Solving Constraints on the Intermediate Result of Decimal Floating-Point Operations

Merav Aharoni, Ron Maharik, Abraham Ziv
IBM Research Lab in Haifa
email: merav@il.ibm.com

Abstract

The draft revision of the IEEE Standard for Floating-Point Arithmetic (IEEE P754) includes a definition for decimal floating-point (FP) in addition to the widely used binary FP specification.

The decimal standard raises new concerns with regard to the verification of hardware- and software-based designs. The verification process normally emphasizes intricate corner cases and uncommon events. The decimal format introduces several new classes of such events in addition to those characteristic of binary FP.

Our work addresses the following problem: Given a decimal floating-point operation, a constraint on the intermediate result, and a constraint on the representation selected for the result, find random inputs for the operation that yield an intermediate result compatible with these specifications.

The paper supplies efficient analytic solutions for addition and for some cases of multiplication and division. We provide probabilistic algorithms for the remaining cases. These algorithms prove to be efficient in the actual implementation.

1 Introduction

The previous IEEE standard 754 [2] specified the formats and behavior for binary floating-point (FP). The draft for the revised standard P754 [3] specifies them both for binary and decimal FP. Many of the earliest computers used decimal arithmetic, but computer hardware gradually evolved toward usage of binary arithmetic, to the point that today most computers support only binary arithmetic in hardware [6].

Decimal arithmetic, both integer and FP, has widespread applications, primarily financial and commercial. Usage of binary FP in these applications implies inexact conversions between binary and decimal representations, and roundoff in binary operations, which change the data in ways that are hard to understand and debug; therefore decimal arithmetic is supported through software on most machines. Recently, there is renewed interest in usage of decimal FP implementations both in software [5], [7] and hardware [8], [10].

Verification of binary FP hardware is known to be an intricate problem. Both formal methods and simulation methods have been developed to deal with this challenge. Verification by simulation cannot cover the entire FP domain, which is huge and involves many corner cases. For this purpose, coverage models are developed [17], that define interesting cases for verification purposes. A coverage case is said to be *covered* if we have at least one test that hits this case. Coverage cases can be defined on different levels: they can be defined in English, in terms of the implementation signals, or in some formal or mathematical expression language. The language determines to a great extent the kind of cases that can be covered.

Generating the test cases is often a manual process, which is quite difficult and limits the number of test cases that can be produced. Software tools have been developed in order to cope with this problem, for example [4, 11, 13, 15, 16, 19].

Several papers propose algorithms that generate random solutions for interesting special cases of constraints on binary FP operations. One example can be found in [12], which describes algorithms for solving constraints on the unbounded intermediate result of arithmetic operations on binary FP numbers.

As decimal FP is a newly defined format, it lacks the existing technology developed for binary FP verification. Decimal FP has several notable differences as compared to binary FP. The most obvious is that the significand comprises decimal digits rather than binary bits. The second major difference is that for decimal FP numbers, the representation of the number is deemed important, in addition to its numerical value. The set of all representations of a single numerical value is known as the *cohort* of this value.

The standard allow two representations: a *normalized format*, in which the significand is in the range $[0,10)$ and has the form $d_0.d_1 \ldots d_{p-1}$, and an *unnormalized format*, in

which the significand is an integer in the range $[0, 10^p - 1]$. In the discussion that follows, we choose to use the unnormalized representation. Since most of the algorithms in this paper involve calculations on the significand, it is more convenient to deal with it as an integer, rather than as a fraction.

Trailing zeros in the significand of a decimal FP number are meaningful; for example, $(2 \cdot 10^0) \cdot (3 \cdot 10^0) = 6 \cdot 10^0$, but $(20 \cdot 10^{-1}) \cdot (30 \cdot 10^{-1}) = 600 \cdot 10^{-2}$. The numerical value of these two results is the same, but the values are considered different in the sense that they represent different degrees of precision in the result. To quantify the definition of the precision of the result, the IEEE Standard P754 [3] defines a *preferred exponent* for the result of every arithmetic operation. The preferred exponent uniquely defines the representation selected for the result among all members of its cohort.

When defining test cases for FP, input constraints provide only limited capabilities. Targeting outputs, or in particular, intermediate results, allows much better control over FP corner cases. In this paper, we define constraints on the intermediate result of arithmetic operations.

For example, in a decimal FP system with 4 digits of precision, an intermediate result of 9999.9 may have to be shifted one digit to the right after it is rounded to the target precision, therefore this is an interesting result for testing the shifter mechanism. An intermediate result such as 2222.50 can be used test the rounding mechanism, as it is on the threshold between results that are rounded up and those rounded down.

