1 Testing

Michael Parks has written a division-testing program, along with several papers describing the techniques used. These papers are recommended as a source reference for the following material.

Consider the following two sequences of numbers:

\[ n \in \{1, 2, 3, 4, 5, 6, \ldots, 8, 9, 10, \ldots, 12, \ldots, 16, \ldots\} \]

\[ m \in \{1, 2, \ldots, 800000\} \]

If a user computes in single precision

\[ q := m/n \text{ rounded} \]
\[ x := q \cdot n \text{ rounded} \]

will \( x = m \)? If arithmetic is done in IEEE 754 style, the answer is yes. But if the arithmetic is rounded differently, or if the radix is not 2, then for some \( m \) and \( n \), this identity will not hold.

How are the magic integers \( m \) chosen? All these numbers have only one or two bits set in their binary representation.

In the world of floating point computation, some people acquire the notion that none of the ordinary relationships hold. It is nice to know that this isn’t always the case. Problems in which exact arithmetic relationships fail to hold flood help desks; but surprisingly many properties are actually maintained in spite of roundoff. This poses a problem in system design: what properties of ordinary arithmetic ought to be maintained. For example, in spreadsheets we might want decimal floating point, so that people can actually see what they have. This is a marketing issue more than anything else. As system designers, we must understand our market and attempt to serve them well.

In the past, everyone was expected to be able to change spark plugs and fuel bowls, set the choke, etc. Car sales might be inhibited if these skills were still necessary to operate an automobile. However, with the advent of more reliable spark plugs and quality unleaded fuel, these skills are less likely to become frequently necessary. Even tire-changing has become a little-used skill as steel-belted radials make flats less frequent and AAA roadside assistance has reduced the need for drivers to be able to service problems themselves. The point of this is not to condemn modern drivers, but to illustrate the requirements for floating point arithmetic support. If we expected people to understand the details of floating point arithmetic, our expectations would be thwarted. We must make floating point arithmetic safe in the hands of people as we know them to be, which is “very much as God made them, or somewhat the worse,” to quote Mark Twain. This is not moraliastic in nature. It is a matter of marketing.

We now return to testing. We will present two interesting underlying principles:
1. Use of $p$-adic arithmetic

2. Seeking singularities

Mike Parks’ papers present implementation details.

Our presentation of $p$-adic arithmetic will differ somewhat from the standard presentation. A survey article on $p$-adic arithmetic appeared some while ago in the American Math Monthly; if you read that article, you will quickly realize that it sounds little like what we will talk about.

Consider a polynomial equation $p(z) = 0$, where $p$ has integer coefficients. For any reasonable probability distribution for the coefficients, this equation will not have integer roots. However, the equation 

\[ p(z) = 0 \quad (\text{mod } p^k) \]

is rather a different matter.

What has this to do with floating point? Suppose we wanted to compute a chopped square root. The interesting cases are when the rounded portion consists of mostly zeros (to check correct roundoff) or mostly ones (to make sure there is no overshoot).

The hope is that you will be able to partition your task into pieces, each of which can be tested more easily separately. If each module requires a relatively small number of inputs, then the test problem becomes additive in the number of modules, not multiplicative.

How do you test a module? To test accuracy, you can manipulate inequalities until you have a proof. Or you can do a battery test, trying all operands (or at least many operands) and comparing the results with the results from a more accurate program. Obtaining a more-accurate program which is slow enough that it would not be preferred to the program you have, yet fast enough to be of practical use in a battery test, is problematic. This is part of why competent testing requires a lot of smarts. But at least we know what we have to do.

The extrema in our error function occur at boundary points, at points in the interior where derivatives are zero, or at places where the derivative does not exist.

The functions we compute with are generally piecewise analytic. Recall that a (complex) analytic function is described by the Cauchy integral formula:

\[ f(z) = \frac{1}{2\pi i} \int_C \frac{f(w)}{w-z} \, dw \]

where $C$ is a simple closed contour. A singularity is a boundary point of the domain of analyticity. Singularities need not necessarily come only from the failure of differentiability; consider the function $\sqrt{z}$, for example.

Riemann’s principle states that, given the behavior of a function near its singularities, it is possible to reconstruct the rest of the function. This is provided the set of singularities is countable, except perhaps for slits which join branch points in a reasonable way. It doesn’t reckon with functions like

\[ \sum_{n=0}^{\infty} \frac{z^n}{n!} \]

which has singularities dense on the unit circle. But we only compute with piecewise analytic functions (piecewise because of tests and branches), so this does not bother us.

The strategy then, is to divide the data up into regions where it is analytic. Riemann’s principle indicates that if the program is correct on or near the singularities, it is correct. Identifying singularities is not trivial, though! Besides the intrinsic singularities in the functions involved, there may be additional singularities from your program, including some extraneous ones. Still,
by identifying these singularities we can reduce the dimensionality of the test set, rendering the problem of testing manageable.

We have already discussed how roundoff can only be amplified by nearness to a singularity. Singularities in inner modules for a composed program add to the singularities that must be considered. But what if we want to compute a very ill-conditioned function? The nearness to a singularity may explain why our computation fails to produce a reasonable answer, but that does not necessarily exculpate the code.

So if I want to test a chopped square root, I should test

- At values for which the square root is exact (singularities)
- Near these points

For a correctly rounded square root, we would test near mid-points. This is the principle which leads to solving equations like

\[ p(z) = \text{small integer} \mod 2^{\text{big}} \]

in order to find test arguments.

