A Machine-Checked Theory of Floating Point Arithmetic

John Harrison

Intel Corporation, EY2-03

- Introduction to IA-64 and HOL Light
- Floating point formats
- Ulps
- Rounding
- Proof tools and execution
- Conclusions

IA-64 overview

IA-64 is a new 64-bit computer architecture jointly developed by Hewlett-Packard and Intel, and the forthcoming Merced chip from Intel is its first silicon implementation. Among the special features of IA-64 are:

- An instruction format encoding parallelism explicitly
- Instruction predication
- Speculative and advanced loads
- Upward compatibility with IA-32 (x86).

The IA-64 Applications Developer's Architecture Guide is now available from Intel in printed form and online:

`http://developer.intel.com/design/ia64/downloads/adag.htm`

HOL Light

Verifications are conducted using the HOL Light theorem prover.

- A simplified version of HOL:
  - Coded in CAML Light
  - More minimalist axiomatic foundations
  - Structured more rationally
- LCF-style system:
  - Every theorem created by primitive rules
  - All theories developed definitionally
  - Full programmability in ML toplevel

We are using HOL Light to formally verify various pieces of mathematical (floating point) software. This talk covers the basic theories.

Floating point numbers

There are various different schemes for floating point numbers. Usually, the floating point numbers are those representable in some number n of significant binary digits, within a certain exponent range, i.e.

$$(-1)^s \times d_0.d_1 d_2 \cdots d_n \times 2^e$$

where

- $s \in \{0, 1\}$ is the *sign*
- $d_0.d_1 d_2 \cdots d_n$ is the *significand* and $d_1 d_2 \cdots d_n$ is the *fraction*. These are not always used consistently; sometimes 'mantissa' is used for one or the other
- e is the exponent.

We often refer to $p = n + 1$ as the *precision*.

HOL floating point formats

We have formalized a generic floating point theory in HOL, which can be applied to all the IA-64 formats, and others supported in software such as quad precision.

A floating point format is identified by a triple of natural numbers `fmt`.

The corresponding set of real numbers is `format(fmt)`, or ignoring the upper limit on the exponent, `iformat(fmt)`.

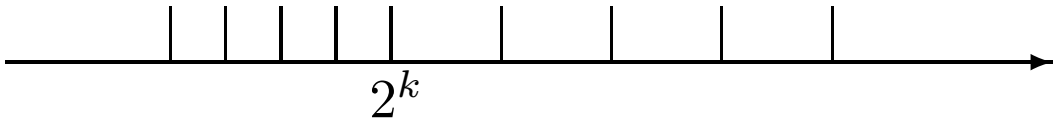
```
|- iformat (E,p,N) =
     { x | ∃s e k. s < 2 ∧ k < 2 EXP p ∧
                (x = --(&1) pow s * &2 pow e *
                     &k / &2 pow N)}
```

We distinguish carefully between actual floating point numbers (as bitstrings) and the corresponding real numbers. For the central concept, rounding, only the latter is relevant.

Units in the last place

It's customary to give a bound on the error in transcendental functions in terms of 'units in the last place' (ulps).

While ulps are a standard way of measuring error, there's a remarkable lack of unanimity in published definitions of the term. One of the merits of a formal treatment is to clear up such ambiguities.



Roughly, a unit in the last place is the gap between adjacent floating point numbers. But at the boundary 2^k between 'binades', this distance changes.

Two definitions

Goldberg considers the binade containing the computed result:

> In general, if the floating-point number $d.d \cdots d \times \beta^e$ is used to represent z, it is in error by $|d.d \cdots d - (z/\beta^e)|\beta^{p-1}e$ units in the last place.

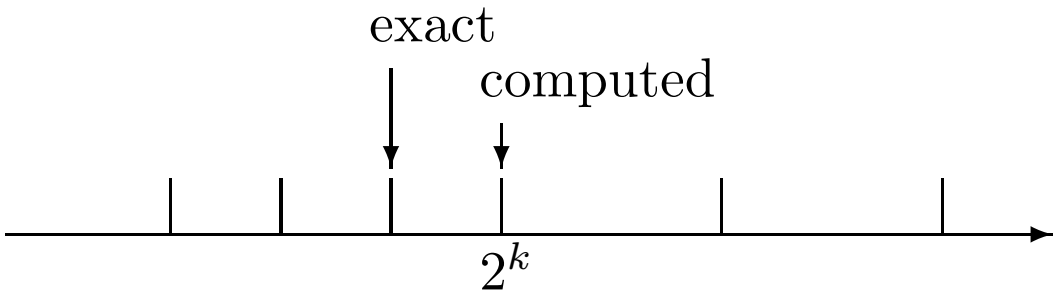
Muller considers the binade containing the exact result:

> The term *ulp(x)* (for *unit in the last place*) denotes the distance between the two floating point numbers that are closest to x.