In addition to a constraint on the intermediate result, we define a constraint over the difference between the actual exponent of the intermediate result and the preferred exponent. This allows us to test various uncommon scenarios and to verify that the correct representation of the result value is selected.

As an example, consider a coverage model that tests all possible values of the exponent difference in a division operation. Suppose that the decimal format in question supports 4 digits of precision. As we shall see in Section 5, the exponent difference in this case is between 0 and 7. To generate the extreme case where the difference is 7, the dividend's significand must be selected so that it has 3 leading zeros and only 1 significant digit. The divisor must have 4 significant digits, and the resulting quotient must have at least 4 significant digits. Here is one possible solution: $1 \cdot 10^0 \div 3200 \cdot 10^0 = 3125.0 \cdot 10^{-}7$. The preferred exponent, as defined in Section 2, is 0.

Constraints on the allowed intermediate results and exponent differences can appear as ranges, masks, or other set definitions. We choose to deal with the strictest constraint among these, namely a specific intermediate result and a specific difference between actual and preferred exponents. Any other constraint type can be reduced to this problem by first selecting a specific value from the set of allowed intermediate results and then calling the suggested algorithms.

We provide algorithms that, given a specific intermediate result and difference between actual and preferred exponents, provide two inputs that yield the specified result. We provide algorithms for addition, multiplication, and division. These algorithms are random, in that successive usage potentially yields different pairs of inputs that generate the same result. This is important for verification purposes, allowing more complete coverage of the scenario space and faster uncovering of design bugs. We try to keep the selection of different inputs within the solution space as uniform as possible, and demand that no solutions be lost.

These algorithms were implemented in the context of a custom verification tool for FP designs, named FPgen [4]. The tool was used for the verification of the decimal FP implemented in millicode in IBM System z9 [9] and in verification of decimal FP hardware in IBM Power6 [14]. Test cases generated by this tool proved effective in uncovering design bugs, as well as in attaining functional coverage.

FPgen was also used to generate a test suite for the IEEE standard for binary and decimal FP. The test suite is based on a set of coverage models described in a document that can be found on the FPgen web site [1]. Each model in the document describes a set of interesting cases for testing. The full test suite for decimal FP has not yet been made public. A preliminary version of the decimal test suite can be found on the web site.

In Section 2, we provide some necessary definitions and formally define the problem. In Sections 3, 4, and 5, we present algorithms for addition, multiplication, and division operations, respectively. Section 6 provides a summary of the results and suggestions for future work in this area.

2 Problem Definition

We will need the following definitions before we proceed to define the problem:

Decimal Floating-point number: Using the notation of IEEE standard P754, 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 FP, decimal FP numbers are not normalized and as a result a single value may have multiple representations.

p (precision): The maximal number of digits in the significand.

Unbounded intermediate result: The result of a floating-point operation, assuming unbounded precision and unbounded exponent range.

Intermediate result: 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. If at least one of the remaining digits of the exact result, beyond d_p is non-zero, then σ is 1, otherwise σ is 0.

Exact result: An operation's result is said to be exact when the intermediate result's guard digit and sticky bit are both 0.

Preferred exponent: Because decimal FP numbers have multiple representations, the IEEE standard defines a *preferred exponent* for each operation. Denoting by e_x, e_y the exponents of the first and second operands of an arithmetic operation, the preferred exponent for addition and subtraction is $min(e_x, e_y)$, for multiplication it is $e_x + e_y$ and for division it is $e_x - e_y$. If the result is inexact, the representation of the final (and the intermediate) result will be chosen so as to lose the least number of significant trailing digits possible. If the result is exact, the representation of the final (and the intermediate) result will be selected so that the actual exponent is as near as possible to the preferred exponent. These principles are compromised near the boundaries of the legal range for decimal FP numbers, but such cases are beyond the scope of this paper.

Problem definition: *Given a decimal floating-point operation in $\{+, -, \times, \div\}$, 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 that, when combined by the given decimal floating-point operation, give an intermediate result, z, that is compatible with the constraints.*

3 Addition

We use addition in this context to mean the addition of two numbers with the same sign (or the subtraction of two numbers with opposite signs). The algorithm for subtraction is similar to the one we use for addition, although each case needs to be analyzed differently. We do not describe the algorithm used for subtraction. We also ignore the cases in which one of the operands is 0.

