Simulation Based Verification of Floating Point Division

Elena Guralnik, Merav Aharoni, Ariel J. Birnbaum, Anatoly Koyfman

Abstract—Floating point division is known to exhibit an exceptionally wide array of corner cases, making its verification a difficult challenge. Despite the remarkable advances in formal methods, the intricacies of this operation and its implementation often render these inapplicable. Simulation based methods remain the primary means for verification of division.
FPgen is a test generation framework targeted at the floating point datapath. It has been successfully used in the simulation based verification of a variety of hardware designs. FPgen comprises a comprehensive test plan and a powerful test generator. A proper response to the difficulties posed by division constitutes a major part of FPgen's capabilities. We present an overview of the relevant verification tasks supplied with FPgen and the underlying algorithms used to target them.

Index Terms—G.1.0.a Computer arithmetic, B.2.2.b Verification, B.2.3.d Test generation.

1 Introduction

Floating-point unit verification presents a unique challenge in the field of processor verification. The particular complexity of this area stems from the vast test space, which includes many corner cases, each of which should be targeted, and from the intricacies of the implementation of floating-point operations.

Both formal methods and simulation methods have been developed to deal with this challenge (e.g., [8], [1], [7], [10], [11]). Formal verification techniques are successfully employed to prove implementation correctness. However, they require a significant investment of machine and manual work time, and are limited to small and neatly defined implementation fragments. On the other hand, simulation based techniques can be applied regardless of state space size. In this paradigm, instead of proving correctness, one acquires confidence in it by running a set of tests that are assumed to be representative samples of the entire space. In practice, a typical verification process uses a combination of both techniques.

Historically, the division operation has been regarded as a significant verification challenge. This operation is usually implemented as a sequence of basic operations (often referred to as "microcode") rather than a dedicated logic circuit; an example can be found in [3]. As a result, it exhibits both an exceptionally wide array of intricate corner cases and an immense state space, challenging both simulation and formal methods.

The formal verification camp has addressed this challenge with an alternative approach based on theorem proving technology [8]. While attractive, this technique requires 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 [9], often deviate from this form (e.g., because of optimizations) and render this technique inapplicable. Simulation based methods must therefore step up to the challenge.

This paper presents our experience addressing the problem posed by division, accumulated through involvement in the verification process of multiple designs. This experience is embodied in the FPgen verification solution. *FPgen* [5] comprises a comprehensive test plan for floating point arithmetic and a model based random test generator, following the methodology outlined in [19]. FPgen's generation capabilities are mainly based on constraint satisfaction technology [18]. As part of a verification process, FPgen produces a battery of test cases, in the form of input to floating point operations, targeting the corner cases outlined by the test plan. In the case of division these can be inherent to the operation as defined in IEEE Standard 754 [2] or can be particular to a specific implementation. In FPgen we limit ourselves to the pure data path—in other words, we ignore the subtle issues of the operation's interaction with the hardware environment and focus on the correctness of the calculation's actual result.

The primary focus of FPgen is to solve data constraints on operands of individual floating point instructions. A data constraint on an operand is defined as the set of values that can be selected for this operand. An individual instruction may have independent data constraints for each one of its operands. Solving all the instruction constraints is equivalent to selecting a value from each given set, such that the instruction semantics are satisfied. FPgen provides engines that solve these constraints within a reasonable time. Also, when multiple solutions exist for a constraint, one should be selected at random, with uniform probability where possible. This randomness is important because the constraints only reflect a suspected area. One instance in this area might reveal a problem, while another might not.

Constructing an appropriate set of such constraints is of utmost importance to ensure successful verification. Exhaustive checking implies testing an enormous, practically unbounded number of different calculation cases; practical computational resources suffice only to simulate a meager fraction of these.

• *All authors are with the IBM Haifa Research Lab, University Campus, Carmel Mountains, Haifa, 31905, Israel*
Email: {elenag,merav,arielb,anatoly}@il.ibm.com

We need to choose these cases very carefully in order to obtain a representative sample of the state space. In particular, a proper focus on the corner cases is a crucial factor in providing a sufficiently comprehensive test battery. Continued analysis of the operations themselves and of the various bugs appearing in their implementations has provided us with valuable knowledge, reflected as an integral part of FPgen's test plan template.

The next section presents FPgen's test plan for division. Then we proceed to describe the solving algorithms implemented to handle this plan. Finally, we lay out directions for future development.

2 FPgen Test Plan for Division

Construction of a sufficiently comprehensive test plan for floating point operations requires a thorough knowledge of the continuous and discrete mathematics involved, as well as specific details of their implementation. This knowledge is crucial to ensure that simulation covers the entire test area. The division operation is widely recognized to be particularly insidious in this regard, mainly because of the intricate methods required to compute it efficiently. These yield additional corner cases, which are hard to recognize and describe, thereby producing a much more complex test area (see [4]). This section presents the test plan we developed to respond to this challenge. This is only one part of a larger, comprehensive test plan for floating point arithmetic in general. For this paper, we extracted the sections concerning division.

To describe our test plan, we introduce the notions and terminology of *functional coverage* [12]. The basic concept is that of an *event*, which is an observable property that can hold (or not) for a given test case. If the property does hold, we say that the test case *covers* the event. A *coverage model* is a set of related events that we expect to witness during simulation. By defining such a model, we can give a formal criterion to measure the thoroughness of our tests. In particular, we seek to capture the conditions leading to the interesting, "corner" cases where bugs often lurk.

Note that a given event might be covered by more than one test case. In this case, it is desirable to select a random sample of these cases, ideally with a uniform distribution within the boundaries set by the event.

Coverage models can be used in two different manners: *a posteriori*, tracking the events that occur during simulation and reporting the ones not being covered, and *a priori*, generating for each given event one or more tests assured to cover it. FPgen employs the latter approach, combining a wide array of coverage models with constraint satisfaction technology to generate the appropriate tests.

We present the coverage models from FPgen's test plan that address the division operation. We distinguish between three families of models. The *IEEE 754 oriented* models are derived from the standard's definition of division and defined in terms of the input and output of the operation. The *intermediate result* models are inspired by properties of internal representations of floating point numbers in existing designs and their interactions with division. The *implementation specific* models explicitly target concrete division algorithms and their implementations.

2.1 Notation

In order to define the models, we introduce some basic concepts.

2.1.1 Basic Floating Point Classes

IEEE 754 defines the following number classes: zeroes, normal numbers, denormal numbers, infinities, and not-a-numbers (NaNs). In addition, we distinguish the minimal and maximal magnitudes of normal and denormal numbers as special cases. Also, we distinguish quiet and signaling NaNs. Decimal floating point introduces the notion of cohorts; in this domain we add as special cases the representations of zero and the minimal normal magnitude with the smallest possible exponent. We call each of the preceding cases a *basic class*.

