Formal Proofs of Floating-Point Algorithms

John Harrison
Intel Corporation

SCAN 2010

28th September 2010

Overview

- Formal verification and theorem proving
- Formalizing floating-point arithmetic
- Square root example
- Reciprocal example
- Tangent example
- Summary

The FDIV bug

There have been several notable computer arithmetic failures, with one being particuarly significant for Intel:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $\$475$ million to cover the costs.

Limits of testing

Bugs are usually detected by extensive testing, including pre-silicon simulation.

- Slow — especially pre-silicon
- Too many possibilities to test them all

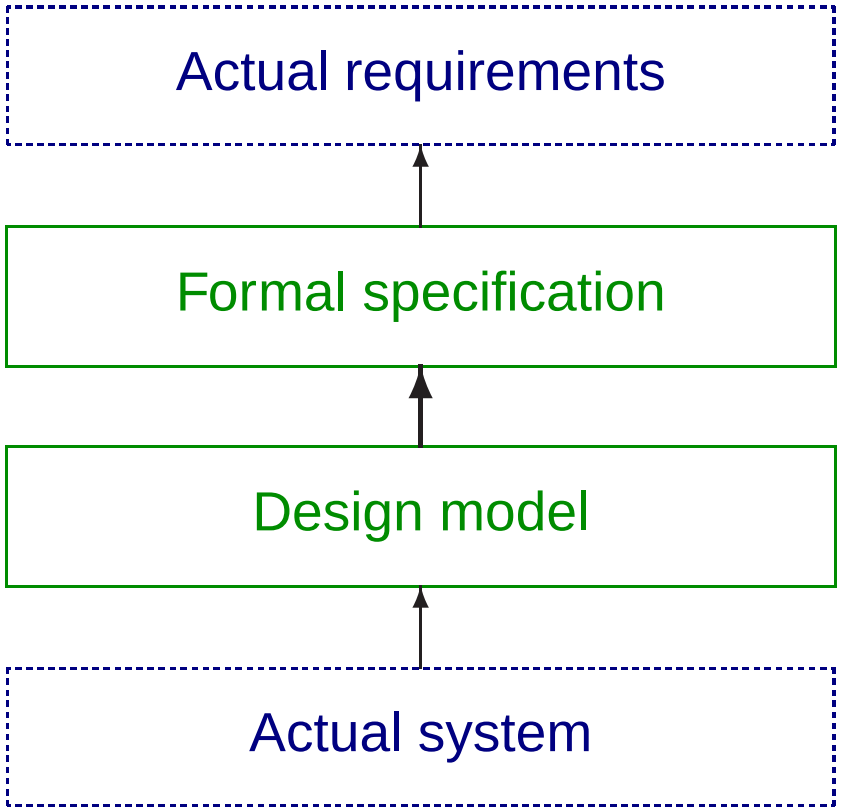
For example:

- 2^{160} possible pairs of floating point numbers (possible inputs to an adder).
- Vastly higher number of possible states of a complex microarchitecture.

The alternative is formal verification (FV).

Formal verification

Formal verification: mathematically prove the correctness of a *design* with respect to a mathematical *formal specification*.



Verification vs. testing

Verification has some advantages over testing:

- Exhaustive.
- Improves our intellectual grasp of the system.

However:

- Difficult and time-consuming.
- Only as reliable as the formal models used.
- How can we be sure the proof is right?

Formal verification methods

Many different methods are used in formal verification, mostly trading efficiency and automation against generality.

- Propositional tautology checking
- Symbolic simulation
- Symbolic trajectory evaluation
- Temporal logic model checking
- Decidable subsets of first order logic
- First order automated theorem proving
- Interactive theorem proving

Intel uses pretty much all these techniques in various parts of the company.

Our work

We will focus on our own formal verification activities:

- Formal verification of floating-point operations
- Targeted at the Intel® Itanium® processor family.
- Conducted using the interactive theorem prover HOL Light.

Why floating-point?

There are obvious reasons for focusing on floating-point:

- Known to be difficult to get right, with several issues in the past.
  We don't want another FDIV!
- Quite clear specification of how most operations *should* behave.
  We have the IEEE Standard 754.