We denote the addend with the smaller exponent by x, and the addend with the larger exponent by y. S_x, S_y represent the significands of x and y respectively, and the significand of the intermediate result is denoted by S_z. S'_x and S'_y will denote the aligned coefficients so that the least significant digit of S'_x has the same decimal position as that of S'_y. Note that $S'_x = S_x$. The actual exponent of an addition operation cannot be less than the preferred exponent; otherwise, it would mean the intermediate result had extra trailing zeros, and these would obviously be shifted out to give an intermediate result with the preferred exponent. We use the notation $d =$ *actual exponent - preferred exponent*.

We divide the problem into four subcases:

Case 1: Result is exact and the actual exponent equals the preferred exponent.

Since the guard and sticky bit are 0, this case can be viewed as a problem of decimal integer addition. For S_x, we randomly select any decimal integer less than S_z. We calculate S_y as $S_z - S_x$. As the base exponent, each exponent is assigned the exponent of the intermediate result.

Next we select one of the operands for a possible exponent shift. Exponent shift is possible only if the operand has trailing or leading zeros, where we take leading zeros to mean the number of digits is less than p. If exponent shift is possible both for S_x and for S_y, randomly select one of them. If it is possible for only one of them, select it. Otherwise, the final exponents equal the base exponent. For the operand selected, determine the set of all possible results by shifting the result one position left for every leading zero and one position right for every trailing zero. The exponents are adjusted accordingly. Select one result out of the set of all such possible results. This freedom in the choice of exponents is similar in the cases that follow. We leave this part up to the reader, and explain in detail only how to select the significands of x and y.

Case 2: Result is exact and the actual exponent differs from the preferred exponent.

We calculate an upper bound on the number of trailing zeros that were lost. Since the guard digit and sticky bit are 0, therefore the most significant digit of the smaller operand cannot be positioned to the right of the least significand digit of the intermediate result. So the number of trailing zeros is at most $p-1$. If the exponent, e, of the intermediate result is near *emin*, then the number of trailing zeros is at most $e - emin$, because the trailing zeros must be derived from digits of one of the operands.

We set the number of trailing zeros in the intermediate result to d, and create a new intermediate result significand, $S'_z = S_z \cdot 10^d$. We differentiate between the case where $|S'_z| = max|S_x|, |S'_y|$ and the case $|S'_z| = max\{|S_x|, |S'_y|\} + 1$, the latter case indicating the operation produced carry.

Carry is possible only when the intermediate result has the following form:

$$10^{p-1} \leq \frac{S_z}{10} \leq 10^{p-1} + 10^{p-d} - 2.$$

If this condition is true, we calculate the proportion of cases in which carry was generated out of all possible S_x, S_y whose sum is S_z. We randomly decide whether to apply the carry conditions or not according to this proportion. If the carry condition holds, $\frac{S_z}{10} = 10^{p-1} + C$ for some $C \leq 10^{p-d} - 2$.

To generate carry, the portion of S_x without the trailing zeros, $\frac{S_x}{10^{d-1}}$, must satisfy

$$10 \cdot C + 1 \leq \frac{S_x}{10^{d-1}} \leq 10^{p-d+1} - 1.$$

If there are no S_x that satisfy this inequality carry is not

possible. The number of values of S_x that give carry is

$$10^{p-d+1} - 10 \cdot C - 2.$$

If there is no carry, S_x will have d trailing zeros; therefore the number of such significands is $10^{p-d} - 1$. From here we see that the proportion of carry cases out of the total number of possible values for S_x is

$$\frac{10^{p-d+1} - 10 \cdot C - 2}{11 \cdot 10^{p-d} - 10 \cdot C - 3}.$$

It is easy to see that when C is large, carry cases will be rare and when C is small, most cases will be carry. Having selected the carry case or non-carry case according to the above proportion, we proceed to select S_x and S_y as follows.

In the case of no carry, S_x must have at least d trailing zeros. S_z must have p digits, otherwise it would have been shifted left to allow more trailing zeros, thus decreasing the actual exponent. Therefore, S_y must have p digits, as there is no carry in this case. This imposes the following limitation on the $p-d$ most significant digits of S_x:

$$\frac{S_x}{10^d} \leq S_z - 10^{p-1}.$$

We randomly select such a significand S_x and subtract it from S'_z to obtain S'_y.