2.1.2 Unit in the Last Place

An *ulp* is defined as the value of a 1 in the last place of a number representation. For the binary floating point number $b_0.b_1 \ldots b_k \times 2^m$, $1\,\mathrm{ulp} = 2^{m-k}$. For the decimal floating point number $d_0 d_1 \ldots d_k \times 10^m$, $1\,\mathrm{ulp} = 10^m$. An ulp is a relative quantity, defined in terms of a given number. However, this number is often clear from the context, in which case we use ulp as a quantity without further qualification.

2.1.3 Intermediate Results

The IEEE standard requires each operation to be computed as if an infinitely precise result had been rounded in the specified direction to fit into the format of the destination. To comply with this requirement, it is sufficient to produce an intermediate value with the following characteristics:

- An exponent range four times larger (i.e., two bits wider) than that of the result format.
- One additional digit in the fractional part, known as a *guard digit*.
- A single bit representing the truncated part of the infinitely precise result. This *sticky bit* is zero if and only if the intermediate value is exactly equal to the operation's result (i.e., the truncated part equals zero).
- For decimal floating point, an extra bit indicating that the result's exponent differs from the ideal one for the operation, known as the *rounded bit*.

We refer to this value as the operation's *architectural intermediate result*. It allows us to define a variety of interesting models related to different aspects of rounding.

The above argument notwithstanding, it is common for implementations to maintain intermediate results whose precision exceeds the minimum required for strict standard compliance. We refer to these as *microarchitectural intermediate results*, and to the excess precision digits as *extra digits*. In addition, subtraction and multiplication, the microarchitectural intermediate result can be long enough to capture the infinitely precise intermediate result of the operation. In case of division, since the result may truly be infinite, the least significant bit in the microarchitectural intermediate result is the sticky bit.

We henceforth use the term *intermediate result* without qualification where there is no ambiguity in doing so.

We now describe the models themselves. Recall that a model is defined as a set of events, which are properties of a given test case. We consider a test case as a choice of operands (dividend and divisor) and the state of the floating point environment (rounding mode, enabling of exceptions, etc.), along with the intermediate and final results computed from them.

We define the events in our models as combinations of certain properties of the test case, which must all hold true at the same time for the event to be covered. Every such property is given by an attribute of the test case (an operand, rounding mode, etc.) and a value for that attribute. The model is then defined as the Cartesian product of the sets of possible values for each attribute[1].

Note that the properties examined by a given model need not determine a test case uniquely, in which case FPgen selects a random case meeting the requirements.

2.2 IEEE 754 Oriented Models

These generic models are based on analysis of the properties of the division operation as defined by the IEEE floating point standard [2]. They mainly revolve around the various combinations of number classes appearing as operands or results, as well as the different rounding modes.

2.2.1 Basic Input Classes

Attributes:

- Basic class (as defined in 2.1.1) of the divisor; all possible values (as enumerated in the same subsection).
- Basic class of the dividend; all possible values.

This model addresses the interactions between the different classes of numbers in both operand positions. Note that for this basic model we do not constrain any environment parameter; these are dealt with in later models.

2.2.2 Basic Output Classes

Attributes:

- Sign of the quotient; all values (positive and negative).
- Basic class (as defined in 2.1.1) of the quotient; all possible values (as enumerated in the same subsection).
- Rounding mode (as defined in [2]); all possible values.
- Enable bits (as defined in [2]); all possible combinations.

This model addresses the rounding mechanism and some basic calculation mechanisms.

2.2.3 All Basic Classes

The event set of this model is the Cartesian product of "Basic Input Classes" and "Basic Output Classes". This model systematically covers all corner cases derived from the format definitions. For instance, we wish to observe a division between two normal numbers yielding a denormal quotient.

1. Or rather a subset of it, since not all such combinations are possible within the semantics of the operation. For example, a constraint of the form $0/x = \mathrm{MaxNorm}$ is impossible to satisfy.

The reader will rightly observe that this model subsumes the previous two, and thus they seem redundant. However, these models have distinct roles in the verification process. The simpler models are used in earlier verification stages to ensure that all basic classes have been addressed in the implementation of the division operation. This model, in contrast, aims for a more comprehensive testing of the pitfalls and corner cases.

2.2.4 Output Corner Case Neighborhoods

Attributes:

- Quotient; all neighborhoods (see below) of the following special values: zero, one, minimal and maximal denormal, minimal and maximal normal.
- Rounding mode; all possible values.
- Enable bits; all possible combinations.

In the above, we define the *neighborhood* of a number f as the union of two kinds of neighborhoods: the punctured interval $[f-4\,\mathrm{ulp}, f+4\,\mathrm{ulp}]\setminus\{f\}$, representing a neighborhood of f over the number line, and the set of numbers that differ from f in a single digit of the fractional part, representing a neighborhood of f as a coordinate vector.

This model was inspired by a variety of rounding problems observed in several implementations over the years, caused by a number of mistaken assumptions made in the course of optimizing the operation.

2.2.5 Almost Equal Significands

Attributes:

- $\lfloor \log_b |s - s'| \rfloor$, where s and s' are the significands of the dividend and divisor, respectively (represented as natural numbers); all values between 0 and the width in digits of the significand.
- Rounding mode; all possible values.

This model addresses a particular class of rounding problems. Certain implementations tend to drop several least significant digits from the quotient before rounding, causing a loss of precision when the significands are close to each other. In particular, in the case where the significands differ by one ulp they produce a "near exact" rounding case for the result close to one.

2.2.6 Division By Zero

Attributes:

- Dividend class: zero, finite nonzero, infinite, NaN. Note that these are not the "basic classes" defined above but a different partition of the range of floating point numbers.
- Divisor: zero.
- Enable bits: all combinations.

This model tests for correct handling of the division by zero exception.

2.2.7 Special Input Significands

Attributes:

- Significand of divisor; all special patterns (see below).
- Significand of dividend; all special patterns (see below).

We seek to observe a number of special patterns in the significand of each of the input operands. These patterns are:

- Sequence of leading zeroes.
- Sequence of leading ones (nines for decimal).
- Sequence of trailing zeroes.
- Sequence of trailing ones (nines for decimal).

where the length of each such sequence runs from 1 to the precision in digits of the significand. Additionally, for binary floating point we consider these additional patterns:

- The number of ones is significantly less than the number of zeroes.
- The number of zeroes is significantly less than the number of ones.
- Checkerboard pattern (for example: $00110011\ldots$ or $011011011\ldots$).

The precise definition of these patterns is rather cumbersome; we elide it for the sake of clarity. These models reflect recurring patterns observed in several bug situations.

2.3 Intermediate Result Oriented Models

These models observe the intermediate results of the division computation, both general and microarchitectural (see 2.1). The first four models refer to different aspects and cases of the result's rounding after the computation. The next two refer to overflow and underflow detection. The last one refers to correct calculation of the sticky bit.

2.3.1 Rounding Direction

Attributes:

- Sign; all values.
- Least significant digit of the significand; all values.
- Guard digit; all values.
- Sticky bit; all values.
- Rounding mode; all values.
- Enable Inexact bit; all values.

