Solving Constraints on the Invisible Bits of the Intermediate Result for Floating-Point Verification

Merav Aharoni, Sigal Asaf, Ron Maharik, Ilan Nehama, Ilya Nikulshin, Abraham Ziv
IBM Research Lab in Haifa
Haifa University Campus, Haifa, 31905 Israel
email: merav@il.ibm.com

Abstract

Test generation for datapath floating-point verification involves targeting intricate corner cases, which can often be solved only through complex constraint solving.

In the process of calculating the result, we use an intermediate result whose significand comprises a finite number of bits and a sticky bit that is 0 *if and only if the intermediate result is exact. We refer to all the bits beyond those represented in the final result as the invisible bits. We deal with corner cases that can only be defined via constraints on the intermediate result.*

Our work investigates the following problem: Given a floating-point operation, and constraints on the invisible bits and the sticky bit, find two inputs for the operation that yield an intermediate result compatible with the constraints.

The paper supplies a deterministic solution for addition and subtraction, and probabilistic solutions for multiplication and division. It also discusses the application of these algorithms to the verification of floating-point implementations.

1 Introduction

Verification of the floating-point unit 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 that need to be targeted, and from the intricacies of the implementation of floating-point operations. Verification by simulation involves executing a subset of tests that is assumed to be a representative sample of the entire test-space. In doing so, we would like to be able to define a particular subspace that we consider to be "interesting" in terms of verification; we can then generate tests selected at random out of the subspace.

In general, a *floating-point test subspace* is defined by specifying a floating-point operation and a set of constraints on the inputs and on the final result. We then generate a *representative sample* for the test subspace. This representative sample is a set of instances that belong to the test subspace.

Test subspaces defined by constraints on input and output operands are relatively straightforward. However, there are many interesting cases that cannot be defined solely in terms of input or output constraints. Let us look at one such example involving verification of the rounding process. *Hard to round* cases are those in which the infinitely precise result of the operation is very close, but not equal to, a rounding boundary value (i.e., a floating-point number or the midpoint between two floating-point numbers). In such a case, the decision about the side of the rounding boundary on which the exact output must fall, requires a high level of accuracy. A mistake in this decision may cause incorrect rounding and therefore, an incorrect result. The problem of verifying *hard to round* cases has received much attention, both in practice and in academia ([8], [9], [5]).

This test subspace cannot readily be defined solely in terms of input and output constraints, because it involves constraints on bits that were truncated during the rounding stage. We name these bits *invisible bits*. Therefore, we define a new operand that corresponds to the infinitely precise result of the operation. We name this operand the *intermediate result*. Assuming the precision of the final result is p, the significand of the intermediate result will have more than p bits. To account for the finiteness of the intermediate result, we add a sticky bit whose value is 0 if and only if the intermediate result is exactly equal to the unbounded result. Hard to round cases can be defined using constraints on the intermediate result's invisible bits and on the sticky bit. For example, an infinitely precise result that is just below a floating-point number can be specified using the following constraint: The invisible bits have the pattern 111...111 and the sticky bit is 1. Our definition of a test subspace can now be extended to include constraints on the intermediate result.

Although the general problem of providing random solutions for a given test subspace is NP-complete, it is pos-

sible to provide algorithms that generate random solutions for many interesting special cases of subspaces. One example can be found in [12], which describes a test generator for the *add* instruction, where the input and result operands are described as masks. Another example is found in [11], which describes a test generator for the *add, subtract, divide,* and *multiply* instructions, where the input and result operands are constrained to given ranges.

We focus on solving constraints on the invisible bits, including the sticky bit, of the intermediate result, so that each bit is individually constrained to 0 or 1, or else is unconstrained. Using constraints defined in this manner, we can target a large variety of test subspaces, including intermediate results that are very close to floating-point numbers (the hard to round cases). Another interesting test subspace involves cases in which only one bit affects the setting of the inexact-bit. In these cases, the invisible bits and the sticky bit are all zeros, except for a single bit that is set to 1.

