Test Generation for the Binary Floating Point Add Operation With Mask-Mask-Mask Constraints

Abraham Ziv
Laurent Fournier
IBM Israel Science and Technology
Matam-Advanced Technology Center, Haifa 31905, ISRAEL
E-mail address: laurent@il.ibm.com, ziv@il.ibm.com

September 29, 2002

Abstract

The mathematical problem discussed is important for generating test cases in order to debug floating point adders designs.

Floating point numbers are assumed to be written as strings of $\{0, 1\}$ bits, in a format compatible with IEEE standard 754. A mask is a string of characters, composed of $\{'0', '1', 'x'\}$. A number and a mask are compatible if they have the same length and each numerical character of the mask ($'0'$ or $'1'$) is equal, numerically, to the bit of the number, in the same position. The problem discussed is: Given masks M_a, M_b, M_c, of identical lengths, generate three floating point numbers $\bar{a}$, $\bar{b}$, $\bar{c}$, which are compatible with the masks and satisfy $\bar{c} = round(\bar{a} \pm \bar{b})$. If there are many solutions, choose one at random.

1 Introduction

IEEE compliance to floating-point hardware in microprocessors has traditionally been a challenging task to achieve. Many escape bugs, including the infamous Pentium bug, belong to the floating-point unit and reveal that the verification process in this area is still far from being optimal. The ever-growing demand for performance and time-to-market causes the verification work to become increasingly harder. So does the low tolerance for bugs on the finished product. There are many sources of problem in the implementation of the floating point unit: they range from data problems on single instructions to the correct handling of sequences of instructions in which back-to-back events challenge superscalar implementations. The roots of the complexity stem both from the interpretation of the specification (architecture) and from the pecularities of the implementation (microarchitecture). Verification has traditionally been targeted through the simulation of test-programs [3,5]. Lately, the area of formal methods has significantly evolved, especially for the floating-point unit verification [9,10,11,13], but is still far from providing a complete answer to the problem. Hence, in most environments, the simulation of test cases is still a major component of the verification process. For this goal, we are developing FPgen, a floating-point random test generator which is expected to provide a quasi-optimal framework for the generation of test cases. The present paper describes a particularly interesting and important algorithm used by FPgen to solve one of its numerous constraints.

It is clear that there is an enormous, practically illimited number of different calculation cases to test. In practice then, simulation can be done on only a very small portion of the existing space. The rationale beyond verification by simulation is that one acquires confidence on design correctness by running a set of test cases exercising a sufficiently large number of different cases, which in some sense are assumed to be a representing sample of the full space. It is inferred that the correct handling of the tested cases is a testimony for the design correctness on all the cases. The difficult question is how to build such a representative set of test cases. Since both the architecture specification and the microarchitecture implementation are yielding a myriad of special cases, pure (uniform) random generation of test cases would be largely inefficient. As a simple example, it is common that a 0 result on an FADD instruction is exercising a very specific part of the design logic (and thus such a case should be verified), while the relative probability to get such a case randomly is extremely low. Therefore, random test generators [3-5] usually possess some internal Testing Knowledge (TK) to bias the test generation towards interesting cases. In effect, this TK is changing the probability distribution of the test space, making it more adapted to the existing knowledge of the space. In the Genesys test-generator [3,5], the TK is in the form of C functions (called generation functions) which can be added incrementally to the tool, even by users themselves. The problem with this approach is that these generation functions are very complex to write, requiring deep FP (Floating Point) understanding. In practice, very few have been added. In contrast, as will be briefly described below, FPgen offers generic TK, requiring no additional effort when new cases need to be checked.

