Implementation Specific Verification of Divide and Square Root Instructions

Elena Guralnik, Ariel J. Birnbaum, Anatoly Koyfman
IBM Haifa Research Lab
University Campus, Carmel Mountains
Haifa, 31905, Israel
{elenag,arielb,anatoly}@il.ibm.com

Avraham Avi Kaplan
Unknown Affiliation
Unknown Address
Unknown Email

Abstract

Floating point operations such as divide and square root are typically implemented in microcode rather than with dedicated logic. Bugs in these operations that were missed by generic, black-box verification tools, were analyzed. This led to the conclusion that the corner cases, in addition to being implementation dependent, could not be characterized in terms of special input or output values in a straightforward manner.

Fortunately, many of those corner cases can be easily generalized for many known implementations. The typical implementation uses a known iterative approximation algorithm, such as the Newton-Raphson method, to calculate the desired result; thus, it is sufficient to produce the corner cases associated with the specific algorithm.

We investigated the following problem: given a binary floating point operation from the list above, the algorithm, the iteration number, and an interval, find random inputs for the operation that, after the requested iteration, yield a relative error within the specified interval. This paper describes an algorithm that solves this problem; it also describes how to randomize it to improve verification power.

This algorithm was implemented in a floating-point test generator and is currently being used to verify the floating-point units of several processors.

1 Introduction

Verification of the floating point unit (FPU) hardware implementation is known as an intricate problem. The numerous corner cases of the vast test space, coupled with the complexity of the implementation of floating point operations, turn the FPU verification effort into a truly unique challenge in the field of processor verification.

Both formal methods and simulation methods have been developed to deal with this challenge. When dealing with floating-point verification by simulation, there is a practically unbounded number of different calculation cases to test. In practice, simulation is performed only on a very small portion of the existing space. The rationale is that one acquires confidence in the design's correctness by running a set of tests that are assumed to be representative samples of the entire space.

Test suites and coverage models have been developed for this purpose (for example [1], [8]. These suites and models define interesting cases for the verification objectives based on an FP standard (usually IEEE 754 [2]) and do not cover many of the corner cases generated by the internal implementation of the FP unit. The formal methods provide a good solution for the verification of the actual implementation of FP units, but they are restricted by the unit's complexity. A typical verification process uses both methods to achieve a comprehensive solution. Unfortunately, certain implementation areas are not sufficiently verified with the existing methods.

The usual approach in simulation based verification is to treat the implementation as a black box; inputs are driven into the design and outputs are checked against the specification—in our case, the IEEE standard for binary floating point arithmetic [2]. In general, this provides excellent coverage of a wide spectrum of corner cases, involving inputs, outputs, and intermediate results.

However, as design complexity increases, the picture begins to shift. An analysis of certain floating point bugs missed by generic verification techniques led us to conclude that the scenarios that reveal these problems could not be characterized in terms of the visible inputs and outputs; in these scenarios, there is a need for white-box inspection of the actual implementation.

These bugs occurred in operations implemented via iterative numerical methods. For example, in the square root computation, the result after the first iteration was rounded in the wrong direction, leading to a different result than specified. The trigger for this problem was an extremely small relative error after the first iteration. This very rare

event, not identified with any common corner case, was not generated by an IEEE-oriented test generator. Bugs with similar characteristics also surfaced in the divide operation. In all of these, the recurring pattern was the appearance of near-extreme relative errors in the run of the algorithm. These values cannot be reached through constraints on inputs and outputs alone.

The elusiveness of these bugs is by no means exclusive to simulation based methods. Standard model checking technology is not equipped to deal with microcoded instructions, the chief reason being the so-called "state explosion" due to the sheer complexity of their implementation. An alternative approach exists, based on theorem proving technology, to verify the divide instruction [9]. While attractive, the assumptions on which this technique relies require the implementation to adhere to a specific form, expressible in terms of well-defined operations whose implementation is assumed to be correct. Hardware implementations, such as P6 ([10]), often deviate from this form (e.g., as an optimization) and render this technique inapplicable. Simulation based methods must therefore step up to the challenge.

The verification difficulties posed by these instructions have been recognized by other researchers as well. W. Kahan in [4] and McFearin and Matula in [5] deal with corner cases of the outputs of square root and divide, respectively. These approaches, however, do not address problems that emerge in specific implementations of these operations.

Our approach is to develop the ability to express constraints on the relative error, in an attempt to cover the corner cases noted above. From deeper analysis of the bugs we learned that the desired corner cases can be described as intervals around extreme values of the relative error. We thus proceeded to study the following problem:

Given a binary floating point operation (divide, square root, reciprocal estimate, or reciprocal square root estimate), the algorithm, the iteration number, and an interval, find random inputs for the operation that, after the requested iteration, yield a relative error within the specified interval.

We developed a randomized algorithm to solve this problem and implemented it within *FPgen*, an automatic floating point test generator [6]. FPgen receives as input the description of a floating point coverage task and uses various algorithms to produce a random test that covers this task. A coverage task is defined by specifying a floating point instruction and a set of constraints on the inputs, the intermediate result, or the final result.