However, Intel is also applying FV in many other areas, e.g. control logic, cache coherence, bus protocols . . .

Why interactive theorem proving?

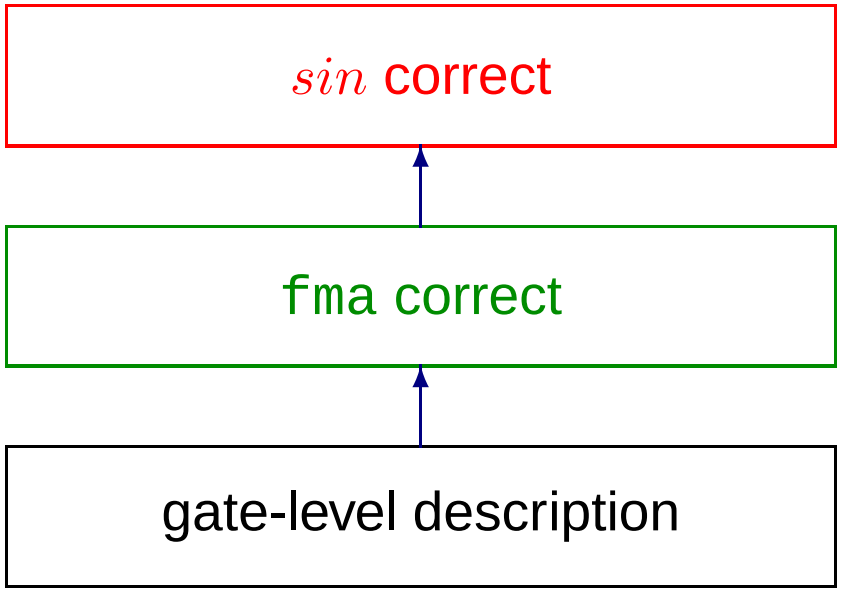
Limited scope for highly automated finite-state techniques like model checking.

It’s difficult even to specify the intended behaviour of complex mathematical functions in bit-level terms.

We need a general framework to reason about mathematics in general while checking against errors.

Levels of verification

High-level algorithms assume correct behavior of some hardware primitives.



Proving my assumptions is someone else's job . . .

Characteristics of this work

The verification we're concerned with is somewhat atypical:

- Rather simple according to typical programming metrics, e.g. 5-150 lines of code, often no loops.
- Relies on non-trivial mathematics including number theory, analysis and special properties of floating-point rounding.

Tools that are often effective in other verification tasks, e.g. temporal logic model checkers, are of almost no use.

HOL Light overview

HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

HOL Light is designed to have a simple and clean logical foundation.

Uses Objective CAML (OCaml) as both implementation and interaction language.

What does LCF mean?

The name is a historical accident:

> The original Stanford and Edinburgh LCF systems were for Scott's Logic of Computable Functions.

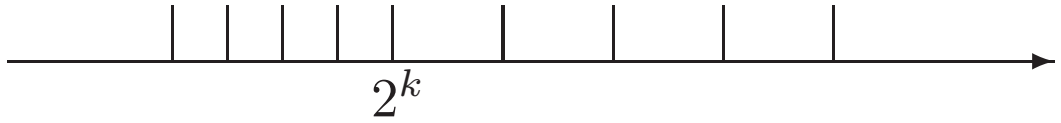
The main features of the LCF approach to theorem proving are:

- Reduce all proofs to a small number of relatively simple primitive rules
- Use the programmability of the implementation/interaction language to make this practical

Gives an excellent combination of reliability/security with extensibility/programmability.

The problem of specification: 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), but the formal specification needs care.



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.

Do we consider the binade containing the exact or computed result? Are we taking enough care when we talk about the 'two closest floating-point numbers' to the exact answer?

IEEE-correct operations

The IEEE Standard 754 specifies for all the usual algebraic operations, including square root and fused multiply-add, but *not* transcendentals:

> . . . each of these operations shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range and then coerced this intermediate result to fit in the destination's format [using the specified rounding operation]

The Standard defines rounding in terms of exact real number computation, and we can render this directly in HOL Light's logic.