The algorithms presented in this paper, and the algorithms in [12], [11], have been implemented in FPgen [2]. FPgen is an automatic floating-point test-generator that receives as input the description of a floating-point subspace and generates a random test case out of this subspace. FPgen employs a variety of algorithms, both analytic and heuristic, to solve the various constraint types. When constraints are given on the invisible bits of the intermediate result, FPgen employs the algorithms described in this paper to generate the test cases.

In Section 2, we formally define the problem. In Sections 3, 4, and 5, we present algorithms for multiply, add and subtract, and divide operations, respectively. We end each of these sections with a short discussion about the algorithm's complexity. Section 6 includes a summary of the results and suggestions for future work in this area.

2 Problem Definition

2.1 Notations and Definitions

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

- **Floating-point number:** Using the notation of IEEE standard 754 [1], a floating-point number is defined by $(-1)^s 2^E (b_0.b_1 b_2 \ldots b_{p-1})$ where s is the sign, E is the (unbiased) exponent, $E_{min} \leq E \leq E_{max}$, and $b_0.b_1 b_2 \ldots b_{p-1}$ is the significand with b_i being the binary digits, $b_i = 0$ or 1.
- **p (precision)**: This signifies the number of bits in the significand.
- **Exact 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, and a significand, S, of the form $1.b_1 b_2 \ldots b_{q-1}\sigma$ where $q > p$ and σ is the sticky bit ($\sigma = 0$ or 1). Each bit b_i, of the intermediate result is identical to the corresponding bit of the exact result of the operation. If at least one of the remaining bits of the exact result, beyond b_{q-1} is 1, then σ is 1, else σ is 0.
- **Operand bits:** The p most significant bits in the significand of the intermediate result.
- **Invisible bits:** The $q - p$ least significant bits in the significand of the intermediate result.
- **Mask:** A set of binary numbers, represented by a string containing the characters $\{0, 1, x\}$. A bit set to 0 or 1 must always take on that value, and a bit set to x can take on either value. A binary number is said to be *compatible* with a mask if it belongs to the set defined by the mask.

2.2 Invisible Bits Problem

In this paper, we investigate the following problem:

Invisible bits problem: *Given a floating-point operation in $\{+, -, \times, \div\}$ and masks for the invisible bits and for the sticky bit of the intermediate result, find, if possible, two operands a and b that, when combined by the given floating-point operation, give an intermediate result, c, that is compatible with the masks, or else state that no solution exists.*

According to the above definition of the intermediate result, the parameter q can take on any value. For practical purposes, we find that there is only a limited range of values for q, as a function of p, that are of interest in terms of verification. We will investigate the problem with the largest value of q that is of interest for each operation. In the following sections, we will analyze the choice of the appropriate q for each operation.

For the sake of simplicity, we omit from our discussion denormalized numbers and cases of overflow or underflow. Hereafter, all floating-point numbers involved are considered normalized (namely $E_{min} \leq E \leq E_{max}$, $b_0 = 1$) as are all results of floating-point operations.

Normalized significands fall in the range [1, 2). In the algorithms for addition and multiplication, we scale up the floating-point input operands by a factor of 2^{p-1} and thus our calculations will involve binary integers instead of normalized fractions, without affecting the result. For division, we also use the scaling, but in this case, it is specified explicitly in the calculations.

3 Multiplication

3.1 Determining the Value of q

Since the input operands have p binary digits, the exact product has at most $2p$ bits; therefore, there is no point in taking $q > 2p$. We assume $q = 2p$, because if $q < 2p$ we can always add arbitrary bits $b_q \dots b_{2p-1}$ and solve the problem with an intermediate result of size $2p$ bits. Because $q = 2p$ and the product has at most $2p$ bits, $\sigma = 0$.

3.2 Algorithm

In the first stage, we select a specific value for the invisible bits, named $c_0 c_1 \dots c_{p-1}$, that is compatible with the invisible bits mask. Next, we execute the algorithm described below to find significands for the input operands.