2.3.2 Rounding Corner Cases - General Intermediate

Attributes:

- Sign; all values.
- Intermediate result class (see below).
- Guard digit; all values.
- Sticky bit; all values.
- Rounding mode; all values.
- Overflow, Underflow, and Inexact bits; all combinations.

The intermediate result range is partitioned for this model as follows:

- Zero.
- $(0, \mathrm{MinDenorm}/2]$ (where $\mathrm{MinDenorm}$ is the minimal denormal number).
- $(\mathrm{MinDenorm}/2, \mathrm{MinDenorm}]$.
- $(\mathrm{MinDenorm}, \mathrm{MinNorm}]$ (where $\mathrm{MinNorm}$ is the minimal normal number).
- $(\mathrm{MinNorm}, \mathrm{MaxNorm}]$ (where $\mathrm{MaxNorm}$ is the maximal normal number).
- $(\mathrm{MaxNorm}, \infty]$.

2.3.3 Rounding Corner Cases - Microarchitectural Intermediate

Attributes:

- Sign; all values.
- Rounding mode; all values.
- Overflow, Underflow, and Inexact bits; all combinations.

The microarchitectural intermediate result, Sticky and Guard bits enumerates on the cross-product of the values in the following table:

Fraction	Guard-Bit	Extra-Bits	Sticky
XXX...XX0	0	[0000...0000,0000...0111]	0
XXX...XX1	1	[11...1000,1111...1111]	1

2.3.4 Subnormal Rounding Cases

Attributes:

- Sign; all values.
- Intermediate result exponent; all values corresponding to denormal numbers.
- Intermediate result significand; all sequences of trailing zeroes, all sequences of trailing ones (nines for decimal).
- Guard digit; all values.
- Sticky bit; all values.
- Rounding mode; all values.
- Enable Inexact bit; all values.

2.3.5 Near Overflow

Attributes:

- Sign; all values.
- Intermediate result class (see below).
- Rounding mode; all values.
- Overflow and Inexact bits; all combinations.

Intermediate result classes:

- All values in $[\mathrm{MaxNorm} - k\,\mathrm{ulp}, \mathrm{MaxNorm} + k\,\mathrm{ulp}]$.
- Any value from $(\mathrm{MaxNorm} + k\,\mathrm{ulp}, \infty)$.
- For every $e \in [e_{\mathrm{MaxNorm}} - k, e_{\mathrm{MaxNorm}} + k]$, any value with exponent e.

2.3.6 Near Underflow

Attributes:

- Sign; all values.
- Intermediate result class (see below).
- Rounding mode; all values.
- Underflow and Inexact bits; all combinations.

Intermediate result classes:

- Any value from $[0, \mathrm{MinDenorm} + k\,\mathrm{ulp})$.
- All values in $[\mathrm{MinDenorm} - k\,\mathrm{ulp}, \mathrm{MinDenorm} + k\,\mathrm{ulp}]$.
- All values in $[\mathrm{MinNorm} - k\,\mathrm{ulp}, \mathrm{MinNorm} + k\,\mathrm{ulp}]$.
- For every $e \in [e_{\mathrm{MinNorm}} - k, e_{\mathrm{MinNorm}} + k]$, any value with exponent e.

2.3.7 Sticky Bit Calculation

Attributes:

- Sign; all values.
- Extra digits; all values with a single nonzero digit.
- Rounding mode; all values.

2.4 Implementation Specific Models

Observation over the years of several hard-to-find bugs led us to conclude that the generic, standards based models do not suffice to address the subtle intricacies introduced by the different implementations of division [17]. It thus becomes necessary to define specific coverage models for this purpose.

A major distinction appears when considering binary and decimal floating point. The former has been thoroughly studied in the field of hardware design over the last decades, producing increasingly sophisticated implementations (and accordingly sophisticated bugs) as technology and demands evolve. The latter, on the contrary, has only recently been gaining attention in this field, being until now relegated to the realm of software. Therefore, our efforts so far have focused on the more mature binary division.

There exist several different algorithms to implement the divide instruction. In this paper we address only the iterative numerical approximation methods [3]. In these methods, an initial guess is determined with the aid of a precalculated lookup table, and then subsequently improved by successively adding a correction term, depending on the relative error of the current calculation. Through static analysis, the number of such steps required to reach the sufficient accuracy for the desired floating point format is known in advance. At this point, the result after rounding is provably correct.

The intricacies of implementing such an algorithm accommodate new families of corner cases, which differ significantly from those observed in the previous models. Failure to recognize these cases caused a number of bugs to pass undetected until late stages of verification.

For example, in a certain implementation, the internal ternary multiply-add operation used to compute the correction term gives special meaning to one of its excess precision bits when its addend is 0; this condition can arise when the relative error itself is close enough to zero. This extra bit is dropped in regular calculations. However, the division implementation saved it to improve accuracy and computation time. Consequently, the correction term was miscalculated and so was the end result.

In the example above, the corner case leading to the bug could be characterized as the case where the relative error after a given iteration lies within a given interval (in this case a neighborhood of zero). Other cases we encountered were amenable to a similar description. This led us to formulate the following model.

2.4.1 Relative Error

Attributes:

- Iteration; all iterations, from the initial approximation to the next-to-last refinement step. The following attributes refer to the signed relative error *after* this iteration.
- Sign of relative error; all values.
- Magnitude of relative error; Some value from an appropriate neighborhood of each of 0 and all local and global extrema.
- Rounding mode; all values.

Solver	Models Targeted
Input	Almost Equal Significands
	Basic Input Classes
	Division by Zero
	Special Input Significands
Range	All Basic Classes
Intermediate	Basic Output Classes
	Output Corner Case Neighborhoods
	Near Overflow
	Near Underflow
	Rounding Corner Cases
	Rounding Direction
	Sticky Bit Calculation
	Subnormal Rounding Cases
Relative Error	Relative Error

Fig. 1. Constraint solvers included in the FPgen test generator. Next to each solver are the models it targets.

In the above, the extrema and their neighborhoods are determined by manual analysis of the specific algorithm used and its actual implementation.

This model targets the corner cases described above, yielding test cases covering interesting absolute values of relative error after all iterations.

3 FPGEN TEST GENERATOR

To make use of the test plan outlined above, we need a tool capable of generating sets of test cases that cover each of the models, thus guiding the simulation effort into all interesting, bug-prone areas.

To yield the required coverage for each model, FPgen provides mechanisms to define and solve an array of constraints for a variety of floating point operations. We describe four of these solvers, namely, the ones needed to implement the test plan for division. Table 1 lists each solver next to the models from our test plan that it targets.

3.1 Input Solver

To describe the different models referring to the operation's input operands (dividend and divisor), we need the following basic building blocks:

Interval

For example: $[\mathrm{MinDenorm}, \mathrm{MaxDenorm}]$. This is a convenient way to define the basic floating point classes (e.g., denormal numbers), and to target the vicinity of critical values.

Mask

Each bit can be specified as 0, 1, or X, where X signifies "don't care". For example, the set ± 0 is defined by $X00\ldots00$. This provides a means of targeting the neighborhood (in terms of Hamming distance) of critical values. An important use of masks is constraining the lower bits of the intermediate result to test for correct rounding.