How does one know that a certain set of tests is sufficient? This question is related to the notion of coverage, i.e. to the comprehensiveness of the set related to the verification target [2,6-8]. Usually, coverage models are defined and the set of tests should fulfill all the existing tasks. A coverage model is a set of related cases. For example, a common - albeit far from trivial to fulfill - coverage model is one which requires to enumerate on all major IEEE Floating Point types simultaneously for all operands of all FP instructions. For a given instruction with three operands, say ADD, this potentially yields a thousand (10^3) of cases to cover, assuming 10 major FP types ($\pm$NaNs, $\pm$Infinity, $\pm$Zero,

±Denormal, ±Normal). This model can be further refined by adding more FP types, such as Minimum and Maximum denormals, etc... Obviously, not all cases are possible (for example, the addition of 2 positive denormal numbers cannot reach infinity), so that the actual number of cases is in fact lower than the size of the cartesian product. A coverage model, or the set of all coverage models, is really an attempt to partition the set of all the calculation cases in such a way that the probability distribution should be similar for all subsets. Again, as will be explained below, FPgen offers what is called Coverage by generation, or in other words, it takes as an input the request of a coverage model, and outputs the set of tests covering it.

The following paragraph offers a high level description of FPgen. More detailed information appears in a dedicated paper to be published in the near future. FPgen is an automatic test generator which gets as input the description of a coverage model, and outputs a set of tests covering the model. A coverage model will be defined by specifying a set of different constraints to be fulfilled, each constraint corresponding to a particular task targeted by the coverage model. More precisely, a coverage model will have the form of a sequence of FP instructions, with sets of constraints on the input, intermediate result(s) and result operands of the participating instructions. Covering the model translates then to provide a set of tests which, on one hand, display the instruction sequence, and on the other hand possess the following property: each constraint is satisfied by at least one test of the set. In effect, FPgen will generate exactly one test for each constraint. The general outlook of a single instruction constraint is of the following form:

FPinst (Op1 in Pattern1) (Op2 in Pattern2) (IntRes in Pattern3) (Res in Pattern4)

where FPinst is a generic floating point instruction with 2 input operands (Op1 and Op2), 1 intermediate result (IntRes), and a result (Res). The case of 2 operands and a single intermediate result is used here for simplicity of notation, but of course generalization to any number of such parameters is immediate. A Pattern is a construct representing a logical OR among sets of FP numbers. The sets serve as constraints defining (in fact limiting) the allowable FP numbers for each term of the outlook. Pattern have the general following form:

Pattern = Set1 OR Set2 OR ... OR SetN

where each Set is a set of FP numbers. Each task of the coverage model corresponds to a specific selection of Set for each Pattern. Covering the task reduces then to select a data tuple where each individual data belongs to the corresponding selected Set. Thus, the number of different tasks engendered by such a single instruction is the multiplication of the number of Sets for each participating Pattern. The number of tasks for a sequence is obtained by multiplying the number of tasks of each individual instruction.

The different ways to define sets of FP numbers serve to conveniently translate typical constraints emanating from verification plan's tasks. They constitute therefore a cornerstone of the tool.

- Ranges and masks. Separate range constraints should be possible on the Exponent and Mantissa. A mask is represented by a FP number where some bits are X (don't cares) while the others are regular 0's and 1's.
- The ability to specify the number of bits equal to 1 within any given field of the FP number. Exact, MIN and MAX should be given (for example: at least 1 bit set in bits 61-63).
- Ability to specify length of continuous stream of 1's or 0's. As before, Exact, MIN and MAX should be given for any field without overlap between fields, and without crossing the mantissa-exponent border. For example, a number with a continuous stream of at least 45 1's in its mantissa.
- Specifying a Set for which the selected value should be a function of the value selected for another operand (usage of symbol). As a start, $+$ and $-$ are sufficient. These operations have to be understood as distance in term of representable numbers. The symbols must be enabled on any field of the number. (Example: exponent at a distance of at most 2 from the exponent selected for previous operand).
- Sets operations (Intersection, union, complement, of same and different Set types). No Set operation can be done for Sets defined with symbols. For practical reasons, there will be a limitation to the number of Set operations per constraint.

The ultimate goal of FPgen is to be floating-point generic, and thus applicable to any floating point architecture. Reasonably, its first goal will be to support the IEEE standard arithmetic and to limit its genericity to include therein all the allowed FP format sizes (i.e., 32, 64, 80, 128).

In general, any architecture resource which might influence FP instruction's results should be controllable. For IEEE standard architecture, this bounds to Rounding Modes and Enabled Flags, and they will therefore be settable (0,1,X) from FPgen.