Rounding (1)

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)
```

Rounding (2)

Hence the actual definition of 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)
```

Note that this is almost a direct transcription of the standard; no need to talk about ulps etc.

But it is also completely non-constructive!

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))
```

The $(1 + \epsilon)$ property

Designers often rely on clever “cancellation” tricks to avoid or compensate for rounding errors.

But many routine parts of the proof can be dealt with by a simple conservative bound on rounding error:

```
|- normalizes fmt x ∧
   ¬(precision fmt = 0)
   ⇒ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
           round fmt rc x = x * (&1 + e)
```

Derived rules apply this result to computations in a floating point algorithm automatically, discharging the conditions as they go.

Exact calculation

A famous theorem about exact calculation:

```
|- 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-add.

```
|- 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 can prove equivalence to a constructive definition and hence prove particular numerical results:

```
#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.

Division and square root

The Intel® Itanium® architecture uses some interesting multiplicative algorithms relying *purely* on conventional floating-point operations, à la Markstein.

Easy to get within a bit or so of the right answer, but meeting the IEEE spec is significantly more challenging.

In addition, all the flags need to be set correctly, e.g. inexact, underflow, ...

Typical verification process

The verification process for all these algorithms typically involves two different parts:

- Accumulate bounds on the (usually relative) accuracy of the various intermediate quantities up to the penultimate iteration.
- Use some more delicate theorems or explicit case analysis to complete the proof.

The first part is a generic problem in many floating-point algorithms, and we can automate it to a large extent.

The second part can allow automation, but the details are different for various algorithms and proof techniques.

Accumulating error bounds

This error accumulation is automated by proof tools that

- Apply the $1 + \epsilon$ property and prove side-conditions. This often involves proving absence of overflow/underflow.
- Automatically bound error in composite expressions by naive 'triangle law' reasoning. Often necessary to normalize polynomials first, e.g. in $x(1 + \epsilon)(1 - \epsilon)(1 + \delta)$.

Such techniques are used all over the place in analyzing floating-point algorithms. In less 'subtle' algorithms this would be pretty much the whole verification.

Proving perfect rounding

There are two distinct approaches to justifying perfect rounding:

- Specialized theorems that analyze the precise way in which the approximation y^* rounds and how this relates to the mathematical function required.
- More direct theorems that are based on general properties of the function being approximated.

Techniques of both kinds have been formalized in HOL.

- Verification of division algorithms based on a special technique due to Peter Markstein.
- Verification of square root algorithms based on an ‘exclusion zone’ method due to Marius Cornea

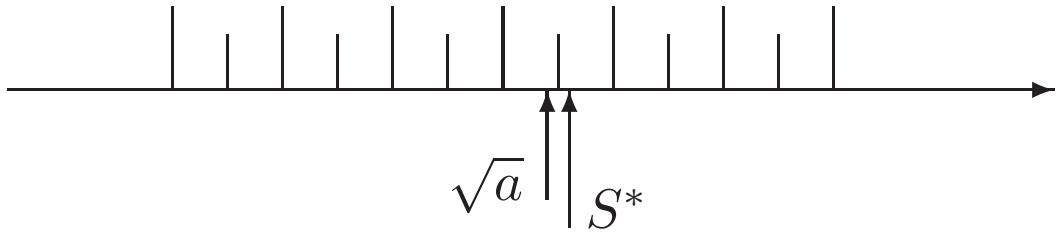
A square root algorithm

Also based on refinement of an initial approximation.

1. $y_0 = \frac{1}{\sqrt{a}}(1+\epsilon)$ $\quad b = \frac{1}{2}a$
2. $z_0 = y_0^2$ $\quad S_0 = ay_0$
3. $d = \frac{1}{2} - bz_0$ $\quad k = ay_0 - S_0$ $\quad H_0 = \frac{1}{2}y_0$
4. $e = 1 + \frac{3}{2}d$ $\quad T_0 = dS_0 + k$
5. $S_1 = S_0 + eT_0$ $\quad c = 1 + de$
6. $d_1 = a - S_1 S_1$ $\quad H_1 = cH_0$
7. $S = S_1 + d_1 H_1$

Condition for perfect rounding

For perfect rounding we want to ensure that the two real numbers $\sqrt{a}$ and $S^* = S_1 + d_1 H_1$ never fall on opposite sides of a midpoint between two floating point numbers, as here:



Rather than analyzing the rounding of the final approximation explicitly, we can just appeal to general properties of the square root function.