We define the following problem:

Multiply Invisible Bits Problem: *Find two inputs for the multiply operation such that the significand of the intermediate result is of the form $x_0 x_1 \dots x_{p-1} c_0 c_1 \dots c_{p-1}$, where for each i, c_i is preset to 0 or 1.*

In order to solve the Multiply Invisible Bits Problem, we formulate the following integer problem:

Integer LSBs Problem Definition: *Find a random triplet (A, B, C) so that:*

1. *A and B are each p-bit wide positive integers*
2. *C is a $2p$-bit wide positive integer*
3. *The p least significant bits of C are $c_0 c_1 \dots c_{p-1}$*
4. *$A \times B = C$*

For the sake of simplicity, we assume that $c_{p-1} = 1$; in other words, C is odd. The extension to the case that $c_{p-1} = 0$ is straightforward and is covered in the full description of the algorithm below. Several different methods can be applied to solve this problem such as Hensel Lifting [9] and group theory. We use the group theory based method.

The set of odd integers $1, 3, \dots, 2^p - 1$ is a group over multiplication modulo 2^p; therefore, for every number A in this group, there exists an inverse, A^{-1} in the group, such that $A \times A^{-1} \equiv 1 (mod 2^p)$.

The inverse is calculated by relying on a basic theorem of group theory [7] that states that in any finite group, any element raised to the power of the group size equals 1. Hence, we calculate the inverse as $A^{-1} \equiv A^{S-1} (mod 2^p)$, where $S = 2^{p-1}$ is the size of the group in our case. This can be computed using fewer than 2p multiplications mod 2^p, by computing A^{2^i} for $i = 0, 1, \dots, p-2$, by repeated squaring and by multiplying the results.

For odd C, we can select at random any A out of the group, and then set B as $B = A^{-1} \times C (mod 2^p)$. $A \times B \equiv A \times A^{-1} \times C \equiv C (mod 2^p)$.

The following formally describes the solution.

Integer LSBs Problem Algorithm

1. Let $C' = C = c_0 c_1 \dots c_{p-1}$
2. $NumberZeros = 0$
   While C' is even,
   $C' := C' \div 2$;
   $NumberZeros := NumberZeros + 1$
3. Choose random $A = a_0 a_1 \dots a_{p-1}$, and set $a_{p-1} = 1$
4. $B := (A^{-1} \times C') mod 2^p$
5. Select a random k in the range $[0, NumberZeros]$
6. $A := (A \times 2^k) mod 2^p$
7. $B := (B \times 2^{NumberZeros-k}) mod 2^p$

Using the integer LSBs problem, we find significands for the input operands. We must also choose random signs and exponents. We omit this process, as it is straightforward. We can then construct the input operands.

The complexity of solving the integer LSBs problem is equal to that of finding the inverse number, which is $O(p)$ multiplication operations.

It is important to note that not every solution to the integer LSBs problem is a suitable solution to the invisible bits problem. The algorithm described does not ensure that A and B are each p bit wide. Furthermore, the product is not necessarily $2p$ bits wide, as required by our definition of the intermediate result. We need to modify the algorithm in order to obtain the additional requirements. In Step 3, when selecting our random A, we set a_0 to 1. Following the calculation of B, we test if the product has a leading 1. If not, we go back to Step 3, and select a new value for A.

Next, we compute the expected number of iterations needed to find such suitable A and B. We need to calculate $PR = Prob(AB \geq 2^{2p-1} | A \geq 2^{p-1})$. We conjecture that the random variable defined by $B = A^{-1}C$ is asymptotically independent of A. Though we have no proof, this conjecture is supported by experimental evidence. Using elementary calculus, we calculate $PR = \frac{Prob(AB \geq 2^{2p-1})}{Prob(A \geq 2^{p-1})} = \frac{1}{2} \ln 2 \approx 0.35$, therefore the expected number of iterations needed to find such a solution is 3.