Number of ones/zeros

Constrains the number of digits equal to zero or one to be within a given interval. These values often trigger special behavior in implementations.

Sequence of ones/zeros

Similar to the previous constraint, but defined in terms of sequences of contiguous, identical digits.

These can be applied to each part of a floating point number (sign, exponent, and significand) independently, or to several parts jointly. Also, the definitions above are appropriate for binary floating point; their extension to decimal is straightforward.

To cover the relevant models, it suffices to choose a random element from each of the four kinds of sets listed above. Of these, masks are fairly simple to handle since they only refer to the number as a vector of digits, each of which is selected independently. All the others are straightforward to describe in terms of masks themselves.

3.2 Range Solver

To cover the "All Basic Classes" model described in 2.2.3, we need to solve the following problem:

> Given three intervals $I_x = [L_x, H_x]$, $I_y = [L_y, H_y]$, and $I_z = [L_z, H_z]$, find two floating point numbers $x \in I_x$ and $y \in I_y$ such that $\mathrm{round}(x/y) \in I_z$.

Without loss of generality, we may assume that all three intervals lie on the positive side of the real line. All other cases can be easily reduced to this. For the sake of simplicity, we first solve the problem in a continuous domain (i.e., we work with real instead of floating point numbers). Then we show how the solution applies in our actual, discrete setting.

We begin by eliminating the rounding operation. Let

$$I'_z \triangleq \{z|\ \mathrm{round}(z) \in I_z\}$$

Clearly I'_z is an interval, and $I_z \subseteq I'_z$. We denote the endpoints of this interval by L'_z and H'_z; their open or closed nature depends on the rounding mode. We may now formulate our problem as finding two numbers $x \in I_x$ and $y \in I_y$ such that $x/y \in I'_z$.

Our next step is to minimize the problem intervals. By this we mean to find the smallest subintervals of I_x, I_y and I'_z that yield the same solution set for the problem.

For every $L_x \le x \le H_x$ and $L_y \le y \le H_y$, we have

$$L_x/H_y \le x/y \le H_x/L_y$$

Furthermore, if $L'_z \le x/y \le H'_z$, we have

$$L'_z L_y \le L'_z y \le x \le H'_z y \le H'_z H_y$$

as well as

$$1/H'_z \le y/x \le 1/L'z$$
$$L_x/H'_z \le x/H'_z \le y \le x/L'z \le H_x/L'_z$$

By intersecting these with the original intervals we obtain:

$$A_x = \max\{L_x, L'_z L_y\} \le x \le \min\{H_x, H'_z H_y\} = B_x$$
$$A_y = \max\{L_y, L_x/H'_z\} \le y \le \min\{H_y, H_x/L'_z\} = B_y$$
$$A_z = \max\{L'_z, L_x/H_y\} \le x/y \le \min\{H'_z, H_x/L_y\} = B_z$$

Some of the inequations are strict, depending on the open or closed nature of the endpoints of I'_z.

The inequations yield the minimal subintervals we desired. We show minimality for $[A_z, B_z]$; proofs for other cases are similar.

It suffices to show that for every z in this interval there are $x \in I_x$ and $y \in I_y$ such that $x/y = z$. Hence, every strict subinterval of $[A_z, B_z]$ yields a strictly smaller solution set. Let $A_z \le z \le B_z$. In particular, $L_x/H_y \le z \le H_x/L_y$. Division is continuous over the positive real numbers ($\mathbb{R}^2_+$, to be precise). Hence, by the Mean Value Theorem $(x, y) \in I_x \times I_y$ must exist such that $x/y = z$, which proves the claim.

Now, to generate a random solution to the original problem, choose x at random from the reduced interval $[A_x, B_x]$, then choose y at random from the interval $[\max\{A_y, x/B_z\}, \min\{B_y, x/A_z\}]$. Based on the argument above, $A_z \le x/y \le B_z$, thereby yielding the requested solution.

To apply this method to the discrete floating point domain, we need to account for the possibility that either of the (real) intervals $[A_x, B_x]$ and $[\max\{A_y, x/B_z\}, \min\{B_y, x/A_z\}]$ above fails to contain any numbers representable in the desired format. In the first case, we conclude there is no solution. In the second case, we simply discard our random choice of x and start over, giving up after a preset number of attempts. In addition to this, after choosing x and y, we need to check that the result of x/y still lies in the desired interval after rounding. While this algorithm might fail to find a solution when one exists when the domains involved are sparse, in practice it works reasonably well. In particular, it yields effective coverage of the "All Basic Classes" model.

3.3 Intermediate Solvers

Correct computation of the result of any floating point operation involves first computing an infinitely precise intermediate result, and then rounding according to the required rounding mode, to produce the final result. In practice, the finite intermediate result described in 2.1.3 suffices. Correct computation of the intermediate result is crucial for correct behavior of the operation.

Several models in the test plan (see Fig. 1) target corner cases of the intermediate result of the divide operation. Additional models refer to cases of the final result, which can in turn be easily recast into appropriate choices of intermediate result and rounding mode.

The solver for the intermediate result of binary floating point is straight-forward and we do not describe it here.

We present here the solver algorithms for the binary microarchitectural intermediate result and for the decimal intermediate result.

3.3.1 Binary Microarchitectural Intermediate Solver

Denote the final result's precision in bits by p. Denote the precision in bits of the microarchitectural intermediate result by q, where $q > p$. Different models require different values of q. However, an accuracy of $q = 2p$ bits is always sufficient for correct rounding of the infinitely precise quotient. This is true, because the first $2p$ bits of the quotient determine whether the remaining bits in the (possibly infinite) tail of the quotient must all be zero, or can contain bits that are 1 (see [15]).

Examples of interesting microaarchitecural intermediate results are hard-to-round cases and extreme cases of sticky-bit calculation. To target such cases, we must generate random inputs whose division yields this pattern of result. Generating the desired exponent for the result is straightforward. Selecting appropriate significands for the inputs is not so. Thus we define the following problem:

> Given a specific value S_c, for the significand of the microarchitectural intermediate result, and a specific value for the sticky bit σ_c, find two significands, S_a and S_b, for the input operands a and b, so that the microarchitectural intermediate result of their division, c, has significand S_c and sticky bit σ_c. If this is not possible, state that no solution exists.

We divide the discussion into two cases: when the sticky bit of the microarchitectural intermediate is zero and when the sticky bit is one.