In addition to implementing the algorithm, we provide several sets of coverage tasks that target the new corner cases. These tasks cover very small and very large (in absolute value) positive and negative relative errors for every iteration of the underlying approximation algorithm.

The remainder of this paper is built as follows: In Section 2, we provide an overview of the algorithms used to implement the chosen instructions, present two motivating examples, and define related terminology. In Section 3, we present the problem more formally and outline our general approach. In Section 4, we present the algorithm for solving constraints on relative errors and describe the coverage tasks that target the related corner cases. Finally, Section 6 includes a summary of the results and some suggestions for future work in this area.

2 Background

2.1 Hardware Implementation of the Instructions

In contrast to other floating point operations, the instructions discussed in this paper are not implemented directly as a logic circuit. Rather, they are translated by the processor into a sequence of operations (known as "microcode") yielding the desired result.

What we present now is a simplified view of the algorithms described by Agarwal *et al.* in [3]. The unifying theme is the use of a cheap initial approximation that is iteratively improved upon by introducing a correction term in each stage. In a bounded (and small) number of iterations, the error of the intermediate result drops below the required threshold. At this point, the correct answer is guaranteed to be delivered after rounding.

2.1.1 Reciprocal

If b is anything other than a finite, normal, nonzero number, $\frac{1}{b}$ can be computed trivially—denormalized numbers will overflow while all other cases are singular. Assuming that b is finite, normal, and nonzero, we derive $\frac{1}{b}$ as follows:

$$b = (-1)^s \cdot 2^e \cdot (1+f) \qquad (0 \le f < 1)$$
$$\tfrac{1}{b} = (-1)^s \cdot 2^{-e} \cdot \tfrac{1}{1+f} \qquad (\tfrac{1}{2} < \tfrac{1}{1+f} \le 1)$$

If $f = 0$:

$$\tfrac{1}{b} = (-1)^s \cdot 2^{-e} \cdot (1+0)$$

Otherwise:

$$\tfrac{1}{b} = (-1)^s \cdot 2^{-e} \cdot (\tfrac{1}{2} + f') \qquad (0 < f' < \tfrac{1}{2})$$
$$\tfrac{1}{b} = (-1)^s \cdot 2^{-e-1} \cdot (1 + 2f') \qquad (0 < 2f' < 1)$$

The sign and exponent of $\frac{1}{b}$ can thus be determined with little effort; it remains only to obtain the fractional part. We achieve this by iterative refinement of an initial approximation, obtained by extracting a linear function from a lookup table and applying it to b's fractional part[1]. At each refinement step, a correction term is added to the approximation, based on an estimate of the relative error.

[1] There is an inaccuracy here that we will address later.

2.1.2 Divide

As with reciprocal, we can assume the numerator a and denominator b are finite, normal and nonzero. As we show in Section 2.3 below, given an approximation q of $\frac{1}{b}$, we can approximate a by $q \cdot a$ with the same relative error. We note that additional rounding errors can appear from the multiplication step. However, Agarwal *et al.* show in [3] that these errors are negligible. Therefore, we can use the same method (and indeed, the same lookup table) mentioned above for the reciprocal.

2.1.3 Reciprocal Square Root

To compute $\frac{1}{\sqrt{b}}$, we follow a procedure and analysis similar to that shown for $\frac{1}{b}$, up to the fact that for odd exponents of b we need to multiply the fractional part by a constant value. We can factor the multiplication into the computation of the fractional part using a different approximation table for even and odd exponents. In either case, the relative error is not significantly affected. It is worth noting that the denormal case can be computed by adjusting the fractional and exponent parts appropriately.

2.1.4 Square Root

We can express $\sqrt{b}$ as $b \cdot \frac{1}{\sqrt{b}}$. This allows us to use the same reduction we used to compute $\frac{a}{b}$ in terms of $\frac{1}{b}$. The same considerations apply.

The input domain is divided into sectors (indexed by the higher bits of the fractional part) and the approximation parameters are precomputed for each sector. A different linear approximation is used for each sector; in other words, the approximation function is *piecewise* linear. See Figure 1 for an illustration.

As an additional detail, note that the approximation and correction terms are derived from a power series expansion of the function being computed.

This algorithm is used, with small variations, across a range of hardware designs.

2.2 Motivating Examples

Over the years we have seen several bugs in the divide and square root instructions that have escaped the verification efforts. These bugs were not algorithmic problems—the underlying algorithms worked perfectly—but rather implementation problems. We present here two examples. Neither bug was detected until late in the verification process; both were detected by sheer chance. In neither case did the usual methods and test cases described in the literature cover the bug.