Exclusion zones

It would suffice if we knew for any midpoint m that:

$$|\sqrt{a} - S^*| < |\sqrt{a} - m|$$

In that case $\sqrt{a}$ and S^* cannot lie on opposite sides of m. Here is the formal theorem in HOL:

```
|- ¬(precision fmt = 0) ∧
   (∀m. m IN midpoints fmt ⇒ abs(x - y) < abs(x - m)
   ⇒ round fmt Nearest x = round fmt Nearest y
```

Square root exclusion zones

This is possible to prove, because in fact every midpoint m is surrounded by an 'exclusion zone' of width $\delta_m > 0$ within which the square root of a floating point number cannot occur.

However, this δ can be quite small, considered as a relative error. If the floating point format has precision p, then we can have $\delta_m \approx |m|/2^{2p+2}$.

Example: square root of significand that's all 1s.

Difficult cases

So to ensure the equal rounding property, we need to make the final approximation before the last rounding accurate to *more than twice* the final accuracy.

The fused multiply-add can help us to achieve *just under twice* the accuracy, but to do better is slow and complicated. How can we bridge the gap?

We can use a technique due to Marius Cornea combining analytic and combinatorial techniques.

Mixed analytic-combinatorial proofs

Only a fairly small number of possible inputs a can come closer than say $2^{-(2p-1)}$.

For all the other inputs, a straightforward relative error calculation yields the result.

We can then use number-theoretic reasoning to isolate the additional cases we need to consider, then simply ‘execute’ them (logically).

More than likely we will be lucky, since all the error bounds are worst cases and even if the error is exceeded, it might be in the right direction to ensure perfect rounding anyway.

Isolating difficult cases

Straightforward to show that the difficult cases have mantissas m, considered as p-bit integers, such that one of the following diophantine equations has a solution k for d a small integer.

$$2^{p+2}m = k^2 + d$$

or

$$2^{p+1}m = k^2 + d$$

We consider the equations separately for each chosen d. For example, we might be interested in whether:

$$2^{p+1}m = k^2 - 7$$

has a solution. If so, the possible value(s) of m are added to the set of difficult cases.

Solving the equations

It's quite easy to program HOL to enumerate all the solutions of such diophantine equations, returning a disjunctive theorem of the form:

$$2^{p+1}m = k^2 + d \Rightarrow m = n_1 \vee \ldots \vee m = n_i$$

The procedure simply uses even-odd reasoning and recursion on the power of two. For example, if

$$2^{25}m = k^2 - 7$$

then we know k must be odd; we can write $k = 2k' + 1$ and get the derived equation:

$$2^{24}m = 2k'^2 + 2k' - 3$$

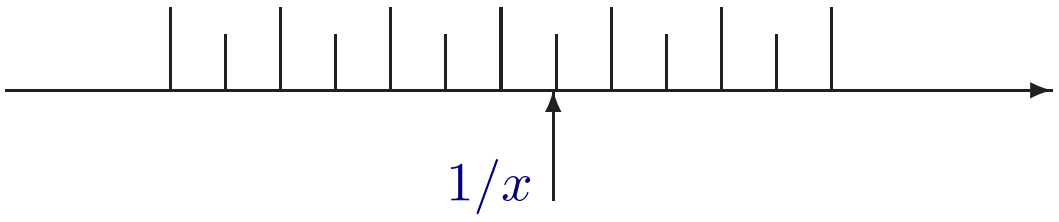
By more even/odd reasoning, this has no solutions. Always recurse down to an equation that is unsatisfiable or immediately solvable.

A similar example: Difficult cases for reciprocals

Some algorithms for floating-point division, a/b, can be optimized for the special case of reciprocals ($a = 1$).