In the case of carry, S_x must have at least $d-1$ trailing zeros. Since S'_z has $p+d$ digits, S'_y has $p+d-1$ digits, therefore

$$10^{p+d-2} \leq S'_y \leq 10^{p+d-1}.$$

From here we get the following condition on S_x:

$$S'_z - 10^{p+d-1} < S_x \leq min\{10^p - 1, S'_z - 10^{p+d-2}\}.$$

Dividing by 10^{d-1}, we get the following condition on the leading non-zero digits of S_x (in addition to the conditions which must hold for carry):

$$10S_z - 10^p < \frac{S_x}{10^{d-1}} \leq min\{10^{p-d+1} - 1, 10S_z - 10^{p-1}\}.$$

We select such an S_x, add $d-1$ trailing zeros to it, and then compute $S'_y = S'_z - S_x$. Finally, we remove $d-1$ trailing zeros, and get S_y.

Example. Let $p = 4$, $d = 3$ and $z = 12340 \cdot 10^{-1}$, and let's assume we would like to create an addition operation that produces carry. We check if carry is possible, that is if $\frac{S_z}{10} = 1234$ is in the range $[10^{4-1}, 10^{4-1} + 10^{4-3} - 2] = [1000, 1008]$. This condition does not hold, so we cannot fulfill this constraint. Let us select instead a value that is in the range allowing carry, for example let's select $S_z = 10050$, i.e., $S'_z = 1005000$. S_x must have at least 2 trailing zeros, and $\frac{S_x}{100}$ must be in the range $[10 \cdot 5 + 1, 10^{4-3+1} - 1] = [51, 99]$. Let's choose the smallest possible value, i.e., $S_x = 5100$. By subracting S_x from S'_z we obtain $S'_y = 999900$, and finally $S_y = 9999$.

Case 3: Result is inexact but the sticky bit is 0, d > 0.

The guard digit is non-zero and S'_z has $p+d$ digits, including exactly $d-1$ trailing zeros, where $d \geq 1$. S_x must have the same number of trailing zeros as S'_z, that is $d-1$. The least significant non-zero digit of S_x must be the same as the guard digit of S_z. As before, we observe two possible cases here: the case in which the operation did not create carry and the case in which it did. The condition for carry and the proportion of cases with carry are calculated as in Case 2.

In the case of no carry, S'_y must have at least d trailing zeros, and a total of $p+d$ digits. In other words, $\frac{S'_y}{10^d}$ is an integer and $10^{p-1} \leq \frac{S'_y}{10^d} < 10^p$. By substituting the lower and upper bounds on S'_y in the equation $S_x + S'_y = S'_z$, we get the following condition on the leading portion of S_x (without the trailing zeros):

$$S_z - 10^{p+1} \leq \frac{S_x}{10^d - 1} < 10^{p-d}.$$

We choose S_x in this range and then compute $S'_y = S'_z - S_x$.

The case of carry is similar, the only difference being that S'_y must have at least $d-1$ trailing zeros and a total of $p+d-1$ digits.

Case 4: The result is not exact and the sticky bit is 1.

This case implies $d \geq 2$ and the number of digits of S'_z equals $d+p$. We select the solution from the following three possible subcases:

d > p : In this case S'_z must be composed of three disjoint substrings. A *head* of at most p digits that are equal to the digits of S_y, a *tail* of at most p digits that are equal to the digits of S_x, and a *middle*, which is all zeros. This solution is possible if and only if the guard digit of S_z is 0.

d = p : In this case, we observe two possible subcases: that in which the number of digits of S'_z equals the number of digits of S'_y, and that in which S'_z has one digit more. In the first subcase, the solution is similar to that of the previous case, the only difference being that the middle part has zero length. In the second subcase, the addition operation must have created carry. Both S_x and S_y must have p digits, and the lowest non-0 digit of S'_y must have the same position as the highest digit of S_x, which is the position of the guard digit of S_z. The sum of these two digits must create a carry that is propagated through all the digits of S'_y to create a carry on the complete addition operation. This can only occur if S_y has the form $99...9y$, where only the least significant digit is free (though it cannot be 0). This only occurs if S_z has the form $100...0z$, where z is the guard digit, and $z \neq 9$. If S_z indeed has this form, then the lower digits of S'_z can be selected at random. S_x is constructed by setting its most significant digit to be greater than the guard digit

of S_z. The remaining digits of S_x must be identical to the lower digits of S'_z. As usual, we compute S'_y by subtracting S_x from S'_z.