How do we solve such equations? The trick is known as Hensil lifting. Suppose we have a solution \( p(z_k) = 0 \mod 2^k \). How would we get a solution \( z_{k+1} \)? We know that

\[ p(z_k) = j_k \cdot 2^k = 0 \text{ or } 2^k \mod 2^{k+1} \]

An ordinary Taylor expansion tells us that for \( z_{k+1} = z_k + h \),

\[ p(z_k + h) = p(z_k) + hp'(z_k) + \ldots = 0 \mod 2^{k+1} \]

If we could write \( h = \theta 2^k \), then all the higher order terms would be zero modulo \( 2^{k+1} \), so the equation would become

\[ p(z_k + \theta 2^k) = p(z_k) + \theta 2^k p'(z_k) = 0 \mod 2^{k+1} \]
\[ = (j_k + \theta p'(z_k))2^k = 0 \mod 2^{k+1} \]

When \( p'(z_k) \) is odd, the equation

\[ j_k + \theta = 0 \mod 2 \]

obtained by dividing out the \( 2^k \) factor in the previous equation tells us that \( z_{k+1} \) will have \( j_k \) as its highest order bit (in position \( k \)), and the rest of the bits will be the same as \( z_k \). \( p'(z_k) \) is odd for most of our cases, at least when \( k \) is large enough. When \( k \) is small, it is fast enough to just test each number modulo \( 2^k \) in turn in order to find a root.

This provides sufficient theory to provide a running leap into Mike Parks’ stuff.

2 IEEE 754R and Projects

We know turn to some possible projects related to the IEEE 754R committee’s work. There are a number of interesting tasks, but some of these ideas are sufficiently open-ended that they might take much longer than we might initially think.
2.1 Gradual underflow

One possibility is to explain how to implement gradual underflow without forcing a trap. This would help us argue against introducing flush-to-zero mode. We would have to be able to both generate and perform arithmetic with subnormals without a problem. The 8087 had support for subnormal arithmetic with one extra exponent bit and a tag. Later models slowed down their hardware for backward compatibility with the 8087's design, unfortunately.

The RS6000 also did gradual underflow entirely in hardware. In order to support fused-multiply add, they already had sufficiently wide registers in order to normalize everything as much as necessary to do arithmetic with it. The PowerPC had similar support. However, the PowerPC machines have two extra pipeline stages to complete their gradual underflow support: one to normalize the operand, and one to denormalize the result if necessary. The combination bumps operation latency from 3 cycles to 4 cycles. The issue rate is not affected, but it would be preferable not to take even the latency hit.

We may be able to implement both prenormalization and denormalization in such a way as to allow ops to go through the pipeline at maximal speed by using the same hardware that supports cache misses. This should be sketched out, checked with competent computer architects to make sure it is realistic, and written up. The major contribution of the first IEEE 754 was to show that correct floating point algorithms were not really too expensive. We should continue that tradition of giving vendors example algorithms to imitate.

Note that handling gradual underflows in traps is horribly expensive. If you're on a DEC Alpha, though, it's even worse than you might think. To support gradual underflow on an Alpha, you have to insert trap barriers. These trap barriers are synchronization points, so they slow down execution even if the code generates no underflows. Trap barriers must occur frequently enough that no registers are re-used between traps, so this has the potential to slow things down a lot.

The guys who designed the Alpha came from the land of DEC VAX machines, where flush to zero was the thing you did, and you daren't do otherwise because it's so slow. They recently put in PAL code to fix things somewhat; underflow still traps, but now it traps to the PAL code rather than to user code. This fix was inspired by programs like ScalAPACK, which would sometimes crash on subnormals because no user underflow trap handler was defined. They also sped up trap handling some with trap anticipation logic, which works at the trap barrier to find out fast if there is no trap.

(Alex's comment: The policy in UltraSPARCs is to trap pessimistically for overflow; that is, when a huge number is generated that doesn't quite overflow, a trap may be generated anyhow. Competitors like to latch onto this. For Sun, the pessimistic trapping on overflow is more of a performance problem than gradual underflow traps.)

George Taylor's papers might be a good place to start when looking at how to do divides, but because of Moore's-law-driven technology trends, they are not all relevant. A lot of the lore on how to do fast gradual underflow is, unfortunately, lost.

2.2 Language support

Another possible project would involve demonstrating to language people how to properly support the standard. We would want to give guidelines for when not to optimize, and give programmer locations to explicitly permit certain types of optimization locally. And we need to be able to test flags!

Alex commented that it will be tricky to make this not impede performance some. Sun's noncompliant "fsimple" mode goes significantly faster than the IEEE mode, and everyone has a
similar mode.

Kahan replied that when the standard was designed, the people involved worried a lot about performance. For every step, they found ways to implement their choices without increasing cost or decreasing performance more than 5 to 10%. But confidentiality agreements kept them from later divulging a lot of this information.

A possible approach would be to choose C or C99 as an example, and show how to get the compiler to support the IEEE standard using a language model worth supporting. For example, the compiler should use K&R style double precision intermediates on SPARCs, and double extended intermediate variables on an Intel. Quadruple precision should probably not be included into the mix unless it is explicitly requested, since even the fastest implementations of quadruple precision (the IBM 390 may have the fastest one right now) are not that fast.

We would then need to find one compiler house which was willing to do things right. If we do that, though, we should take care to find one that looks like it will not go under. The Borland compilers, for instance, did a pretty good job with their floating point... but they have since gone under.

2.3 Rules of thumb

We also need a reasonably written presentation of new rules of thumb for casual floating point users who do not want to take a numerical analysis course. The rules are presented at the end of the Java floating point document on Kahan’s web page, but they should be clarified and put into a form that a general audience can easily digest and follow. Additionally, we will need a document describing the rules that compiler writers and language designers should follow in order to make the programmer rules of thumb maximally successful.