Case I: Sticky Bit Equals Zero

In this case, S_c must equal the significand of a/b exactly, where $1 \leq S_a < 2$, $1 \leq S_b < 2$, and $1 \leq S_c < 2$. We recast this into a problem over the integers. First, we note that we must have either (i)$S_c = S_a/S_b$ or (ii) $S_c/2 = S_a/S_b$. Since S_a, S_b have p bits, and S_c has q bits, we get the following two cases, which involve binary integer equations:

$$\text{(i)} \quad (2^{p-1}S_b) \times (2^{q-1}S_c) - (2^{p-1}S_a) \times (2^{q-1}) = 0$$
$$\text{(ii)} \quad (2^{p-1}S_b) \times (2^{q-1}S_c) - (2^{p-1}S_a) \times (2^{q}) = 0$$

In these equations, $(2^{p-1}S_a)$, $(2^{p-1}S_b)$ are unknown integers and $(2^{q-1}S_c)$, (2^{q-1}), (2^q) are known integers. Equations of this type are known as Diophantine equations, and their solution is well known. We fully describe the solution for case (i) only; the solution for case (ii) is similar.

Case (i): We divide the coefficients of the equation, $(2^{q-1}S_c)$ and (2^{q-1}) by their largest common divisor and get two integers, a', b', which are relatively prime. We have, then, the equivalent equation:

$$(2^{p-1}S_b) \times a' - (2^{p-1}S_a) \times b' = 0$$

The set of all positive, integral solutions is given by $(2^{p-1}S_a) = \alpha a'$, $(2^{p-1}S_b) = \alpha b'$, where $\alpha \in \{1, 2, 3, \ldots\}$. Since S_a, S_b are limited to the ranges $1 \leq S_a < 2$, $1 \leq S_b < 2$, α must satisfy $2^{p-1} \leq \alpha a' < 2^p$, $2^{p-1} \leq \alpha b' < 2^p$, which is equivalent to:

$$\max\{2^{p-1}/a', 2^{p-1}/b'\} \leq \alpha < \min\{2^p/a', 2^p/b'\}$$

The existence of solutions depends on the existence of integers α in this interval. Each such integer produces exactly one solution, $S_a = \alpha a' 2^{1-p}$, $S_b = \alpha b' 2^{1-p}$.

Case II: Sticky Bit Equals One

In this case, the significand of a/b must be of the form $S_c + \epsilon$ where $0 < \epsilon < 2^{1-q}$. This means we again have two possibilities: (i) $S_c + \epsilon = S_a/S_b$, and (ii) $(S_c + \epsilon)/2 = S_a/S_b$, which can be converted into the following equations with integer coefficients and variables:

$$\text{(i)} \quad (2^{p-1}S_a)/(2^{p-1}S_b) - (2^{q-1}S_c)/(2^{q-1}) = \epsilon$$
$$\text{(ii)} \quad (2^{p-1}S_a)/(2^{p-1}S_b) - (2^{q-1}S_c)/(2^{q}) = \epsilon/2$$

where in both cases $0 < \epsilon < 2^{1-q}$. Again, $(2^{q-1}S_c)$, (2^{q-1}), (2^q) are known integers and $(2^{p-1}S_a)$, $(2^{p-1}S_b)$ are unknown integers, which are to be generated.

Note that, in both cases, the denominator $2^{p-1}S_b$ of the unknown fraction (i.e., the first fraction) is significantly smaller than the denominator 2^{q-1} or 2^q of the known fraction. Therefore, we are faced with the problem of finding a close approximation to a known fraction with a large denominator, by a fraction with a smaller denominator. We base our solution on the use of continued fractions, which provide this type of approximation in a most natural way.

We avoid a full discussion on the properties of continued fractions; instead we give only a brief explanation and state their main properties providing the underlying intuition. A more detailed exposition can be found in textbooks on the subject, for example [13].

Let $\bar{p}/\bar{q}$ be a fraction where $\bar{p}$, $\bar{q}$ are positive integers.

1) The continued fraction representation of $\bar{p}/\bar{q}$ is
$$\bar{p}/\bar{q} = p_0 + \frac{1}{p_1 + \frac{1}{p_2 + \ldots + \frac{1}{p_n}}} = [p_0; p_1, p_2, \ldots, p_n]$$
where $p_0 \in \{0, 1, 2, \ldots\}$, $p_m \in \{1, 2, 3, \ldots\}$ $(m = 1, 2, \ldots, n-1)$ and $p_n \in \{2, 3, \ldots\}$
2) We denote $\alpha_m = [p_0; p_1, p_2, \ldots, p_m]$. Therefore, $\alpha_0 = p_0$, $\alpha_1 = p_0 + 1/p_1$,..., $\alpha_n = \bar{p}/\bar{q}$. α_m is the m'th order *convergent* of $\bar{p}/\bar{q}$. If m is even, then $[p_0; p_1, p_2, \ldots, p_{m-1}, t]$ is a strictly increasing function of the positive real variable t. If m is odd, then it is strictly decreasing.

Denoting $\bar{p} = (2^{p-1}S_a)$, $\bar{q} = (2^{p-1}S_b)$ the unknown integers to be generated, we write (i), (ii) in the form:

$$\text{(i)} (2^{q-1}S_c)/(2^{q-1}) < \bar{p}/\bar{q} < [(2^{q-1}S_c) + 1]/(2^{q-1})$$
$$\text{(ii)} (2^{q-1}S_c)/(2^{q}) < \bar{p}/\bar{q} < [(2^{q-1}S_c) + 1]/(2^{q})$$

The unknowns must both be p-bit binary integers. Namely, they must satisfy the relations:

$$2^{p-1} \leq \bar{p} < 2^p, \quad 2^{p-1} \leq \bar{q} < 2^p$$

We propose the following algorithm based on continued fractions to generate such random $\bar{p}$ and $\bar{q}$. We only present a short description of it here. A full discussion appears in our previous work [16]. We solve only case (i) here, without loss of generality; the algorithm for case (ii) is similar.

We follow along the formal description with an example. Suppose the given fraction is $C = 1.01101100110_2$.

We start by writing the continued fraction expansion of the given fractions, those that constitute the leftmost and the rightmost sides of inequality (i).

We denote $\bar{p}'/\bar{q}' = (2^{q-1}S_c)/(2^{q-1})$ and $\bar{p}''/\bar{q}'' = [(2^{q-1}S_c) + 1]/(2^{q-1})$. Since the two fractions are close to each other, it is likely that their expansions coincide up to a certain point. Thus we have $\bar{p}'/\bar{q}' = [p'_0; p'_1, \ldots]$, $\bar{p}''/\bar{q}'' = [p''_0; p''_1, \ldots]$ where $p'_0 = p''_0, \ldots, p'_{i-1} = p''_{i-1}$. If we denote the expansion of $\bar{p}/\bar{q}$ by $[p_0; p_1, \ldots]$, its first i terms must coincide with those of $\bar{p}'/\bar{q}'$ and of $\bar{p}''/\bar{q}''$. As a result, the first i convergents of the three fractions are known (they are identical) and we can denote them by P_m/Q_m, $m = 0, 1, \ldots, i-1$.

We denote the tails of the three fractions by $\hat{p}'/\hat{q}' = [p'_i; p'_{i+1}, \ldots]$, $\hat{p}''/\hat{q}'' = [p''_i; p''_{i+1}, \ldots]$, $\hat{p}/\hat{q} = [p_i; p_{i+1}, \ldots]$. Clearly the first two tails are known numbers and the third one is an unknown, which is to be generated. What we know is that $\hat{p}/\hat{q}$ lies between $\hat{p}'/\hat{q}'$ and $\hat{p}''/\hat{q}''$.