d < p : Once more, we divide this into the subcase in which the number of digits of S'_z equals the number of digits of S'_y, and the subcase in which S'_z has one digit more. In the first subcase, the lower n digits of S_x coincide with the lower digits of S'_z, and we are free to choose them at random. The lower d digits of S'_y are 0, and $10^{p-1} \leq \frac{S'_y}{10^d} < 10^p$. From here, we get the following condition on S_x:

$$S'_z - 10^{p+d} < S_x \leq min\{10^p - 1, S'_z - 10^{p+d-1}\}.$$

Computing S_y is done as in previous cases. In the second subcase, the lower $d-1$ digits of S_x coincide with the lower digits of S'_z, and we are free to choose them at random. The lower $d-1$ digits of S'_y are 0, and $10^{p-1} \leq \frac{S'_y}{10^{d-1}} < 10^p$. From here we get the condition $S'_z - 10^{p+d-1} < S_x < 10^p$. Furthermore, for carry to be possible, we notice S'_z must have the form $100\cdots0zz\cdots z$, where the number of zeros is $d-1$.

4 Multiplication

The result of a multiplication operation often requires more than p digits and at most $2p$ digits. Therefore the difference between actual result and preferred result is $0 \leq d \leq p$. We use the notation S'_z to represent the significand of the unbounded intermediate result, which has up to $2p$ digits. The main distinction here is between exact intermediate results, which require factorization, and inexact results.

Case 1: Sticky bit is 0

An exact intermediate result can mean either the guard and sticky are both zero, or the guard is non-zero and the sticky is zero. $d-1$ defines the number of trailing zeros that are in S'_z beyond the guard digit. In this case, we have $S_x \cdot S_y = S_z \cdot 10^{d-1}$, where S_z is the significand of the intermediate result including the guard digit. We find the prime factors of S'_z by some factoring method; for example, the quadratic sieve method [18]. Another option is to first remove all the trailing zeros from S'_z, factorize the remaining number, and for every trailing zero to add a 5 and a 2 to the list of factors.

Finally, we construct S_x and S_y by distributing the factors between the two operands, while making sure each of them remains smaller than 10^p. If that is not possible, no solution exists. Finally, we select the operand exponents so that $e_x + e_y = e_z - d$.

Case 2: Sticky bit is 1

Note that in this case we must have $d \geq 2$, since a significant digit was shifted out beyond the guard digit when forming the intermediate result.

The following inequality holds:

$$S_z < \frac{S_x \cdot S_y}{10^{d-1}} < S_z + 1$$

therefore

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

We use the following straightforward algorithm:

1. Compute the range of possible values for S'_z, i.e., $(S_z \cdot 10^{d-1}, (S_z+1) \cdot 10^{d-1})$.
2. Select the number of digits of S_y.
3. Select a random value for S_y.
4. Compute the range of possible values for S_x, i.e., $(\frac{S_z \cdot 10^{d-1}}{S_y}, \frac{(S_z+1) \cdot 10^{d-1}}{S_y})$.
5. If a decimal integer exists in this range, then set S_x to that number and we have found a solution ; Otherwise, return to step (2).

Steps (2) and (3) require further explanation. We need to determine how many digits S_y can have. We use the notation $|S_x|, |S_y|, |S_z|$ to signify the number of digits in S_x, S_y, and S_z respectively. There are two cases: $|S_x| + |S_y| = |S_z| = p + d$ (the case of a product with carry), or $|S_x| + |S_y| = |S_z| + 1 = p + d + 1$ (a product without carry). In other words, $|S_y| = p + d - |S_x|$ or $|S_y| = p + d - |S_x| + 1$. Since $|S_x| \leq p$ and $|S_y| \leq p$, we have $d \leq |S_y| \leq p$ or $d + 1 \leq |S_y| \leq p$.

In step (3), if we select S_y such that $|S_y| \geq d + 1$, there is no restriction on the values of S_y, since these may either produce carry or not. However, if $|S_y| = d$, we need to make sure this is a case with carry, otherwise we get an S_x that has too many digits. To do this, we divide the minimum value for S'_z by the maximum value for S_x, which is $10^p - 1$. This gives us a lower limit on the possible values of S_y.

It remains to show that the number of iterations of this loop is not too large; that is, we find a solution within a reasonable amount of time. To show this, we calculate the probability that an integer S_x exists in any given iteration. We compute this by calculating the size of the range of solutions for S_x divided by the distance between two decimal machine numbers, assuming a uniform distribution of the solutions.