A direct analytic proof of the optimized algorithm is sometimes too hard because of the intricacies of rounding.

However, an analytic proof works for all but the ‘difficult cases’.



These are floating-point numbers whose reciprocal is very close to another one, or a midpoint, making them trickier to round correctly.

Finding difficult cases with factorization

After scaling to eliminate the exponents, finding difficult cases reduces to a straightforward number-theoretic problem.

A key component is producing the prime factorization of an integer *and* proving that the factors are indeed prime.

In typical applications, the numbers can be 49–227 bits long, so naive approaches based on testing all potential factors are infeasible.

The primality prover is embedded in a HOL derived rule `PRIME_CONV` that maps a numeral to a theorem asserting its primality or compositeness.

Certifying primality

We generate a ‘certificate of primality’ based on Pocklington’s theorem:

```
|- 2 ≤ n ∧
   (n - 1 = q * r) ∧
   n ≤ q EXP 2 ∧
   (a EXP (n - 1) == 1) (mod n) ∧
   (∀p. prime(p) ∧ p divides q
        ⇒ coprime(a EXP ((n - 1) DIV p) - 1,n))
   ⇒ prime(n)
```

The certificate is generated ‘extra-logically’, using the factorizations produced by PARI/GP.

The certificate is then checked by formal proof, using the above theorem.

Typical results

```
0xFFFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFFFD 0xFE421D63446A3B34 0xFBFC17DFE0BEFF04 0xFB940B119826E598
0xFB0089D7241D10FC 0xFA0BF7D05FBE82FC 0xF912590F016D6D04 0xF774DD7F912E1F54 0xF7444DFBF7B20EAC
0xF39EB657E24734AC 0xF36EE790DE069D54 0xF286AD7943D79434 0xEDF09CCC53942014 0xEC4B058D0F7155BC
0xEC1CA6DB6D7BD444 0xE775FF856986AE74 0xE5CB972E5CB972E4 0xE58469F0234F72C4 0xE511C4648E2332C4
0xE3FC771FE3B8FF1C 0xE318DE3C8E6370E4 0xE23B9711DCB88EE4 0xE159BE4A8763011C 0xDF738B7CF7F482E4
0xDEE256F712B7B894 0xDEE24908EDB7B894 0xDE86505A77F81B25 0xDE03D5F96C8A976C 0xDDFF059997C451E5
0xDB73060F0C3B6170 0xDB6DB6DB6DB6DB6C 0xDB6DA92492B6DB6C 0xDA92B6A4ADA92B6C 0xD9986492DD18DB7C
0xD72F32D1C0CC4094 0xD6329033D6329033 0xD5A004AE261AB3DC 0xD4D43A30F2645D7C 0xD33131D2408C6084
0xD23F53B88EADABB4 0xCCCE6669999CCCD0 0xCCCE666666633330 0xCCCCCCCCCCCCCCD0 0xCBC489A1DBB2F124
0xCB21076817350724 0xCAF92AC7A6F19EDC 0xC9A8364D41B26A0C 0xC687D6343EB1A1F4 0xC54EDD8E76EC6764
0xC4EC4EC362762764 0xC3FCF61FE7B0FF3C 0xC3FCE9E018B0FF3C 0xC344F8A627C53D74 0xC27B1613D8B09EC4
0xC27B09EC27B09EC4 0xC07756F170EAFBEC 0xBDF3CD1B9E68E8D4 0xBD5EAF57ABD5EAF4 0xBCA1AF286BCA1AF4
0xB9B501C68DD6D90C 0xB880B72F050B57FC 0xB85C824924643204 0xB7C8928A28749804 0xB7A481C71C43DDFC
0xB7938C6947D97303 0xB38A7755BB835F24 0xB152958A94AC54A4 0xAFF5757FABABFD5C 0xAF4D99ADFEFCAAFC
0xAF2B32F270835F04 0xAE235074CF5BAE64 0xAE0866F90799F954 0xADCC548E46756E64 0xAD5AB56AD5AB56AC
0xAD5AAA952AAB56AC 0xAB55AAD56AB55AAC 0xAAAAB55555AAAAAC 0xAAAAAAAAAAAAAAAC 0xAAAAA00000555554
0xA93CFF3E629F347D 0xA80555402AAA0154 0xA8054ABFD5AA0154 0xA7F94913CA4893D4 0xA62E84F95819C3BC
0xA5889F09A0152C44 0xA4E75446CA6A1A44 0xA442B4F8DCDEF5BC 0xA27E096B503396EE 0x9E9B8FFFFFD8591C
0x9E9B8B0B23A7A6E4 0x9E7C6B0C1CA79F1C 0x9DFC78A4EEEE4DCB 0x9C15954988E121AB 0x9A585968B4F4D2C4
0x99D0C486A0FAD481 0x99B831EEE01FB16C 0x990C8B8926172254 0x990825E0CD75297C 0x989E556CADAC2D7F
0x97DAD92107E19484 0x9756156041DBBA94 0x95C4C0A72F501BDC 0x94E1AE991B4B4EB4 0x949DE0B0664FD224
0x942755353AA9A094 0x9349AE0703CB65B4 0x92B6A4ADA92B6A4C 0x9101187A01C04E4C 0x907056B6E018E1B4
0x8F808E79E77A99C4 0x8F64655555317C3C 0x8E988B8B3BA3A624 0x8E05E117D9E786D5 0x8BEB067D130382A4
0x8B679E2B7FB0532C 0x887C8B2B1F1081C4 0x8858CCDCA9E0F6C4 0x881BB1CAB40AE884 0x87715550DCDE29E4
0x875BDE4FE977C1EC 0x86F71861FDF38714 0x85DBEE9FB93EA864 0x8542A9A4D2ABD5EC 0x8542A150A8542A14
0x84BDA12F684BDA14 0x83AB6A090756D410 0x83AB6A06F8A92BF0 0x83A7B5D13DAE81B4 0x8365F2672F9341B4
0x8331C0CFE9341614 0x82A5F5692FAB4154 0x8140A05028140A04 0x8042251A9D6EF7FC
```