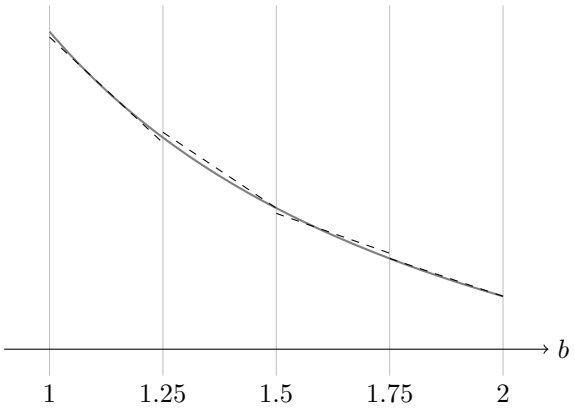


Figure 1. Piecewise Linear Approximation of $\frac{1}{b}$, with the input domain $[1, 2)$ divided into 4 sectors. The grey curve shows the actual function, the dashed black lines show the linear approximation for each sector. (Plot is schematic only.)

2.2.1 Example 1

In the square root operation, the result was rounded in the wrong direction. In the implementation, the desired rounding mode is converted internally to one of three modes (To Nearest, Truncate, or Increment) depending on the sign of the final result. Because of an error, the sign was not taken invariably to be positive but determined by the sign of the approximation after the first refinement step. In very rare cases (estimated to be considerably less than 2^{-15} of the input space) this value was negative; when this happened the result was rounded in the wrong direction. Such negative values can occur only when the relative error is close to its maximal positive value.

2.2.2 Example 2

Wrong fraction produced in the divide operation. To increase precision in the computation of the correction term, the implementation captured the complete result of a multiply-add operation before normalization. However, when the addend of this operation is zero, the lowermost bit of the result (which is never visible after normalization) is set to 1. This caused a bit in the fractional part of the correction term to be set erroneously; the error propagated to the final result. The triggering condition occurred only when the uppermost half of the relative error's fractional part was zero; for the case in point this is equivalent to the relative error itself being close to zero.

We observed that the conditions discussed above are best described in terms of the relative error of the approxima-

tion at some point in the computation. Despite significant effort, we were unable to come up with a concise transformation from this form of problem definition to the usual ones in terms of input and output values for a relevant operation. The major difficulty was the need to address the implementation dependent initial approximation tables. We also observed, as explained above, that the test cases commonly mentioned in the literature do not cover these scenarios. For example, hard to round cases explored by Parks ([12]), Matula ([5]), and Kahan ([4]) do not produce the desired corner cases for the values of the intermediate estimations. The formal techniques presented by Russinoff ([13]) and Harrison ([14]) do not provide the desired cure either. Their techniques were successfully applied to the systems where the operations are implemented "without tricky and time-consuming bit-twiddling" *(sic)*. In such systems, divide and square root are implemented as a sequential invocation of simple, well-defined, and previously verified operations, whence correctness of the implementation can be proven mathematically. Unfortunately, many existing implementations "misbehave" in this respect by manipulating and examining internal bits of the arithmetic hardware (e.g., multipliers), rendering a clear-cut description as the one in [14] impossible.

For the reasons stated above, we decided to address the problem by extending the capabilities of our verification tools and developed the solution described herein.

2.3 Notation

In the following discussion, we use $\widetilde{q}$ to denote an approximation of a quantity q. Given an approximation $\widetilde{q}$, we define its *signed relative error* as $e_q \triangleq \frac{q-\widetilde{q}}{q}$. Equivalently:

$$e_q = \frac{q-\widetilde{q}}{q} = \frac{q}{q} - \frac{\widetilde{q}}{q} = 1 - \frac{1}{q} \cdot \widetilde{q}$$

If we compute q by an iterative process, we can refer to the approximation at the i-th iteration as $\widetilde{q}^{(i)}$ and to its signed relative error as $e_q^{(i)}$. We denote the initial approximation as iteration 0.

Let $q = rs$ and $\widetilde{q} = r\widetilde{s}$. Then:

$$e_q = \frac{q-\widetilde{q}}{q} = \frac{rs-r\widetilde{s}}{rs} = \frac{s-\widetilde{s}}{s} = e_s$$

We make use of this fact in the following analysis.

Given a piecewise linear approximation $\widetilde{q(b)}$, we refer to each of the "pieces" (viz. the linearity intervals of $\widetilde{q(b)}$) as a *sector*.

3 Problem Definition

Given an iterative method (as described above) to compute $q(b)$ (either $\frac{1}{b}$, $\frac{1}{\sqrt{b}}$, or $\frac{\sqrt{2}}{\sqrt{b}}$), an iteration number i, and an interval $[L, H]$, find $b \in [1, 2)$ such that $e_{q(b)}^{(i)} \in [L, H]$.

To see that this definition is sufficient, first we note that by the argument in the preceding section (Section 2.3) $e_{\frac{a}{b}} = e_{\frac{1}{b}}$ and $e_{\sqrt{b}} = e_{\frac{1}{\sqrt{b}}}$, for relevant (nontrivial) inputs. We can therefore limit our discussion to the reciprocal and reciprocal square root operations.

Moreover, by the same reasoning, the sign and exponent do not affect the relative error. In other words, if b' has the same fractional part as b, $e_{q(b')} = e_{q(b)}$. We can thus focus on the fractional part alone.