Using known properties of continued fractions, we can translate this requirement into a set of inequalities. Solving these inequalities gives lower and upper bounds on $\bar{q}$. For a full derivation of the inequalities the reader is referred to [16].

In the example above, the continued fraction of C is $[1; 2, 2, 1, 4, 1, 2, 2, 1, 2]$ and of $C + 1ulp$ is $[1; 2, 2, 1, 5, 1, 1, 2, 3]$. We need to select a fraction from the range between these two fractions. In the realm of continuous numbers (rational numbers) this is always possible. But in the world of floating-point numbers, since the number of bits of the fraction is bounded, this range may be empty.

If the intersection of the bounds is empty, there is no solution that satisfies the requirements. If it is not empty, we select a random positive integer $\hat{q}$ within these bounds.

The choice of $\hat{q}$ along with the inequalities now pose constraints on $\hat{p}$. If such an integer exists within the new bound on $\hat{p}$, then we have a solution. Otherwise we will have to select a new $\hat{q}$. This is an iterative process that might continue indefinitely. However, empirical results show that the probability of not finding a solution after fifteen attempts is below 0.1%.

We can further improve the efficiency of the algorithm by refining the original interval in which $\bar{p}/\bar{q}$ resides, $(S_c, S_c + 2^{1-q})$ in case (i) and $(S_c/2, S_c/2 + 2^{-q})$ in case (ii).

Since it is known that the binary representation of the ratio of two p-bit integers cannot include more than p consecutive 0s nor p consecutive 1s, we will not lose any solutions by replacing the ends of the interval in case (i), by $(S_c + 2^{-q-p}, S_c + 2^{1-q} - 2^{1-q-p})$ and in case (ii) by $(S_c/2 + 2^{-q-p-1}, S_c/2 + 2^{-q} - 2^{-q-p})$. In some cases, this refinement improves the efficiency of the algorithm.

3.3.2 Decimal Intermediate Solver

We begin with the following definitions.

Decimal Floating Point Number

Using the notation of IEEE Standard 754, a decimal floating-point number is defined by $(-1)^s 10^e (d_0 d_1 d_2 \ldots d_{p-1})$ where s is the sign, e is the (unbiased) exponent, $E_{min} \leq e \leq E_{max}$, and $d_0 d_1 d_2 \ldots d_{p-1}$ is the significand with d_i each representing a decimal digit, $d_i = \{0, 1, 2, ..., 9\}$. Unlike binary floating point, decimal floating point numbers are not normalized and as a result a single value may have multiple representations. For example $5 \cdot 10^0$ and $500 \cdot 10^{-2}$ are two different representations of the same value.

Precision

Maximal number of digits in the significand. Denoted as p.

Unbounded Intermediate Result

The result of a floating point operation, assuming unbounded precision and unbounded exponent range.

Intermediate Result

This definition is a special case of that given in 2.1.3. Assuming the final result has a precision p, the intermediate result is defined by a sign, ($+$ or $-$), an exponent, e, a significand, S, and a sticky bit σ and has the form $(-1)^s 10^{e-1} d_0 d_1 \ldots d_{p-1} d_p, \sigma$ where $\sigma \in \{0, 1\}$. We will refer to d_p as the guard digit. Each digit d_i of the intermediate result is identical to the corresponding digit of the exact result of the operation. If all the remaining digits of the exact result (beyond d_p) are zero, then $\sigma = 0$; otherwise $\sigma = 1$.

Exact Result

The intermediate result is said to be exact when the intermediate result's guard digit and sticky bit are both 0. Note, that an exact intermediate result may produce and inexact final result. For example, it happens when an intermediate exponent is out of the legal exponent range for a given precision.

Preferred Exponent

Because decimal floating point numbers have multiple representations, the IEEE standard defines a *preferred exponent* for every operation. Denoting by e_x and e_y the exponents of the dividend x and divisor y respectively, the preferred exponent for the division x/y is $e_x - e_y$. If the result is exact, the exponent of the intermediate and final results will be the closest one to the preferred exponent within the result's cohort. If the result is inexact, the smallest possible exponent is selected in order to minimize loss of accuracy, thus retaining p significant digits for the final result and $p + 1$ significant digits for the intermediate result[2].

We may now cast our problem as follows[3]:

> Given a constraint on the intermediate result and a constraint on the difference between the actual exponent and the preferred exponent, find two operands x and y such that x/y yields an intermediate result z, compatible with the constraints.

2. These principles are compromised near the boundaries of the legal range for decimal numbers, but such cases are beyond the scope of this paper.

3. This problem and its solution are discussed in [14]. Here we present a simplified view, specific to division.

We denote the dividend, divisor, and quotient by x, y, and z respectively. Their significands are denoted S_x, S_y, and S_z, and the exponents are e_x, e_y, and e_z.

Given the definition of the preferred exponent for division, which is $e_x - e_y$, and the nature of the operation, we conclude that the actual exponent of the result is always less than or equal to the preferred exponent. We denote the exponent difference

$$\begin{aligned} d &= \text{preferred exponent} - \text{actual exponent} \\ &= e_x - e_y - e_z \end{aligned}$$

We partition the problem into three subcases:

Case	d	Guard Digit	σ
I	Zero	Zero	0
II	Nonzero	Any	0
	Any	Nonzero	0
III	Any	Any	1

Case I

This case indicates an exact result, and can be viewed as a division between two integers with no remainder. Let S'_z denote the integer value of S_z with the guard digit removed. Note that S'_z has at most p digits. We proceed as follows:

1) Choose a random value for S_y. Since $S_x = S_y \cdot S'_z$ and $S_x < 10^p$, the value chosen for S_y must be in the range $1 \le S_y < \frac{10^p}{S'_z}$. Note that the trivial solution $S_y = 1$ is always applicable.
2) Calculate $S_x = S_y \cdot S'_z$
3) Choose e_x and e_y such that $e_x - e_y = e_z$.

Case II

This case indicates an inexact final result with an exact intermediate result; in other words, an inexact result with $\sigma = 0$. The following equation, based on the definition of decimal division and preferred exponent, applies to any case where the sticky bit is 0:

$$S_x / S_y = S_z / 10^{d+1}$$

therefore

$$S_x \cdot 10^{d+1} = S_y \cdot S_z \tag{1}$$

Our case imposes certain constraints on S_z:

- S_z can have at most one trailing zero (in the guard digit). Any further trailing zeros would be shifted out, increasing e_z and thereby approaching the preferred exponent. If $d = 0$ then there are no trailing zeros, by the definition of the case (otherwise we are back in Case I).
- Let $S_z = S'_z \cdot 2^j \cdot 5^k$, where S'_z is prime to 10. Then S'_z must have p digits or less. Specifically, if S_z has $p+1$ digits, then it cannot be prime to 10. This is a result of Equation 1—note that according to the equation, S_x must be a multiple of S'_z, and remember that S_x has p digits or less.