Transcendental functions: tangent algorithm

- The input number X is first reduced to r with approximately $|r| \leq \pi/4$ such that $X = r + N\pi/2$ for some integer N. We now need to calculate $\pm tan(r)$ or $\pm cot(r)$ depending on N modulo 4.
- If the reduced argument r is still not small enough, it is separated into its leading few bits B and the trailing part $x = r - B$, and the overall result computed from $tan(x)$ and pre-stored functions of B, e.g.

$$tan(B + x) = tan(B) + \frac{\frac{1}{sin(B)cos(B)}tan(x)}{cot(B) - tan(x)}$$

- Now a power series approximation is used for $tan(r)$, $cot(r)$ or $tan(x)$ as appropriate.

Overview of the verification

To verify this algorithm, we need to prove:

- The range reduction to obtain r is done accurately.
- The mathematical facts used to reconstruct the result from components are applicable.
- Stored constants such as $tan(B)$ are sufficiently accurate.
- The power series approximation does not introduce too much error in approximation.
- The rounding errors involved in computing with floating point arithmetic are within bounds.

Most of these parts are non-trivial. Moreover, some of them require more pure mathematics than might be expected.

Automation and re-use

In one sense, HOL Light is an ideal vehicle for this kind of work:

- Everything from the pure mathematics and the general results on floating-point to the explicit ‘execution’ on specific cases, can be done in a single integrated system.
- Because HOL Light is programmable, many of the ‘custom’ tasks can be automated.

In practice, the automation we have implemented is only partial. We get pretty good re-use, but often need to edit big proof scripts, still difficult for a beginner.

Using external tools

Some of the central tasks might be better automated using external tools:

- Accumulation of rounding errors and triangle law reasoning can be performed quite efficiently by GAPPA, which can generate formal proofs
- Polynomial bounds can be certified externally using sum-of-squares decompositions or other techniques, and merely verified in the prover.

Many of the more application-specific tasks might be better kept inside HOL Light, though automated in a more ‘push-button’ style.