The central engine of FPgen solves constraints emanating from set restrictions on instruction operands. Given a restriction, FPgen is tuned to seek for a random instance solving it, one which is uniformly distributed among the set of all solutions. This is the theoretic goal, but the complexity involved is sometimes overwhelming, especially for complex or multiple restrictions. In such cases, FPgen at least ensures that each solution has a reasonable probability to be selected. As described above, constraints can be given on the input operands, output operands or even on both types simultaneously. It should be clear that there is a significant leap in complexity involved in solving constraints on output operands. Indeed, in contrast to the input operands case, the constraint on output operands includes the instruction semantics. However, even output constraints are usually solvable analytically in reasonable time complexity.

Constraint restrictions start to become largely intractable when simultaneous constraints are requested on both input and output operands. For example, it is largely unclear how to find an instance in which the result of a MUL instruction has at least 45 bits set in its mantissa and the inputs are restricted by specific ranges or masks. Such a case might seem artificial, but it is often the case that cases such as this one are important to check due to specific implementation methods. Moreover, during the implementation itself, it is sometimes important to explore whether some cases are possible at all (FPgen also informs when no solution exists). Knowing that some cases can be neglected can be critical to be able to optimize the microarchitecture. In fact, in many cases, it can be shown that the constraint problem is NP-Hard. Thus, FPgen's approach for these problems is heuristic, mixing probabilistic, search space and semi-analytic algorithms. However, some important cases of simultaneous constraints are solvable analytically. For example, Range constraints or Mask constraints on all operands for the FP ADD instruction. In this paper, we will present the algorithm for the simultaneous Mask constraints, i.e., a Mask constraint on both input operand and on the output one for FADD. The algorithm for solving simultaneous Range constraints for FP ADD, which is - somewhat unexpectedly - far from trivial, will be the subject of an additional paper.

Masks are important means to define sets of FP numbers. First, at the architecture level, they enable to define all the IEEE generic types of FP: Normals, Denormals, Infinities, Zeroes, NaNs, etc... Thus, the "all types" coverage model defined above is expressible through masks. Second, it is common that implemetations treat some bit, or set of bits, in a particular manner, and masks are the natural means to bias towards numbers where those are controlled while the others are random. Moreover, the importance of the algorithm described in this paper goes beyond the specific importance of the mask construct for set restrictions. Indeed, as it will become apparent during the explanation of the algorithm below, the genericness of the solution allows one to control (via a mask) the stream of carry bits occurring during the addition. Exercising an adder through different carry configurations usually constitutes an important part of its verification.

In section 2 the main problem, which is the subject of this article is defined. Section 3 outlines an algorithm, that solves the main problem. The discussion includes definitions of several, auxilliary generators and description of their use, by the main generators. Sections 4-7 analyze and describe algorithms for the various auxilliary generators defined in section 3. In section 8 the issue of complexity of the algorithm is discussed shortly.

2 Description of the problem

Before going into details we present a simple example which demonstrates the problem to a reader familliar with the IEEE standard 754. Consider a hypothetical binary floating point format of eight bits, whose structure is, $seeeffff$. Namely, it includes one bit for a sign, three bits for a biased exponent and four bits for a fraction. In analogy with the IEEE formats *single* and *double*, its significand has five bits, $E_{min} = -2$, $E_{max} = bias = 3$. Given three masks $M_a = 0100x101$, $M_b = 001x1011$, $M_c = 010xx10x$ we look for three floating point numbers $\bar{a}$, $\bar{b}$, $\bar{c}$, compatible with the masks respectively, such that $\bar{c} = round(\bar{a} + \bar{b})$. Assuming that *round* stands for round *to nearest/even*, the following is a possible solution, $\bar{a} = 01000101$, $\bar{b} = 00101011$, $\bar{c} = 01001100$. We proceed with more details.