4 Addition and Subtraction

4.1 Determining the Value of q

In order to ensure correct rounding, implementation of the add or subtract operation in hardware requires at least $p+2$ internal bits. However, it is quite common to use more bits than this value. This is done to simplify the hardware implementation. For example, a floating-point unit that implements the fused multiply-add operation $(k \times l + r)$, can use the same hardware to implement both the multiply-add operation and the add operation.

The algorithm we propose does not limit the value of

q. However, for practical purposes, we can safely assume $p+2 \leq q \leq 4p$, where c is some small constant, $c < p$.

4.2 General Algorithm

The general algorithm is based on enumerating all possible *exponent cases*. Each exponent case defines a specific difference (or range of differences) between the exponents of the input operands, a specific difference (or a range of differences) between the exponent of the intermediate result and that of the first input operand, and whether the operation is an effective addition or subtraction.

The major steps of the algorithm are:

1. Prepare a list of all possible exponent cases.
2. Randomly choose one exponent case.
3. Using the selected exponent case, build fixed point masks that represent the significands of the floating-point problem.
4. Find a random solution for the fixed point add operation using the fixed point generator (see [12]).
5. Using the chosen exponent case, convert the integer solution to a floating-point solution.

In Step 4 we invoke the *fixed point generator* as defined below:

Fixed point generator: *Given three masks of length N, for binary integers, M_x, M_y, M_z, and one mask, M_C, of length $N+1$, for a corresponding carry sequence, the fixed point generator either states that no solution exists or generates three binary integers x,y,z and a carry sequence that are compatible with their respective masks, and $x+y=z$.* For a detailed description of the fixed point generator, see [12].

In the following sections, we describe each of the steps in more detail.

4.3 Preparing a List of Exponent Cases (Step 1)

Each exponent case includes the following information:

1. The signs of operands a and b
2. A constraint on the difference between the input exponents, specified as $E_a - E_b = m$, as $E_a - E_b \geq m$ or as $E_a - E_b \leq m$
3. A constraint on the difference between the exponent of the intermediate result and that of the first input, specified as $E_a - E_c = n$ or as $E_a - E_c \geq n$

The exponent cases are divided into several categories, each of which is handled in a slightly different fashion. In what follows, we describe the different types of exponent cases. For any exponent cases, there also exists the symmetric case in which a and b and are interchanged. In addition, we ignore the sign of the operands. Instead, we differentiate only between effective addition and effective subtraction, assuming $a \geq b$. The number of different exponent cases we have is $3q+3p-1$.

We illustrate two of the nine possible categories in detail. In the examples below, we present the alignments of the significand using $p=5$ and $q=12$. The string 1AAAA represents the significand of a, The string 1BBBB represents the significand of b, and the string 1CCCCCCCCCCCTT...TT represents the significand of c, where c is the exact result. The bits TT...TT in c represent the bits of the exact result, beyond position $q-1$. The sticky bit of the intermediate result is calculated by taking the "OR" of these bits. We define the number of such additional bits as required according to the exponent case.

1. **Addition, no carry, shift is at most 1**: the operation is effective addition, $E_a - E_c = 0$, and $E_a - E_b$ enumerates on the values in the range $[1, q-1]$. This case includes $q-1$ different exponent cases. For example, when $E_a - E_b = 1$, the significand structure is:

   S_a: `+ 1AAAA00000000`
   S_b: `+ 01BBBB0000000`
   S_c: `+ 1CCCCCCCCCCCT`

2. **Subtraction, cancellation of one bit, shift is greater than 1**: the operation is effective subtraction, $E_a - E_c = 1$, $E_a - E_b \geq q+1$. This case includes only one exponent case. The significand structure is:

   S_a: `+ 1AAAA000000000000`
   S_b: `+ 00000000000001BBBB`
   S_c: `+ 01CCCCCCCCCCTTTTT`

The remaining categories for addition depend on whether or not carry was generated, and whether the shift was 0, 1, or greater than 1. The categories for subtraction depend on whether there was no cancellation, cancellation of one bit, or cancellation of more than one bit, and whether there was a shift between the operands. Of course, not all combinations of the above are possible.