Since we assume b to be normal, its implicit bit is always 1. In other words, b is of the form $(-1)^s \cdot 2^e \cdot (1+f)$, where $1 + f \in [1, 2)$. From the paragraph above, we can assume $s = 0$ and $e = 0$ without changing the solution. Henceforth, without loss of generality, we assume b to be within $[1, 2)$—that is, b is normal and positive, and its exponent is zero.

As for $\frac{1}{\sqrt{b}}$, in principle, odd and even exponents behave differently and we should consider the interval $[1, 4)$ instead of $[1, 2)$. However:

$$\begin{aligned} b &= 2^{2e-1} \cdot (1+f) \\ \tfrac{1}{\sqrt{b}} &= 2^e \cdot \sqrt{2} \cdot \tfrac{1}{\sqrt{1+f}} \\ e_{\frac{1}{\sqrt{b}}} &= e_{\frac{\sqrt{2}}{\sqrt{1+f}}} = e_{\frac{\sqrt{2}}{\sqrt{2^{2e}\cdot(1+f)}}} \end{aligned}$$

If we recast the problem as $q(b) = \frac{\sqrt{2}}{\sqrt{b}}$, which is what in fact we do when we use a different initial approximation, we see how the reduction to the interval $[1, 2)$ still suffices. Another way of saying this is that we *do* consider the interval $[1, 4)$ by reducing the $[3, 4)$ half into the $[1, 2)$ one.

While we could have defined the problem in terms of units in the last place as opposed to relative error, the latter is more natural in this case since the algorithms and error cases as exposed above are expressed in these terms.

4 Solution Method

The core of our method is to search iteratively for a solution b to the constraint $e_{q(b)}^{(i)} \in [L, H]$. To find such a value, we first use precalculated data to select a subinterval of the domain where a solution is assumed to exist; then we iteratively halve the search interval until a solution is found.

The following section describes our method in more detail. For the sake of clarity, we first solve the problem over a continuous domain, where the approximation $\widetilde{q}$ is linear over the entire domain. We then show how to extend the approach to the actual problem at hand.

We separate the problem into two cases: the initial approximation alone (before any refinement) and the subsequent iterations.

Require: $x \in [X, Y] \wedge \phi(x) \leq L$
Require: $y \in [X, Y] \wedge \phi(y) \geq H$
Ensure: $\phi(r) \in [L, H]$

$r \leftarrow \frac{x+y}{2}$
loop
  if $\phi(r) < L$ **then**
    $x \leftarrow r$
  else if $\phi(r) > L$ **then**
    $y \leftarrow r$
  else[Solution Found]
    return r
  $r \leftarrow \frac{x+y}{2}$

Figure 2. Interval Halving Method

4.1 Initial Approximation

The functions under discussion are all of the form βb^α (e.g $\frac{1}{\sqrt{b}} = b^{-\frac{1}{2}}$. Also $\widetilde{q(b)} = \beta(Ab + C)$. If we define $\phi(b) \triangleq e_{q(b)}$:

$$\begin{aligned}\phi(b) &= \tfrac{q(b)-\widetilde{q(b)}}{q(b)} \\ &= \tfrac{\beta b^{-\alpha}-\beta(Ab+C)}{\beta b^{-\alpha}} \\ &= 1 - b^\alpha \cdot (Ab + C)\end{aligned}$$

Therefore ϕ is a smooth function over the positive reals, in particular over $[1, 2)$. Also:

$$\begin{aligned}\phi'(b) &= -(\alpha b^{\alpha-1}(Ab + C) + Ab^\alpha) \\ &= -b^{\alpha-1}((\alpha + 1)Ab + \alpha C)\end{aligned}$$

This function has at most one root over the positive reals; this means ϕ has at most one critical point over this domain. Therefore, in any given interval we can easily find the extrema (maximum and minimum) of ϕ.

It suffices then to solve the following problem: given a smooth function ϕ over an interval $[X, Y]$, find b in $[X, Y]$ such that $\phi(b) \in [L, H]$. This can be solved in a finite number of iterations by the algorithm shown in Figure 2, known in the literature as the *Interval Halving Method* or *Binary Search*.

The algorithm's correctness is ensured by the invariant that a solution exists between both endpoints of the search (x and y in the figure). From ϕ's continuity and the Intermediate Value Theorem it is straightforward to see that the invariant is kept. We can always provide an initial guess for x and y as

$$x \leftarrow \operatorname{minarg}_{[X,Y]} \phi; \quad y \leftarrow \operatorname{maxarg}_{[X,Y]} \phi$$

In addition, when the constraint allows so, we can provide a different guess in order to start with a wider search interval and capture more potential solutions. For example, if we know $\phi(X) \leq L$ we can guess $x \leftarrow X$ instead of $\operatorname{minarg}_{[X,Y]} \phi$.

When the initial approximation is piecewise linear, we can extend our solution by dividing the input domain into the sectors described in Subsection 2.3. Since q and $\widetilde{q}$ (and therefore ϕ) are known beforehand, we can precalculate the bounds and extrema of ϕ for each sector.