2.1 The set of machine numbers

Our problem is considered here in the framework of IEEE standard 754 computer arithmetic (see [1] and [14]). This is a binary floating point system. We assume that three integral constants are given, E_{min}, E_{max}, p. The machine numbers are those representable in the form $v = (-1)^s \times 2^E \times b_0.b_1b_2 \cdots b_{p-1}$, where $s \in \{0,1\}$ represents the sign of v, E, the exponent of v, is an integer satisfying $E_{min} \leq E \leq E_{max}$, $b_i \in \{0,1\}$ and p is the precision of the floating point system. All machine numbers, v, that satisfy $|v| \geq 2^{E_{min}}$ are assumed to be normalized (i.e. with $b_0 = 1$). Those machine numbers which are smaller, in magnitude, than $2^{E_{min}}$ (including zero) have $E = E_{min}$ and are denormalized (i.e. have $b_0 = 0$). Thus, each machine number has a unique representation (Note that the IEEE standard 754 requires the same uniqueness for its single and double formats but not for its extended formats).

2.2 Binary representations of machine numbers and the mask constraint

We assume throughout that numbers are represented as strings of binary bits. This is true for fixed point numbers as well as for floating point numbers. A mask, related to a number, is assumed to be a string of characters, of the same length as the number, all of whose characters are in the set $\{0, 1, x\}$. A number and a mask are compatible if they are of the same length (hence each bit of the number corresponds to one, particular, character of the mask), each $'1'$ character of the mask corresponds to a $'1'$ character of the number and each $'0'$ character of the mask corresponds to a $'0'$ character of the number. Thus, a $'1'$ or a $'0'$ character of the mask determines uniquely the corresponding character of the number. An $'x'$ character in the mask leaves the corresponding character of the number undetermined.

We may constrain a number by assuming that it is compatible with a given mask.

For our purposes it is not convenient to represent a machine number by a single string of bits. It is preferable to split such a representation into three strings:

Sign: A string of one bit, which is “0” for a $'+'$ and “1” for a $'-'$. We denote its numerical value by s ($s = 0$ or 1).

Biased exponent: A string of w bits. Interpreting it to be a binary integer we denote its numerical value by e ($0 \leq e \leq 2^w - 1$). Generalizing from the single and double formats of the IEEE standard 754 we take $E_{min} = 2 - 2^{w-1}$, $E_{max} = bias = 2^{w-1} - 1$.

Significand: A string of p bits, $b_0 b_1 b_2 \cdots b_{p-1}$. Unlike the single and double formats of IEEE standard 754 we include b_0 explicitely in the string. Interpreting the string as a binary number, with the binary point placed between b_0 and b_1, we get the numerical value of the significand, S ($0 \leq S \leq 2 - 2^{1-p}$).

The value, v, which corresponds to such a triplet of bit strings is given by

1. If $e = 2^w - 1$ and $S \neq 1$, then v is NaN regardless of s.
2. If $e = 2^w - 1$ and $S = 1$, then $v = (-1)^s \times \infty$ (Infinity).
3. If $0 < e < 2^w - 1$ and $S \geq 1$, then $v = (-1)^s \times 2^{e-bias} \times S$ (Normalized numbers).
4. If $e = 0$ and $S < 1$ then $v = (-1)^s \times 2^{E_{min}} \times S$ (Denormalized numbers and zeroes).

2.3 The mask-mask-mask test generation problem for floating point numbers

The test generation problem which interests us is the following: Given masks for three machine numbers, generate machine numbers, $\bar{a}$, $\bar{b}$, $\bar{c}$, which are compatible with the masks and satisfy $\bar{c} = round(\bar{a} \pm \bar{b})$. This problem may be split into two sub-problems, which may be solved by two generators of machine numbers, respectively:

Floating point generator for addition *Given six masks, for biased exponents and for significands, M_{ea}, M_{Sa}, M_{eb}, M_{Sb}, M_{ec}, M_{Sc}, the generator either generates three non-negative machine numbers, $\bar{a}$, $\bar{b}$, $\bar{c}$, whose biased exponents and whose significands are compatible with the corresponding masks and satisfy, $\bar{c} = round(\bar{a} + \bar{b})$, or states that there is no solution.*