4.4 Building an Integer Masks Problem (Step 3)

After choosing a particular exponent case, we build fixed point masks. These masks are fed into the fixed point generator in order to get a fixed point solution for the significands.

We changed the fixed point generator as follows:

Fixed point generator 2: *Given three masks of length N, for binary integers, M_x, M_y, M_z, and operation $op \in \{+,-\}$, the fixed point generator either states that no solution exists or generates three binary integers x,y,z which are compatible with their respective masks, and for which it holds that x op $y = z$.*

When op is $+$, we use the original fixed point generator and set the carry mask to $0xxx...xxx0$. When op is $-$, we change the order of the masks such that $M_x = M_y + M_z$, set the carry mask to $0xxx...xxx0$ and then use the original fixed point generator.

The significands of the floating-point operands are represented as fixed point masks as follows: M_x represents the significand of operand a, M_y represents the significand of operand b and M_z represents the significand of the intermediate result.

The masks for the significands of the input operands are formed by a leading one followed by bits that are all don't care, i.e., $1xx..xx$. The mask for the significand of the intermediate result is formed by a leading one followed by p don't cares, followed by the mask of the invisible bits, followed by a mask that represents bits that contribute to the calculation of the sticky bit. A zero sticky bit is converted to a mask of zeros, while a sticky bit of one is converted to a mask of don't cares. The number of the latter bits depends on the exponent case. In addition, the exponent case imposes specific alignment values as illustrated in Section 4.3.

There is a slight complication that we need to consider because a mask of don't cares for the bits that contribute to the sticky bit, does not force the fixed point test generator to generate $\sigma = 1$. If the fixed point generator generates all these bits as zero, although the sticky bit is constrained to be one, we repeatedly regenerate the fixed-point masks until a solution is found. In these regenerations, we force the sticky bit to be one. In each try, we choose one of the bits that contribute to the sticky bit and set its value to one while the rest are don't cares. At each iteration, we choose another bit in a random manner. In this way, if a solution exists, we will find it in one of the iterations.

We illustrate the construction of the fixed point masks with an example. The example is for effective addition, where the sticky bit is one. We use $p = 5$ and $q = 12$. The mask of the invisible bits is represented as `IIIIIII`. For the exponent case (addition, $E_a - E_b = 8$, $E_a - E_c = 0$, $\sigma = 1$), the fixed point masks are:

M_x: `1xxxx00000000`
M_y: `+ 000000001xxxx`
M_z: `1xxxxIIIIIIIx`

4.5 Constructing the Floating-Point Solution (Step 5)

The fixed point generator generates significands for the input operands assuming specific alignment values. Using these significands and the definitions of the chosen exponent case, we build two input operands a and b, such that the intermediate result of the operation $a + b$ is compatible with the invisible bits mask. We have to choose the sign, exponent, and significand of each input operand.

The signs of the operands are defined by the exponent case, and the significands are extracted from the fixed point solution.

In choosing the exponents, we need to take into account the restrictions on the exponents of a and of b that stem from the exponent case and from the floating-point number definition. Note that the exponent of c is not restricted by the minimum and maximum floating-point exponents. First we choose a random value for E_a, such that $max(E_{min}, E_{min} + m) \leq E_a \leq min(E_{max}, E_{max} + m)$. Then we restrict the possible values of E_b according to the relation between E_a and E_b, and choose a random value for E_b.

The algorithm is based on applying the fixed point test generator on a random exponent case. If the fixed point generator finds a solution, the algorithm generates input operands that satisfy the constraint. If the fixed point generator states that no solution exists, another random exponent case is chosen until either a solution is found or the whole list of exponent cases was covered. The latter case means that no solution exists for the problem.

In the worst case, the algorithm will check the whole list of exponent cases, which is $O(q)$. Because we assume that $p + 2 \leq q \leq 3p + c$, the number of exponent cases is $O(p)$. Since the complexity of the fixed point generator is $O(p^2)$, the total complexity of the algorithm is $O(p^3)$.