The solution is based on Equation 1 and on the factorization given above for S_z. As we saw, S_x must be a multiple of S'_z. We note also that any instances of 2 or 5 in the factorization beyond those available in the term 10^{d+1} must originate from S_x as well. We proceed as follows:

1) Initialize
$$S_x \leftarrow S'_z \cdot 2^{max(0,j-d-1)} \cdot 5^{max(0,k-d-1)}$$
If S_x has more than p digits, there is no solution.
2) Set
$$S_y \leftarrow 2^{max(0,d+1-j)} \cdot 5^{max(0,d+1-k)}$$
At this point Equation 1 holds.
3) Steps 1 and 2 are deterministic. We now randomize the solution by multiplying S_x and S_y by some identical random factor, keeping their size to less than 10^p.
4) Choose exponents e_x and e_y so that $e_x - e_y = e_z + d$.
5) Return the constructed solution.

Case III

This case occurs when the exact quotient x/y has more than $p+1$ significant digits. Conceptually, the result z is formed by dividing the coefficients and then normalizing the result so that the most significant nonzero digit is in the leftmost position of S_z.

The following relationship holds in this case:

$$S_z / 10^{d+1} < S_x / S_y < (S_z + 1) / 10^{d+1}$$

therefore

$$S_z \cdot S_y < S_x \cdot 10^{d+1} < (S_z + 1) \cdot S_y \tag{2}$$

We denote by $|S_x|$ and $|S_y|$ the number of significant digits in S_x and S_y respectively. Let S'_x and S'_y be the values of S_x and S_y respectively, each shifted left so that it has p significant digits, with zeros shifted in on the right. Then the exponent difference depends only on $|S_x|$, $|S_y|$, and the relative magnitude of S'_x and S'_y. Specifically[4]:

- if $S'_x > S'_y$ (*no-borrow* case), then
$$d = p - |S_x| + |S_y| - 1$$
- if $S'_x < S'_y$ (*borrow* case), then
$$d = p - |S_x| + |S_y|$$

From these relations we learn that $0 < d \le 2p - 1$ in the borrow case and $0 \le d < 2p - 1$ in the no-borrow case.

Given a constraint, we may generate a solution for either the borrow case or the no-borrow case at will. For the sake of simplicity, we only show how to construct one for the former. A solution for the no-borrow case is constructed similarly. The explicit analysis is shown in [14]. The algorithm proceeds as follows:

1) Let $t \triangleq |S_y| - |S_x|$ (for the borrow case, $t = d - p$). By definition:
$$|S_y| = t + |S_x|$$
Since $1 \le |S_x| \le p$:
$$t + 1 \le |S_y| \le t + p$$

4. Note that the case where $S'_x = S'_y$ is irrelevant, since this would imply $\sigma = 0$ which is covered by the previous cases.

Furthermore, since $1 \leq |S_y| \leq p$:

$$\max(1, t+1) \leq |S_y| \leq \min(p, t+p)$$

Choose a random value for $|S_y|$ from this range.

2) Choose a random significand S_y with $|S_y|$ significant digits.
3) Calculate

$$\alpha \triangleq S_z/10^{d+1}; \quad \beta \triangleq (S_z+1)/10^{d+1}$$

Now by Equation 2,

$$\alpha < S_x \cdot 10^{d+1} < \beta$$

If a number with at least $d+1$ trailing zeros exists in the open interval (α, β), set S_x to this number, shifting out $d+1$ zeros.

4) Choose exponents e_x and e_y so that $e_x - e_y = e_z + d$.

This algorithm usually requires several iterations, but in practical terms it produces a solution very quickly for most values of d. An iteration fails if the interval between $S_z \cdot S_y$ and $(S_z+1) \cdot S_y$ has no number with $d+1$ trailing zeros. The size of the interval is $S_y - 1$, while the size of the interval between two successive numbers with $d+1$ trailing zeros is 10^{d+1}. The probability of success may therefore be approximated as

$$\frac{S_y}{10^{d+1}} \geq \frac{10^{|S_y|-1}}{10^{d+1}} = 10^{|S_y|-d-2}.$$

When d approaches its maximal value of $2p-1$, this approach fails. Test cases for large d values are often generated by relaxing the constraint on S_z when possible.

3.4 Relative Error Solver

For the implementation dependent "Relative Error" model described in 2.4.1, we need to delve into some of the details of the iterative approximation method mentioned in the model's description. Recall that in contrast to other floating point operations, binary division is tipically not implemented directly as a logic circuit. Rather, it is translated by the processor into a sequence of operations (sometimes referred as "microcode"), yielding the desired result.

We now present a simplified view of the algorithm 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.

For ease of analysis, we examine the reciprocal operation first; we then show how division follows as a simple extension.

If b is a non-singilar (i.e. isn't zero, infinity or NaN), it has a form

$$b = (-1)^s \cdot 2^e \cdot (1+f) \qquad (0 \leq f < 1)$$

Notice, that if b is denormal, then $e = e_{\min} - 1 - z$, where $e_{\min}$ is the minimal exponent allowed for a given precision and z is the number of leading zeros in the fraction of b.

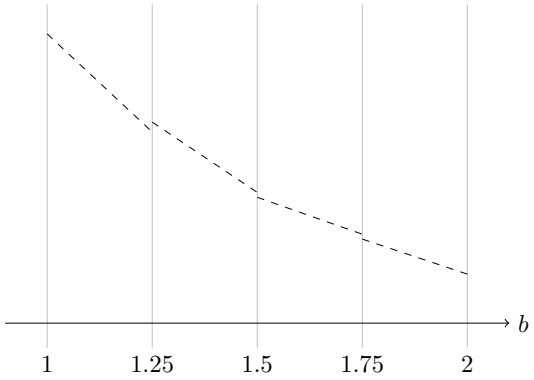


Fig. 2. Piecewise Linear Approximation of $\frac{1}{b}$, with the input domain $[1, 2)$ divided into four sectors. The grey curve shows the actual function and the dashed black lines show the linear approximation for each sector. (Plot is schematic only.)

Now we derive $\frac{1}{b}$ as follows:

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

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[5]. At each refinement step, a correction term is added to the approximation, based on an estimate of the relative error.

As for division, we can assume—as we did for the reciprocal operation—that the numerator a and denominator b are finite and nonzero; other cases are trivial. As we show 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 operation.

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 2 for an illustration).

The approximation and correction terms are derived from a power series expansion of the function being computed.

5. There is an inaccuracy here that we will address later.

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

Before we proceed, we introduce some additional notation to be used in the description of the problem we need to solve and our suggested solution.