Floating point generator for subtraction *Given six masks, for biased exponents and for significands, M_{ea}, M_{Sa}, M_{eb}, M_{Sb}, M_{ec}, M_{Sc}, the generator either generates three non-negative machine numbers, $\bar{a}$, $\bar{b}$, $\bar{c}$, whose biased exponents and whose significands are compatible with the corresponding masks and satisfy, $\bar{c} = round(\bar{a} - \bar{b})$, or states that there is no solution.*

We are interested in these problems for

$$round \in \{down, up, toward\ zero, to\ nearest/even\}.$$

Actually, since we assumed that the three machine numbers are positive, we may omit, with no loss of generality, the round *toward zero* mode.

In case there exist more than one solution, a generator chooses one of them, at random. Multiple invokations of the generator, are supposed to make independent random choices. So, repetitions of solutions might occur, although they are rare, in case there exists a large number of possible solutions. A generator is expected to be fair in the sense that no existing solution to the problem it faces is exluded by its method of operation.

3 Outlines of the method of solution

As was stated earlier, the problem of generating floating point numbers, satisfying $\bar{c} = round(\bar{a} \pm \bar{b})$, may be divided into two cases. Addition of non-negative numbers and subtraction of non-negative numbers.

Let us consider first the addition case. Namely, let $\bar{c} = round(\bar{a} + \bar{b})$, where $\bar{a}$, $\bar{b}$, $\bar{c}$ are non-negative machine numbers. We denote $q_a = e_c - e_a$, $q_b = e_c - e_b$, where e_a, e_b, e_c are the biased exponents. It is not difficult to see that q_a, q_b are non-negative integers, one of which is 0 or 1. Let us denote also $Q_a = E_c - E_a$, $Q_b = E_c - E_b$, where E_a, E_b, E_c are the unbiased exponents. It is easy to see that Q_a, Q_b are also non-negative integers, one of which is 0 or 1. Usually $Q_a = q_a$, $Q_b = q_b$ but this is not always so. Actually there are five cases:

1. $e_a > 0$, $e_b > 0$, $e_c > 0$: $Q_a = q_a$, $Q_b = q_b$
2. $e_a = 0$, $e_b > 0$, $e_c > 0$: $Q_a = q_a - 1$, $Q_b = q_b$
3. $e_a > 0$, $e_b = 0$, $e_c > 0$: $Q_a = q_a$, $Q_b = q_b - 1$
4. $e_a = 0$, $e_b = 0$, $e_c > 0$: $Q_a = q_a - 1$, $Q_b = q_b - 1$
5. $e_a = 0$, $e_b = 0$, $e_c = 0$: $Q_a = q_a$, $Q_b = q_b$

A similar analysis may be performed for the subtraction case. The outcome is very similar: With $\bar{c} = round(\bar{a} - \bar{b})$ we have $q_b = e_a - e_b$, $q_c = e_a - e_c$, $Q_b = E_a - E_b$, $Q_c = E_a - E_c$, one of q_b, q_c is 0 or 1, one of Q_b, Q_c is 0 or 1. Also,

1. $e_c > 0$, $e_b > 0$, $e_a > 0$: $Q_c = q_c$, $Q_b = q_b$
2. $e_c = 0$, $e_b > 0$, $e_a > 0$: $Q_c = q_c - 1$, $Q_b = q_b$
3. $e_c > 0$, $e_b = 0$, $e_a > 0$: $Q_c = q_c$, $Q_b = q_b - 1$
4. $e_c = 0$, $e_b = 0$, $e_a > 0$: $Q_c = q_c - 1$, $Q_b = q_b - 1$
5. $e_c = 0$, $e_b = 0$, $e_a = 0$: $Q_c = q_c$, $Q_b = q_b$

q_a	0	0	1	1	0	0	$\cdots$	0	0	1	1	$\cdots$	1	1	2	3	$\cdots$	q	$> q$	2	3	$\cdots$	q	$> q$
q_b	0	1	0	1	2	3	$\cdots$	q	$> q$	2	3	$\cdots$	q	$> q$	0	0	$\cdots$	0	0	1	1	$\cdots$	1	1

Table 1: Pairs q_a, q_b

3.1 Structure of the algorithm

The idea is to start the generation of machine numbers by choosing values for q_a, q_b, Q_a, Q_b (or q_b, q_c, Q_b, Q_c, in the case of subtraction). Having known values for these numbers we may produce the biased exponents and the significands, independently, by invoking numbers generators such as those defined below (outlines of algorithms for such generators are described later).