5 Division

5.1 Determining the Value of q

An infinite number of bits can be generated during the calculation of the intermediate result of a divide operation. Nevertheless, we claim that an accuracy of $q = 2p$ bits is 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 (possible infinite) tail of the quotient must all be zero, or can contain bits that are 1 ([4]). The algorithms described in this section are applicable to any value of q. However, when running the experiments described in the Complexity Discussion section, we set $q = 2p$.

5.2 Algorithm

The algorithm is based on the following problem:

Specific significand for division problem: *Given a specific value S_c, for the significand of the 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 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.*

The first stage of the algorithm is to choose random values for the significand of the intermediate result and for the sticky bit, which are compatible with the given masks. Then we try to generate two significands for the input operands using the "specific significand for division" problem. If no solution exists, we select another specific value. This process is repeated until a solution is found or until a preset limit on the number of trials has been reached.

Prior to making the random selection, we reduce the domain of the intermediate result by eliminating cases for which it is known that no solution exists. One such case occurs when there exists a bit set to 1 at location $\geq p$ of the significand, while the sticky bit is 0 (see [10]).

In the following sections, we describe an algorithm that solves the specific significand problem. We divide the discussion into two cases: when the sticky bit is zero and when the sticky bit is one.

5.3 Case I: the Sticky Bit is Equal to Zero

In this case, we must have $S_c = significand\,of(a/b)$ exactly, where $1 \leq S_a < 2$, $1 \leq S_b < 2$, $1 \leq S_c < 2$. Let us convert this into a problem with 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:

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

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

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, $S_a = \alpha a' 2^{1-p}$, $S_b = \alpha b' 2^{1-p}$.

5.4 Case II: the Sticky Bit is Equal to One

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

(i) $(2^{p-1}S_a)/(2^{p-1}S_b) - (2^{q-1}S_c)/(2^{q-1}) = \varepsilon$

(ii) $(2^{p-1}S_a)/(2^{p-1}S_b) - (2^{q-1}S_c)/(2^{q}) = \varepsilon/2$

where in both cases $0 < \varepsilon < 2^{1-q}$. 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.

5.4.1 Known Results Related to Continued Fractions

We formulate some results related to continued fractions we will use. Some of the results are extracted from text books that discuss continued fractions (e.g., [6]), some are derived by the authors, and all are easy to prove. The proofs are omitted:

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.

3. $\alpha_0 < \alpha_2 < \alpha_4 < \ldots < \alpha_n = \bar{p}/\bar{q} < \ldots < \alpha_5 < \alpha_3 < \alpha_1$

4. $\alpha_m = P_m/Q_m$, $m = 0,1,2,\ldots,n$ where $P_{-1} = 1$, $Q_{-1} = 0$, $P_0 = p_0$, $Q_0 = 1$, $P_{m+1} = P_m p_{m+1} + P_{m-1}$, $Q_{m+1} = Q_m p_{m+1} + Q_{m-1}$, $(m = 0,1,\ldots,n-1)$. (This implies that $\{P_m\}$, $\{Q_m\}$, $m = 0,1,\ldots,n$ are strictly increasing sequences, with the exception $Q_1 = Q_0 = 1$ if $p_1 = 1$ and $P_2 = P_1 = 1$ if $p_0 = 0$, $p_2 = 1$.)

5. $0 < [0; p_m, p_{m+1}, \ldots, p_n] < 1$, $m = 1,2,\ldots,n$

6. The continued fraction representation for $\bar{p}/\bar{q}$ can be generated by Euclid's algorithm, defined by the following recursion: $r_{-2} = \bar{p}$, $r_{-1} = \bar{q}$, $r_{m-2} = r_{m-1} p_m + r_m$, $m = 0,1,\ldots,n$ (note that $r_m/r_{m-1} = [0; p_{m+1}, p_{m+2}, \ldots, p_n]$, where r_m, $m = -2,-1,0,1,\ldots,n$ are integers satisfying $0 < r_m < r_{m-1}$, $m = 0,1,\ldots,n-1$ and $r_n = 0 < r_{n-1}$).