4.2 Randomization

Although the algorithm above solves the problem, it does so deterministically, i.e., given the same constraint it will always produce the same solution. This implies that we obtained a single test value where we could obtain many, thus losing verification potential. We overcome this obstacle by choosing the initial guess (first $r \leftarrow \frac{x+y}{2}$ line in Figure 2) randomly within the interval (x, y) instead of a fixed choice of $\frac{x+y}{2}$. The invariant is still kept, thus the algorithm remains correct. In the same manner, we can randomize the choice of r from (x, y) at the end of every iteration.

4.3 Iterative Refinement

Most iterative methods used in practice are based on either the Newton-Raphson method, Taylor series expansions or Chebyshev series expansions. In these methods, the approximation is refined at each step by adding a correction term C, which can be expressed as a product of the previous approximation and a rational function F of the error at that step. In other words:

$$\widetilde{q}^{(i+1)} = \widetilde{q}^{(i)}(1 + F(e_q^{(i)}))$$

Hence follows that:

$$\begin{aligned}e_q^{(i+1)} &= 1 - \tfrac{\widetilde{q}^{(i+1)}}{q} \\ &= 1 - \tfrac{\widetilde{q}^{(i)}(1+F(e_q^{(i)}))}{q} \\ &= 1 - \tfrac{\widetilde{q}^{(i)}}{q}(1 + F(e_q^{(i)})) \\ &= 1 - (1 - e_q^{(i)})(1 + F(e_q^{(i)}))\end{aligned}$$

which means the relative error itself can be characterized as a rational function ψ of the error in the previous iteration. By composition, we obtain that $e_q^{(i)}$ can be expressed as a smooth function of the input b. Therefore, known numerical techniques (such as the Newton-Raphson method) can be applied to precompute the extremal values of $e_q^{(i)}$ in each sector. Knowing these values, as seen in Section 4.1, allows us to solve a range constraint on $e_q^{(i)}$ by the same Halving Method shown above.

As an example, let us observe a derivation of F for two implementations presented by Agarwal *et al.* in [3].

4.3.1 Example a. Double Precision Divide

Initially, $\frac{a}{b}$ is approximated by $\widetilde{q}^{(0)}$ with relative error $e_q^{(0)} = 1 - \frac{b}{a}\widetilde{q}^{(0)}$. To refine the approximation, a correction term $C = (a - b\widetilde{q}^{(0)})\frac{\widetilde{q}^{(0)}}{a}P(e_q^{(0)})$ is added, where $P(x) = \sum_{i=0}^{5} x^i$. Thus:

$$\begin{aligned} C &= (a - b\widetilde{q}^{(0)})\tfrac{\widetilde{q}^{(0)}}{a}P(e_q^{(0)}) \\ &= \tfrac{a-b\widetilde{q}^{(0)}}{a}\widetilde{q}^{(0)}P(e_q^{(0)}) = \widetilde{q}^{(0)}(1 - \tfrac{b}{a}\widetilde{q}^{(0)})P(e_q^{(0)}) \\ &= \widetilde{q}^{(0)}e_q^{(0)}P(e_q^{(0)}) \end{aligned}$$

Thus by defining $F(x) \triangleq xP(x)$ we obtain $C = \widetilde{q}^{(0)}F(e_q^{(0)})$ as desired.

4.3.2 Example b. Double Precision Square Root

Initially, $\sqrt{b}$ is approximated by $\widetilde{q}^{(0)}$ with relative error $e_q^{(0)} = 1 - \frac{\widetilde{q}^{(0)}}{\sqrt{b}}$. To refine the approximation, a correction term $C = \widetilde{q}^{(0)}eP(e)$ is added, where e is defined as $1 - \frac{b}{(\widetilde{q}^{(0)})^2}$ and P is a Chebyshev approximation polynomial[2]. Note that:

$$\begin{aligned} e &= 1 - \tfrac{b}{(\widetilde{q}^{(0)})^2} = 1 - (\tfrac{\sqrt{b}}{\widetilde{q}^{(0)}})^2 = 1 - (\tfrac{\widetilde{q}^{(0)}}{\sqrt{b}})^{-2} \\ &= 1 - (1 - e_q^{(0)})^{-2} \end{aligned}$$

Thus, if we define $Q(x) \triangleq 1 - (1 - x)^{-2}$, we obtain $e = Q(e_q^{(0)})$. If we also define $F(x) \triangleq Q(x)P(Q(x))$, we obtain $C = \widetilde{q}^{(0)}F(e_q^{(0)})$ as desired.

4.4 Discrete Domain

Floating-point arithmetic operates on a discrete, finite domain. In such a domain, it is possible that $\phi(x) < L$ and $\phi(y) > H$ yet there is no z between x and y such that $\phi(z) \in [L, H]$. Therefore, we have no way of ensuring beforehand that a solution exists in a given interval.

On the other hand, the same finiteness allows us to adapt the Halving Method in such a manner that when the search endpoints x and y are contiguous, the algorithm halts. This is guaranteed to happen after a bounded number of iterations. Furthermore, if a solution exists by our invariant we do not discard it in any search step. Thus, with a minor modification, our algorithm also works for the discrete case.

