96–105.
4. C. Dulong. The IA-64 architecture at work. *IEEE Computer*, 64(7):24–32, July 1998.
5. D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. J. Harrison. Floating point verification in HOL Light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997.
8. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
9. IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.
10. I. Koren and P. Kornerup, editors. *Proceedings, 14th IEEE symposium on on computer arithmetic*, Adelaide, Australia, 1999. IEEE Computer Society.
11. P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.
12. P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical memorandum 110167, NASA Langley Research Center, Hampton, VA 23681-0001, USA, 1995.
13. J-M. Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 1997.
14. J. O'Leary, X. Zhao, R. Gerth, and C-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.
15. L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.
16. D. Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web via `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
17. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In Koren and Kornerup [10], pages 4–11.

This article was typeset using the LaTeX macro package with the LLNCS2E class.