7. Since $\{r_m\}$ is a strictly decreasing sequence of non-negative integers, every fraction $\bar{p}/\bar{q}$ has a representation as a finite continued fraction (n is the first value of m for which $r_m = 0$).
8. $Q_m P_{m-1} - P_m Q_{m-1} = (-1)^m$, $m = 0, 1, \ldots, n$ (This implies that P_m, Q_m are prime to each other.)
9. $0 < |P_m/Q_m - \bar{p}/\bar{q}| < 1/(Q_m Q_{m+1})$, $m = 0, 1, \ldots, n-1$ and $|P_n/Q_n - \bar{p}/\bar{q}| = 0$
10. If $\hat{p}/\hat{q} = [p_m; p_{m+1}, \ldots, p_n]$ then $\bar{p}/\bar{q} = (P_{m-1}\hat{p} + P_{m-2}\hat{q})/(Q_{m-1}\hat{p} + Q_{m-2}\hat{q})$, $m = 2, 3, \ldots, n$

5.4.2 Using Continued Fractions to Generate Solutions

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:

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

(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:

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

We propose an algorithm based on continued fractions. We will demonstrate this for case (i); the algorithm for case (ii) is similar. 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 property 10 of continued fractions (see the previous sub-section), we can also write $\bar{p} = P_{i-1}\hat{p} + P_{i-2}\hat{q}$, $\bar{q} = Q_{i-1}\hat{p} + Q_{i-2}\hat{q}$. This allows us to write the following conditions that $\hat{p}$, $\hat{q}$ must satisfy. These conditions must be solved in order to generate a solution. We specify the conditions for the case that i is even (if i is odd, the inequalities of the first condition are reversed):

(I) $\hat{p}'/\hat{q}' < \hat{p}/\hat{q} < \hat{p}''/\hat{q}''$

(II) $2^{p-1} \le P_{i-1}\hat{p} + P_{i-2}\hat{q} < 2^p$

(III) $2^{p-1} \le Q_{i-1}\hat{p} + Q_{i-2}\hat{q} < 2^p$.

We generate a solution for $\hat{p}$, $\hat{q}$, and consequently for $\bar{p}$, $\bar{q}$, as follows:

1. Solve each of the conditions, (I), (II), (III), for $\hat{p}$. We get three new inequalities of the form,

   $F_1(\hat{q}) < \hat{p} < G_1(\hat{q})$, $F_2(\hat{q}) \le \hat{p} < G_2(\hat{q})$, $F_3(\hat{q}) \le \hat{p} < G_3(\hat{q})$

   where $F_j(\hat{q})$, $G_j(\hat{q})$ are linear in $\hat{q}$.
2. By removing $\hat{p}$ from the inequalities, we get nine inequalities that involve only the variable $\hat{q}$. These are of the form $F_j(\hat{q}) < G_k(\hat{q})$. Some of these are satisfied for all $\hat{q}$ and can therefore be removed (e.g., $F_j(\hat{q}) < G_j(\hat{q})$).
3. We solve each of the relations for $\hat{q}$ to get lower and upper bounds on q.
4. If the intersection of the bounds is empty, there is no solution. If the intersection is not empty, choose a random positive integer for $\hat{q}$ in this intersection.
5. The relations in Step 1 now pose constraints on $\hat{p}$. Find the intersection of these three intervals. If the intersection includes no positive integer $\hat{p}$, go back to Step 4. If it does, select a random value for $\hat{p}$.
6. Return $\bar{p} = P_{i-1}\hat{p} + P_{i-2}\hat{q}$, $\bar{q} = Q_{i-1}\hat{p} + Q_{i-2}\hat{q}$ and stop.

We can 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.

5.5 Complexity Discussion

We begin with a high level summary of the stages in the divide algorithm:

1. Choose a random significand S_c and a sticky bit σ_c from the reduced domain of the mask constraint.
2. Try to generate two significands for the input operands, S_a and S_b, whose quotient is S_c with sticky bit σ_c.
3. If Step 2 is successful, generate the input operands. Otherwise, go back to Step 1.