In practice, $[L, H]$ is wide enough so that if a solution exists in the continuous case, one also exists for the discrete case. Our algorithm is then guaranteed to find it.

[2]The term has a different form in the referenced paper. This is done to account for rounding errors in its computation, which do not concern us here.

4.5 Nonlinear Initial Approximation

There is one complication we have not mentioned yet: in practice, not only is the initial approximation defined differently for each sector, but it is also computed while ignoring some of the lower order bits of b. Effectively, this makes the approximation within each sector piecewise constant as opposed to linear (i.e., a step function); the error function changes accordingly as shown in Figure 3. Moreover, whereas the small number of sectors made the problem manageable, it now becomes impractical to precalculate the relevant parameters and divide each sector into subintervals, as we did above.

In theory, this destroys our assumption that ϕ is monotonic (or even continuous), introducing the possibility that our algorithm will fail when a solution exists; this will happen if at some point we violate our method's invariant; namely, the assumption that a solution exists between the two search endpoints.

Let us examine the structure of the error function more closely (see Figure 3(b)). Observing this, we note that the only way we can violate the invariant is by having the search endpoints lie on two adjacent monotonicity intervals, in an order such that all the solutions in both intervals fall outside the endpoints. We also make the following observations:

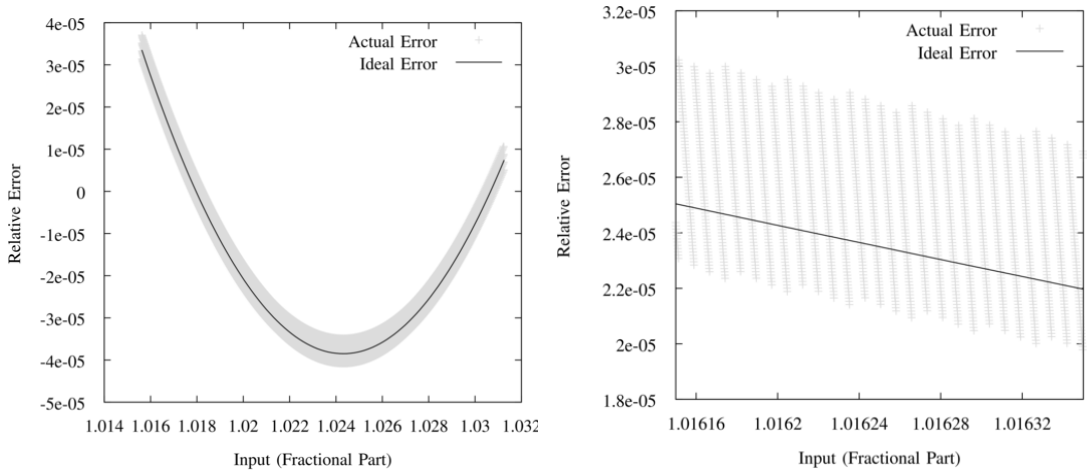
- In intervals where ϕ shows a decreasing trend (that is, where the ideal error is decreasing) it is impossible to reach such a failure state.

- In intervals where ϕ shows an increasing trend, the number of initial guesses that lead to a solution is (at least) the same order of magnitude as that of guesses that lead to failure. In this case, we can repeat the method with a different guess upon failure, thus decreasing the failure probability exponentially until it becomes negligible. In our experiments, three attempts were enough to yield consistent success.

From these we can conclude that our algorithm will, with a high probability, succeed in finding a solution to the constraint.

4.6 Coverage Tasks

In addition to the solver method exposed above, we provide a set of coverage tasks to target the motivating problem. Each such task defines an interval for the relative error in a given iteration, for a given operation, on a certain precision. (There are differences in the treatment for these that we address below.) Every interval is defined by the following variables:



(a) Relative error over the entire sector.

(b) Relative error over a subinterval of the sector. Note the density of the monotonicity subintervals.

Figure 3. Relative error for an approximation of reciprocal in a sector of the interval $[1, 2)$. The solid line indicates the "ideal" error of a linear approximation. The grey cross marks indicate the error of the approximation used by an actual implementation.

4.6.1 Sign

This variable defines whether the relative error should be positive or negative. We do not allow mixed-sign intervals; these are easily expressed as two single-signed ones.

4.6.2 Limit Point

In our models, the interval is always a neighborhood of an extremal point of the relative error, either maximal or minimal. Note that, depending on the sign, one of these points can be zero. For example, a task with a positive sign and a minimal limit point yields a search interval of the form $[0, \delta]$. The width of each interval was determined empirically as the smallest width for which our method reliably yielded a solution.

4.6.3 Search Scope

This variable defines whether to search for a solution globally (across the whole $[1, 2)$ domain) or locally (within an arbitrary sector). In a local search, we first choose a sector and then interpret the limit point as an extremal value within the sector. We then solve for the resulting interval. In a global search, the limit point is the global extremum. We then choose a sector for which the interval has a solution and proceed to search.