S_z has the form $zz\cdots z$, where the rightmost z is the guard digit and the number of digits is $p+1$. S'_z is in the range $[zz\cdots z00\cdots01, zz\cdots z99\cdots99]$, where the number of zeros equals $d-2$ and the number of 9s equals $d-1$. The size of the interval of possible S'_z is therefore $99\cdots98$ where the number of 9s is $d-2$. To compute the size of the interval of possible S_x, we divide the size of the interval on S'_z by S_y that was selected. The size of the interval of possible S_x is approximately $\frac{10^{d-1}}{10^{|S_y|}}$. The distance between two

possible machine numbers on this scale is 1. Therefore, the size of the interval represents the probability that a machine number, S_x, exists in this interval, assuming a uniform distribution of the intervals on the line of machine numbers.

In the worst case, when $d = 2$ and $|S_y| = p$, we get a probability of $\frac{1}{10^{p-1}}$ that such an S_x exists. It is the scenario where the unbounded intermediate result has $p+2$ digits, and one of the significands has p digits. It is intuitively quite unlikely to find a solution in this case. At the other extreme, when d is unrestricted, or $d = p$, the size of the interval is approximately $\frac{1}{10}$. This probability is quite reasonable. On the average we find a solution for S_x within 10 trials. For small values of d, the specified algorithm can be improved by choosing $|S_y|$ so that it is less than or equal to $|S_x|$. This does not harm the uniformity of the solutions because S_x and S_y can be swapped at the end. An additional improvement can be achieved by selecting a very small $|S_y|$. However, this impacts the uniformity of solution.

Example. Let $p = 4$, $d = 2$, and $z = 89030 \cdot 10^{-1}$ with $\sigma = 1$. This means S'_z is in the range $[8903001, 8903099]$. Let's select S_y to have d digits. This means both carry and non-carry cases are possible. Let's assume we want a carry case. The restriction on S_y is $S_y \geq \frac{8903001}{9999} \approx 890.389$. We select $S_y = 935$. The range of possible real values for S_x is approximately $[9521.93, 9522.03]$. Since this range contains an integer, S_x can be chosen to be 9522.

5 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 (as opposed to the cases of addition and multiplication). We denote the exponent difference $d =$ *preferred exponent - actual exponent*.

The division problem is partitioned into three subcases.

Case 1: The exponent difference is 0 and the result is exact.

This case can be viewed as an integer division. The guard digit does not come into play and must be 0. Let S'_z denote the integer value of S_z with the guard digit removed. S'_z has p digits at the most. 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 $1 \leq S_y < \frac{10^p}{S'_z}$. 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 2: The sticky bit is 0, and either the exponent difference is not 0 or the result is not exact (i.e., the guard digit is non-zero).

The following equation, based on the definition of decimal division and preferred exponent, applies to any case where the sticky bit is 0:

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

therefore

$$S_x \cdot 10^{d+1} = S_y \cdot S_z \tag{2}$$

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 1).

- 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 (2) – 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.

Formation of the solution is based on Equation (2) 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 to $S'_z \cdot 2^{max(0,j-d-1)} \cdot 5^{max(0,k-d-1)}$. If S_x has more than p digits, there is no solution.
2. Set S_y so that Equation (2) holds: $S_y = 2^{max(0,d+1-j)} \cdot 5^{max(0,d+1-k)}$.
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 sizes less than 10^p.
4. Choose exponents e_x and e_y so that $e_x - e_y = e_z + d$.

Example. Let $p = 4$, $d = 3$ and $z = 43750 \cdot 10^{-1}$. Since $S_z = 43750 = 7 \cdot 2^1 \cdot 5^5$, we have $S'_z = 7$, $j = 1$ and $k = 5$. Initialize S_x to $7 \cdot 2^{max(0,1-3-1)} \cdot 5^{max(0,5-3-1)} = 35$. Set $S_y = 2^{max(0,3+1-1)} \cdot 5^{max(0,3+1-5)} = 8$. To randomize, multiply S_x and S_y by the random factor 62, obtaining a solution of $S_x = 2170, S_y = 496$. Note that both significands conform to the 4-digit precision bound. Finally, set the exponents e_x and e_y so as to uphold the equality $e_x - e_y = e_z + d = 0 + 3$, for example: $e_x = 8, e_y = 5$, and

obtain $x = 2170 \cdot 10^8, y = 496 \cdot 10^5$. The real quotient of these numbers is $4.375 \cdot 10^3$. The significand is shifted left to allow minimum loss of precision. Therefore the result is re-aligned to $4375 \cdot 10^0$, which corresponds to the intermediate result $43750 \cdot 10^-1$ as required.

Case 3: The sticky bit is 1.

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 left-most position of S_z.