The running time of the algorithm is a function of the number of intermediate results the solver tries until a solution is found. This number depends on two factors: 1) The distribution of the solvable intermediate results within the domain, 2) The probability that FPgen will find a solution for a given solvable intermediate.

Although we have no theoretical calculation of the complexity of the algorithm, we provide experimental evidence of its efficiency. In Section 5.5.1, we describe three experiments that demonstrate the following observations:

- The probability that for a random intermediate result c, chosen from the reduced domain, there exist a and b so that $a/b = c$ is approximately 40%.
- The probability that FPgen will find a solution given a solvable intermediate result is nearly 100%.
- FPgen running time in solving a given intermediate result is much less than one second.

The above observations show that, in practice, the algorithm is very efficient. The algorithm is in daily use by FPgen as the generator for hard to round cases for the divide operation. It solves all the required constraints very efficiently and has found bugs in a divide implementation.

5.5.1 Experimental Results

In the first experiment, we used FPgen to solve the invisible bits problem for divide with no constraint on the invisible bits. In each case, a random intermediate result was chosen. We counted the number of intermediate results for which a solution was found. FPgen tried to solve 400,000 different intermediate results for each precision. For 40% (37% for single precision, 39% for double and 40% for quad) of the results, a solution was found.

Next, we examined all the intermediate results that were not solved in the first experiment. For a given value, c, we checked if there exist a and b such that $a/b = c$. This was done by checking all the values in the range of $\hat{q}$. Since the $\hat{q}$ range was small for all of these intermediate results, we managed to show that these intermediate results are indeed unsolvable, by exhautively checking all values in this range.

We made an additional experiment for double precision: We ran the algorithm on specific intermediate results, c, that are known to be solvable. This was done by first selecting random floating-point numbers a and b, and taking c to be the intermediate result of a/b. We generated around 3.5 million cases. A solution was found for all of them.

6 Summary and Future work

We presented a method for defining interesting verification tasks that target the invisible bits of the intermediate result. This method enables the user to define a mask constraint on the invisible bits for a given operation (op). We described algorithms for solving the invisible bits problem for $op \in \{+, -, \times, \div\}$ to generate random input operands that yield an intermeidate result compatible with the constraint.

Many other instructions are of interest. Among them are reciprocal, square-root, and fused multiply-add ($a \times b + d$). We implemented solutions for multiply-add, based on the algorithm described for the add operation, where one of the operands is the product ($a \times b$) and the second is the addend (d). From the fixed point generator solutions, we can extract values for the product and for the addend. However, in order to generate the multiplication operands (a and b), we have to factorize the product. Several optimizations can be applied to improve this algorithm, but are beyond the scope of this paper. Regarding square-root, a method in which the intermediate result is first randomly selected will not work efficiently, because no solution exists for the majority of intermediate values.

We would also like to verify similar cases for decimal floating-point operations [3]. We will have to rephrase the mask definition to support ten digits instead of two, and of course, we have to define new algorithms that are suitable for the decimal problem.

References

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

[2] M. Aharoni, S. Asaf, 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.

[3] 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.

[4] C. Iordache and D. W. Matula. On Infinitely Precise Rounding for Division, Square Root, Reciprocal and Square Root Reciprocal. In *14th IEEE Symposium on Computer Arithmetic (ARITH14)*, pages 233 – 240, 1999.

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

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

[7] M. Hall Jr. *The Theory of Groups*, 1976. Chelsea, New York.

[8] 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.

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

[10] 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.

[11] A. Ziv, M. Aharoni, and S. Asaf. Solving Range Constraints for Binary Floating-Point Instructions. In *Proc. 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, pages 158 – 164, 2003.

[12] A. Ziv and L. Fournier. Solving the Generalized Mask Constraint for Test Generation of Binary Floating Point Add Operation. *Theoretical Computer Science*, 291:183 – 201, 2003.