Figures 4 and 5 illustrate the different coverage task types.

For each combination of precision, iteration, and operation, we define tasks comprising all four combinations of limit point and search scope. In almost all cases, we define both positive and negative tasks.

For single precision we only define tasks for the initial approximation— after the first refinement step, the error is guaranteed to be below the significant threshold; errors at this stage can be covered by regular output constraints.

For double precision we define tasks for the initial approximation and first refinement step. After the first step, the relative error is always positive, so we do not define negative tasks in this case.

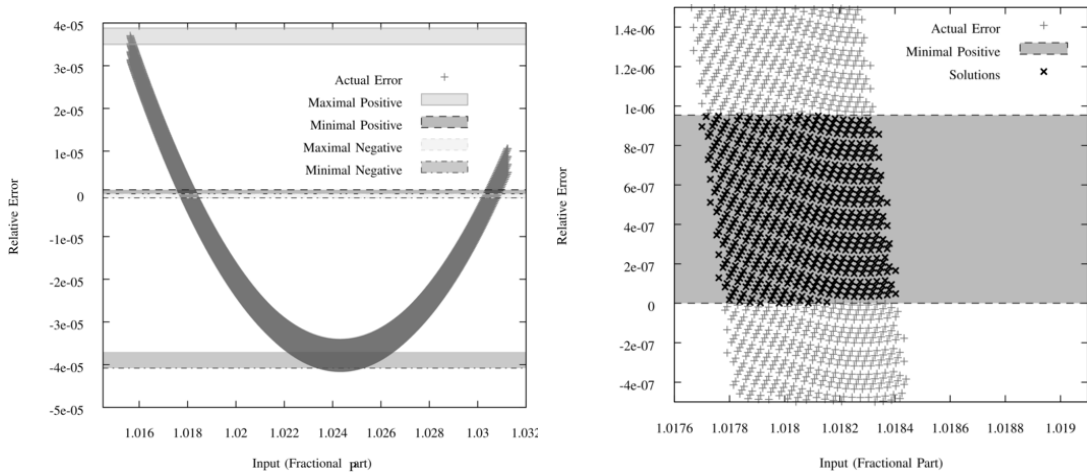
For extended precision, we exploit the fact that the initial approximation and first refinement step are identical to double precision, and define the same set of tasks.

4.7 Deployment

The solution method and coverage tasks described above have been implemented in FPgen and deployed to verification teams working on a large variety of designs. They are now actively integrated in the ongoing verification flow of floating point units, systematically targeting the problem area in the microcode implementations. In particular, the examples we brought forth in Section 2.2 were successfully covered.

5 Implications on Verification

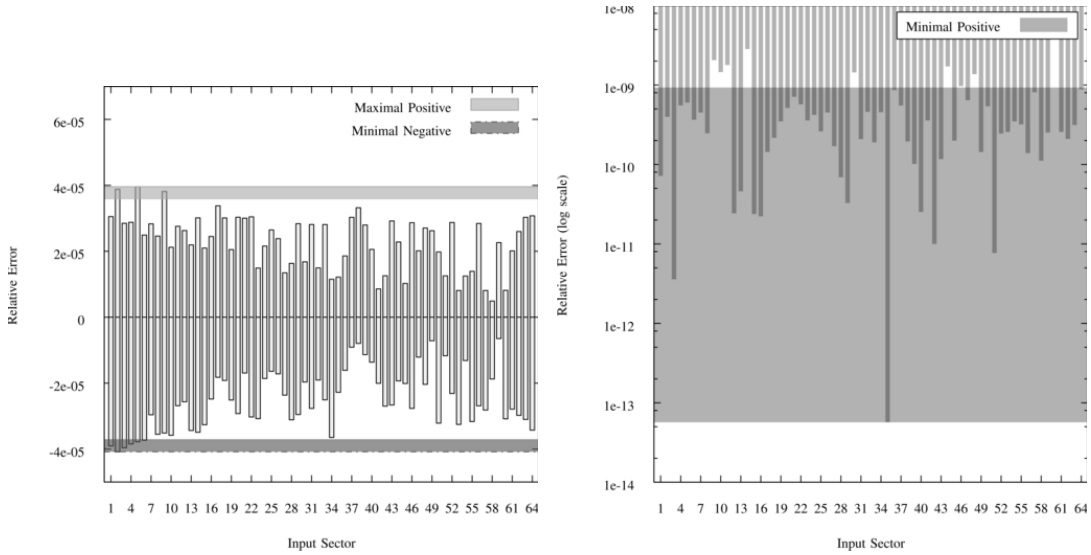
Beyond the problem at hand and the solution we present, there is an important conclusion to be drawn from our experience. As discussed, for complex operations such as divide



(a) Local tasks over the entire sector. A gap exists between the "Minimal Positive" and "Maximal Negative" intervals.

(b) Enlarged view of the "Minimal Positive" model. Points that solve the constraint are marked as black Xs.