The following relationship holds in this case:

$$S_z/10^{d+1} < S_x/S_y < (S_z+1)/10^{d+1} \tag{3}$$

therefore

$$S_z \cdot S_y < S_x \cdot 10^{d+1} < (S_z+1) \cdot S_y \tag{4}$$

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. The exponent difference depends only on $|S_x|$, $|S_y|$, and the relative magnitude of S'_x and S'_y. Specifically: if $S'_x > S'_y$ ("no-borrow" case), then $d = p - |S_x| + |S_y| - 1$, and if $S'_x < S'_y$ ("borrow" case), then $d = p - |S_x| + |S_y|$. Note that the case $S'_x = S'_y$ is irrelevant, since the result would not have its sticky bit set.

From these relations we learn that d belongs to $\{0, 1, ..., 2p-1\}$, where $d = 0$ only applies in the no-borrow case, and $d = 2p-1$ only applies in the borrow case. Other values of d may occur with or without borrow.

Example. Let $p = 4$, let $S_x = 4321$ and $S_y = 59$. Then $|S_x| = 4$, $|S_y| = 2$, and S'_x and S'_y are 4321 and 5900 respectively. The result of dividing the two significands is approximately 73.237288. The amount of alignment required to preserve maximum precision is therefore $d = 2$, which is consistent with the formula given above for the "borrow" case. It is not difficult to observe that selecting different digits for S_x and S_y would have no effect on the resulting value of d, so long as the quotient is not exact and the borrow case is maintained. Switching to the "no-borrow" case, however, results in $d = 1$, for example: $S_x = 4567$, $S_y = 14$, $\frac{S_x}{S_y} \approx 326.214286$.

The solution proceeds as follows:

1. Calculate $t = (-|S_x| + |S_y|)$ according to the borrow-case relation above: $t = d - p$. (this does not necessarily make the solution a borrow case).

2. Use the value of t to determine upper and lower bounds for $|S_y|$. Given that both $|S_x|$ and $|S_y|$ must lie in $\{1, 2, ..., p\}$, we have:

   $$|S_y| = t + |S_x|$$
   $$\Rightarrow\ t+1 \le |S_y| \le t+p$$
   $$\Rightarrow\ 1 + \max(0,t) \le |S_y| \le p + \min(0,t)$$

   for the no-borrow case, we should add 1 to the value of t. The final legal range of $|S_y|$ is the union of the two ranges:

   $$1 + \max(0,t) \le |S_y| \le p + \min(0,t+1)$$

3. Choose a value for $|S_y|$ from the resulting range.

4. Choose a random S_y with the selected number of digits.

   If the maximal allowed value is chosen for $|S_y|$, and is not the trivial maximum p, then only the no-borrow case is possible. We reduce the allowed range of S_y so as to ensure that the operation does not cause borrow. We calculate the bound of $\frac{S_x \cdot 10^{d+1}}{S_z+1}$, which serves as an upper bound for S_y. This bound is reached when S_x is maximal, i.e., $S_x = 10^p - 1$, therefore:

   $$S_y \le \frac{(10^p - 1) \cdot 10^{d+1}}{S_z + 1}$$

5. Calculate lower and upper bounds, α and β, for $S_x \cdot 10^{d+1}$ using Equation (4).

6. 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.

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

Example. Let $p = 4$, $d = 5$ and $z = 13198 \cdot 10^{-1}$ with a sticky bit value of 1. By definition we have $t = d - p = 1$. By step 2 we have: $2 \le |S_y| \le 4$. We randomly choose to set $|S_y| = 4$. By step 4, we may choose any digits for S_y. Let $S_y = 2576$. By equation (4) we have: $[\alpha, \beta] = [33998048, 34000624]$. This range includes the value 34000000, which has $6 = d+1$ trailing zeros. We now set $S_x = 34$, and select appropriate exponents to arrive at the required solution. Note that this is a "no-borrow" solution.

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

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

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.

6 Conclusion

We presented a method for defining interesting verification test cases for IEEE Decimal Floating-Point operations. This definition includes specification of a constraint on the intermediate result of the operation, as well as on the difference between the actual exponent of the result and the preferred exponent.

We proposed algorithms that solve such constraints for addition, multiplication, and division. The algorithms assume the most difficult type of constraint, i.e., a single intermediate result with a single exponent difference. The algorithms for addition, and cases of multiplication and division where the sticky bit is 0, are guaranteed to find a solution efficiently when one exists. Multiplication and division with a sticky bit value of 1 are solved probabilistically, using methods that have been empirically shown to be practical in all but the most extreme cases.