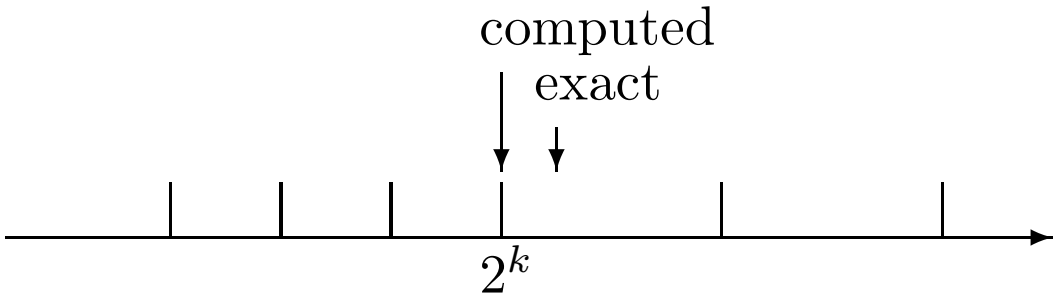
However these both have counterintuitive properties.

Problems with these definitions

An error of $0.5ulp$ according to Goldberg, but intuitively $1ulp$.



An error of $0.4ulp$ according to Muller, but intuitively 0.2 ulp. Rounding up is worse...



Our definition: $ulp(x)$ is the distance between the closest pair of floating point numbers a and b with $a \leq x \leq b$. Note that we are counting the exact result 2^k as belonging to the binade *below*.

Rounding

Rounding is controlled by a rounding mode, which is defined in HOL as an enumerated type:

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

We define notions of 'closest approximation' as follows:

```
|- 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)
```

and hence define rounding:

```
|- (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)
```

Theorems about rounding

We prove some basic properties of rounding, e.g. that an already-representable number 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)
```

and that rounding is monotonic in all rounding modes:

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

There are various other simple properties, e.g. symmetries and skew-symmetries like:

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

Exact calculation

It's often important to prove that certain expressions in terms of floating point numbers are themselves representable, and hence when calculated with machine arithmetic operations incur no rounding error. For example the following is a classic result:

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

The following shows how we can retrieve the rounding error in multiplication using a fused multiply-accumulate (available on IA-64).

```
|- 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
```

Here's a similar one for addition and subtraction:

```
|- x IN iformat fmt ∧ y IN iformat fmt ∧ abs(x) <= abs(y)
   ⟹ (round fmt Nearest (x + y) - y) IN iformat fmt ∧
       (round fmt Nearest (x + y) - (x + y)) IN iformat fmt
```

Proof tools and execution

Several definitions are highly non-constructive, notably rounding. However we often need to prove what the result of rounding a particular number is. We have a conversion `ROUND_CONV` that will round a rational number to a given format while returning an equational theorem, e.g.

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

Internally, HOL derives this using theorems about sufficient conditions for correct rounding. In ACL2, we would be forced to adopt a non-standard constructive definition, but would then have such proving procedures without further work and highly efficient.

More generally, we have a number of proof tools to apply routine forms of reasoning automatically, disposing of side-conditions. These tools can, for example, derive absolute or relative error bounds in a sequence of floating point operations.

Conclusions

Our formalization has the following properties:

- Complete genericity over arbitrary floating point formats, which even includes (sign-magnitude) integers as a special case.
- Precise formalization of all the main IEEE concepts such as formats, rounding, flags and exceptions, as well as notions like ulps.
- Extensive collection of important and non-trivial lemmas that are often needed.
- Support from automatic proof tools to automate explicit execution and other important proof steps like error analysis.

It has been used in the formal verification of software for division, square root and various transcendental functions. As we do more examples, we will further extend and refine the theory.