Figure 4. Local coverage tasks for reciprocal in a sector of the interval $[1, 2)$. The grey marks show the approximation error at each point. The horizontal bands indicate the intervals of the signed relative error for different combinations of limit point and sign.



(a) "Maximal Positive" and "Minimal Negative" global models, shown over all input sectors.

(b) Enlarged view of the "Minimal Positive" global model. Relative error is plotted in a logarithmic scale. "Maximal Negative" behaves analogously.

Figure 5. Global coverage tasks for reciprocal over the interval $[1, 2)$. The vertical rectangles indicate the error range for each sector. The horizontal bands indicate the intervals of the signed relative error for different combinations of limit point and sign. Note that the bands intersect only part of the rectangles.

and square root it becomes necessary to descend to a lower level of abstraction and "peek inside the black box" in our search for hard-to-hit bugs. To overcome this verification shortcoming, we need tools and constructs to identify, describe, and target the gaps between the abstract algorithms and their actual implementations. The work we present here is a first step towards filling this hole. The coverage models we describe attack the problems that motivated our research; we believe other such problems can be identified by a skilled verification engineer.

6 Summary and Future Work

We identified the need for implementation oriented testing of instructions whose implementations in FP units involve approximation algorithms. Our focus was on divide, square root, reciprocal estimate, and reciprocal square root estimate operations. We defined corner cases specific to the implementation of these instructions and created coverage models for these corner cases. We proposed algorithms that solve such constraints for the above instructions. These algorithms were implemented in the framework of FPgen, a test generator for floating-point data, and found to be effective in generating the desired corner scenarios. We have used FPgen to generate the described coverage models.

We recognize several directions where we can improve and extend our work. First, for extended precision, there is a variant algorithm that uses a different approach after the first refinement step. A solution for this setting is beyond our solver's current capabilities. Second, we see a strong verification need to extend the solvers ability in such a way that constraints placed on both the input and relative error can be solved.

We would also like to verify similar cases for decimal floating-point operations [11]. Further complications emerge in this domain, such as multiple representations of the same value with different exponents. These have to be addressed in order to adapt the method successfully.

Finally, we would like to develop a comprehensive treatment of the discontinuities in the relative error for piecewise continuous approximations.

References

[1] Nelson H. F. Beebe's IEEE754 Floating-Point test software [Online], Available: `http://www.math.utah.edu/beebe/software/ieee`.

[2] *IEEE Standard for Binary Floating-Point Arithmetic*, An American National Standard, ANSI/IEEE Std 754, 1985.

[3] R. C. Agarwal, F. G. Gustavson and M. S. Schmookler, "Series Approximation Methods for Divide and Square Root in the Power3™ Processor", in *Proc. 14th IEEE Computer Arithmetic Symp. (ARITH '99)*, 1999, pp. 116-123.

[4] W. Kahan, *"A Test for Correctly Rounded SQRT"* [Online], Available: `http://www.cs.berkeley.edu/wkahan/SQRTest.ps`, 1996.

[5] L. D. McFearin and D. W. Matula, "Generation and Analysis of Hard to Round Cases for Binary Floating Point Division", in *Proc. 15th IEEE Computer Arithmetic Symp. (ARITH 15)*, 2001, pp 119-127.

[6] M. Aharoni, S. Asaf, L. Fournier, A. Koyfman, and R. Nagel, "FPgen—A Test Generation Framework for Datapath Floating Point Verification", presented at *IEEE 2003 Int. High Level Design Validation and Test Workshop (HLDVT03)*.

[7] R. Ziv, M. Aharoni, and S. Asaf, "Solving Range Constraints for Binary Floating-Point Instructions", in *Proc. 16th IEEE Computer Arithmetic Symp. (ARITH 16)*, 2003, pp. 158-164.

[8] "Floating-Point Test Suite for IEEE 754R Standard" [Online], Available: `http://www.haifa.il.ibm.com/projects/verification/fpgen/ieeets.html`.

[9] E. M. Clarke, S. M. German and X. Zhao, "Verifying the SRT Division Algorithm Using Theorem Proving Techniques", *Formal Methods in System Design*, vol. 14, issue 1 pp. 7-44, 1999.

[10] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener, "P6 Binary Floating-Point Unit", in *Proc. 18th IEEE Computer Arithmetic Symp. (ARITH 18)*, 2007, pp. 77-86.

[11] M. Cowlishaw, E. Schwarz, R. Smith, and C. Webb. "A Decimal Floating-Point Specification", in *Proc. 16th IEEE Computer Arithmetic Symp. (ARITH 15)*, 2001, pp. 147-154.

[12] M. Parks, "Number-Theoretic Test Generation for Directed Rounding", *IEEE Trans. Comput.*, vol. 49, issue 7 pp. 651-658, 2000.

[13] D.M. Russinoff, "A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode", *Formal Methods in System Design*, vol. 14, issue 1 pp. 75-125, 1999.

[14] J. Harrison, "Formal Verification of IA-64 Division Algorithms", *Lecture Notes In Computer Science*, vol. 1869, pp. 233-251, 2000.