These algorithms have been implemented in a the framework of a test generator for floating-point data. Experience has shown that the methods presented here are very effective in generating corner cases, covering segments of the FP domain as defined in various coverage models, and reconstructing known bugs so as to refine the faulty scenario and pinpoint the root cause.

The algorithms described in this paper apply to most forms of the general problem, but may not be suitable for some corner cases (e.g., when the inputs are subnormal numbers). Several refinements and heuristics have already been identified and included in our implementations, to improve the handling of corner cases. There is room for further analysis of these cases, and development of efficient random algorithms for them. In addition, when the constraint is looser, such as a range of possible results, or no constraint on the exponent difference, it may be possible to find more efficient algorithms.

Also of interest are additional arithmetic instructions that are specified in the standard and not discussed in the paper, in particular fused multiply-add $(a \times b + c)$ and square root. An additional direction for development is to tackle various types of constraints, e.g., simultaneous constraints on the inputs and output, or constraints on the unbounded intermediate result. Most of the results can be generalized for unnormalized arithmetic using any radix r.

References

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

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

[3] "Draft Standard for Floating Point Arithmetic - P754", 2006. http://754r.ucbtest.org/drafts/754r.pdf.

[4] M. Aharoni, S. A. L. Fournier, A. Koifman, and R. Nagel. "FPgen - A Test Generation Framework for Datapath Floating-Point Verification". In *Proc. IEEE International High Level Design Validation and Test Workshop 2003 (HLDVT03)*, 2003.

[5] M. Cornea and C. Anderson. "Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic". *Real Numbers and Computers*, 2006.

[6] M. Cowlishaw. "Decimal Floating-Point: Algorism for Computers". In *Proc. IEEE 16th Symp. Computer-Arithmetic (ARITH16)*, pages 104 – 111, 2003.

[7] M. Cowlishaw. The decNumber C library, 2005. http://www2.hursley.ibm.com/decimal/decnumber.pdf.

[8] M. Cowlishaw, E. Schwarz, R. Smith, and C. Webb. "A Decimal Floating-Point Specification". In *Proc. IEEE 15th Symp. Computer-Arithmetic (ARITH15)*, pages 147 – 154, 2001.

[9] A. Y. Duale, M. H. Decker, H. G. Zipperer, M. Aharoni, and T. J. Bohizic. "Decimal Floating-Point in z9: An Implementation and Testing Perspective". *IBM Journal of Research and Development*, 51, 2007.

[10] M. Erle, E. Schwarz, and M. Schulte. "Decimal Multiplication with Efficient Partial Product Generation". In *Proc. IEEE 17th Symp. Computer-Arithmetic (ARITH17)*, pages 21 – 28, 2005.

[11] W. Kahan. "A Test for Correctly Rounded SQRT". http://www.cs.berkeley.edu/∼wkahan/SQRTest.ps.

[12] M. Aharoni, S. Asaf, R. Maharik, I. Nehama, I. Nikulshin, and A. Ziv. "Solving Constraints on the Invisible Bits of the Intermediate Result for Floating-Point Verification". In *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH17)*, pages 76 – 86, 2005.

[13] D. W. Matula and L. D. McFearin. "A p X p Bit Fraction Model of Binary Floating Point Division and Extremal Rounding Cases". *Theoretical Computer Science*, 291:159 – 182, 2003.

[14] B. McCredie. "POWER Roadmap". http://www2.hursley.ibm.com/decimal/IBM-Power-Roadmap-McCredie.pdf.

[15] L. D. McFearin and D. W. Matula. "Generation and Analysis of Hard to Round Cases for Binary Floating Point Division". In *15th IEEE Symposium on Computer Arithmetic (ARITH15)*, pages 119 – 127, 2001.

[16] M. Parks. "Number-Theoretic Test Generation for Directed Rounding". In *Proc. IEEE 14th Symp. Computer-Arithmetic (ARITH14)*, pages 241 – 248, 1999.

[17] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. "User defined coverage - a tool supported methodology for design verification". *Proc. 35th Design Automation Conference*, pages 158 – 163, 1998.

[18] D. R. Stinson. *"Cryptography Theory and Practice"*. CRC Press, second edition, 1996.

[19] B. Verdonk, A. Cuyt, and D. Verschaeren. "A Precision and Range Independent Tool for Testing FP Arithmetic: Basic Operations, Square Root and Remainder". *ACM TOMS*, 20, Number 1:92 – 118, 2001.