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:

$$\begin{aligned} e_q &= \tfrac{q-\widetilde{q}}{q} \\ &= \tfrac{q}{q} - \tfrac{\widetilde{q}}{q} \\ &= 1 - \tfrac{1}{q} \cdot \widetilde{q} \end{aligned}$$

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:

$$\begin{aligned} e_q &= \tfrac{q-\widetilde{q}}{q} \\ &= \tfrac{rs-r\widetilde{s}}{rs} \\ &= \tfrac{s-\widetilde{s}}{s} \\ &= e_s \end{aligned}$$

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*. With these definitions in hand, we can now cast the "Relative Error" model as the following problem:

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

To see how this definition matches the earlier description of the model, first we note that the neighborhoods mentioned in that section are, in fact, intervals. Then we note that, by the argument presented when defining e_q, $e_{\frac{a}{b}} = e_{\frac{1}{b}}$ for relevant (nontrivial) inputs. We can therefore limit our discussion to the reciprocal operation. Moreover, by the same reasoning, the sign and exponent of b do not affect the relative error. If b' has the same fractional part as b, then $e_{q(b')} = e_{q(b)}$. We can thus focus on the fractional part alone. 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.

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.

We now give a solution to this problem. The core of our method is to search iteratively for a solution b to the constraint $e^{(i)}_{q(b)} \in [L, H]$. 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.

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 ← (x+y)/2
loop
    if φ(r) < L then
        x ← r
    else if φ(r) > L then
        y ← r
    else[Solution Found]
        return r
    r ← (x+y)/2
```

Fig. 3. Interval Halving Method

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.

3.4.1 Initial Approximation

Let $\widetilde{q(b)} = \beta(Ab + C)$. If we define $\phi(b) \triangleq e_{q(b)}$:

$$\begin{aligned} \phi(b) &= \tfrac{q(b)-\widetilde{q(b)}}{q(b)} \\ &= \tfrac{\beta b^{-1} - \beta(Ab+C)}{\beta b^{-1}} \\ &= 1 - b \cdot (Ab + C) \\ &= 1 - Ab^2 - Cb \end{aligned}$$

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

$$\phi'(b) = -2Ab - C$$

This function has at most one root over the positive real numbers; 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 3, 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, 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 above. Since q and $\widetilde{q}$ (and therefore ϕ) are known beforehand, we can precalculate the bounds and extrema of ϕ for each sector.

3.4.2 Randomization

Although the algorithm above solves the problem, it does so deterministically, i.e., given the same constraint, it always produces 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 3) 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.

3.4.3 Iterative Refinement

Most iterative methods used in practice are based on 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:

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

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 3.4.1, allows us to solve a range constraint on $e_q^{(i)}$ using the same Interval Halving Method shown above.

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

Example

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.

3.4.4 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 in advance that a solution exists in a given interval.

On the other hand, the same finiteness allows us to adapt the Interval 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.

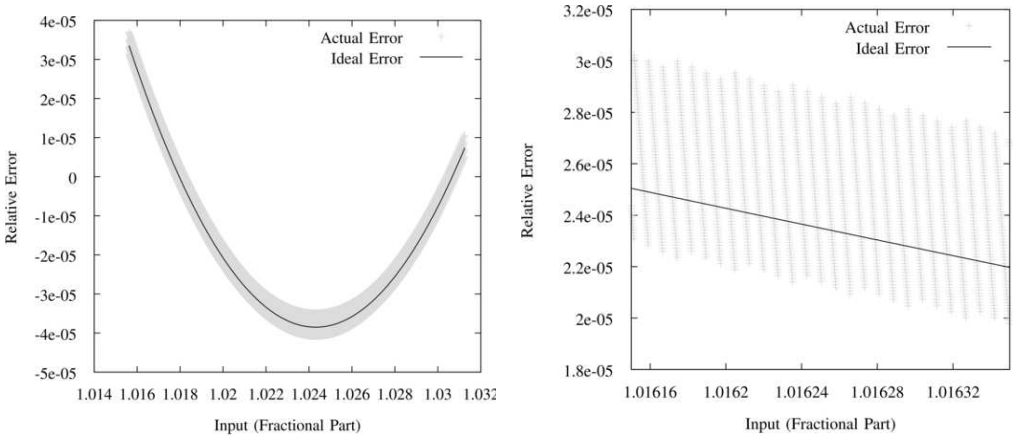
3.4.5 Nonlinear Initial Approximation

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 4. 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 fails 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 4(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 such an order that all the solutions in both intervals fall outside the endpoints. We make the following observations:

- In intervals where ϕ shows a decreasing trend (i.e., 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) Relative error over the entire sector.

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

Fig. 4. 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.

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 Summary and Future Work

We presented a comprehensive verification solution for floating point division embodied in the framework of the FPgen test plan and generator. This solution includes a collection of coverage models and the solving capabilities to cover them, providing treatment of a large variety of corner cases addressing different aspects of the division operation.

We pointed out that some of our algorithms are usually efficient, but still may fail to yield a timely solution for particular cases. There remains room for identifying these cases and providing an efficient alternative to handle them.

We also pointed out our lack of implementation specific models for decimal division. More study is required to advance our understanding of this area before we can formulate the appropriate models.

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*, in IEEE Std. 754-2008 (2008), pp. 1-58.

[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] 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.

[5] 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)*.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] J. Harrison, "Isolating critical cases for reciprocals using integer factorization", *Proc. 16th IEEE Computer Arithmetic Symp. (ARITH 16)*, 2003, pp. 148-157.

[12] R. Grinwald, E. Harel, M. Orgad, S. Ur, A. Ziv, "User Defined Coverage—a Tool Supported Methodology for Design Verification", in *Proc. 35th Design Automation Conf. (DAC)*, pp. 158-165, June 1998.

[13] A. Y. Khinchin, *Continued Fractions*, 1997, Dover, New York.

[14] M. Aharoni, R. Maharik, A. Ziv, "Solving Constraints on the Intermediate Result of Decimal Floating-Point Operations", in *Proc. 18th IEEE Computer Arithmetic Symp. (ARITH 18)*, 2007, pp. 38-45.

[15] C. Iordache and D. W. Matula, "On Infinitely Precise Rounding for Division, Square Root, Reciprocal and Square Root Reciprocal", in *Proc. 14th IEEE Computer Arithmetic Symp. (ARITH 14)*, 1999, pp. 233-240.

[16] M. Aharoni, S. Asaf, R. Maharik, I. Nehama, I. Nikulshin, A. Ziv, "Solving Constraints on the Invisible Bits of the Intermediate Result for Floating-Point Verification", in *Proc. 17th IEEE Computer Arithmetic Symp. (ARITH'05)*, 2005, pp. 76-83.

[17] E. Guralnik, A. J. Birnbaum, A. Koyfman, A. Kaplan, "Im-

plementation Specific Verification of Divide and Square Root Instructions", in *Proc. 19th IEEE Computer Arithmetic Symp. (ARITH 19)*, 2009, pp. 114-121.

[18] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek, "Constraint-Based Random Stimuli Generation for Hardware Verification", in *AI Magazine* 28(3), 2007, pp. 1330.

[19] A. Aharon, Y. Lichtenstein, Y. Malka, "Model-Based Test Generator for Processor Design Verification", in *Innovative Applications of Artificial Intelligence (IAAI)*, 1994