Biased exponents generator I: *Given the two non-negative integers q_1, q_2, with $q_1 \in \{0, 1\}$, and three masks of length w, M_1, M_2, M_3, for the biased exponents, the biased exponents generator I either generates three biased exponents e_1, e_2, e_3, which are compatible with the masks, respectively, and satisfy $e_3 = e_1 + q_1 = e_2 + q_2$, or states that no solution exists.*

Biased exponents generator II: *Given the two non-negative integers q_1, q, with $q_1 \in \{0, 1\}$, and three masks of length w, M_1, M_2, M_3, for the biased exponents, the biased exponents generator II either generates an integer q_2 and three biased exponents e_1, e_2, e_3, which are compatible with the masks, respectively, and satisfy $e_3 = e_1 + q_1 = e_2 + q_2$, $q_2 > q$, or states that no solution exists.*

Significands generator for addition: *Given two non-negative integers, Q_a, Q_b, one of which is 0 or 1, and three masks, of length p, for the significands, M_{Sa}, M_{Sb}, M_{Sc}, and a rounding mode, $round \in \{down, up, to\ nearest/even\}$, the significands generator either generates three significands S_a, S_b, S_c, which are compatible with the masks, respectively, and satisfy $S_c = round(2^{-Q_a} S_a + 2^{-Q_b} S_b)$, or states that no solution exists.*

Significands generator for subtraction: *Given two non-negative integers, Q_b, Q_c, one of which is 0 or 1, and three masks, of length p, for the significands, M_{Sa}, M_{Sb}, M_{Sc}, and a rounding mode, $round \in \{down, up, to\ nearest/even\}$, the significands generator either generates three significands S_a, S_b, S_c, which are compatible with the masks, respectively, and satisfy $2^{-Q_c} S_c = round(S_a - 2^{-Q_b} S_b)$, or states that no solution exists.*

The pair q_a, q_b, in the addition case, and the pair q_b, q_c, in the subtraction case, are each, one of a list of possible pairs, similar to the list given in Table 1 (we do not specify the exact value of q in the table. A value close to p is convenient). Such a list includes about $2p$ pairs. An outline of an algorithm, for the addition (or subtraction) case would be:

1. Choose one pair q_a, q_b (or q_b, q_c) out of the list, at random. If, however,

the list is empty, state that there is no solution and stop.

2. Use the exponents masks and one of the biased exponents generators to produce e_a, e_b, e_c.

3. If the exponents generator states that there is no solution, erase the pair q_a, q_b (or q_b, q_c) from the list and go back to step 1. Else, compute Q_a, Q_b (or Q_b, Q_c) and use the significands masks and the appropriate significands generator to produce S_a, S_b, S_c.

4. If the significands generator states that no solution exists, erase the pair q_a, q_b (or q_b, q_c) from the list and go back to step 1. Else, return e_a, e_b, e_c, S_a, S_b, S_c and stop.

Note: This version of the algorithm does not account properly for the fact that the values of Q_a, Q_b (or Q_b, Q_c) are not uniquely determined by q_a, q_b (or q_b, q_c). The reason is clarity of presentation. This can be fixed easily.

3.2 The need for a fixed point generator and its definition

In order to understand the general idea behind the significands generator, note that, since Q_a, Q_b (or Q_b, Q_c) are known to this generator, it is possible to shift the significands until they are properly aligned and then add (or subtract) them the way fixed point numbers are added (or subtracted). We need, then, to have a fixed point generator.

Before we specify the exact function of this generator let us examine the process of adding two positive, binary integers, $x + y = z$: We start by adding the rightmost bits of x and y. If the sum is less than 2 then it is equal to the rightmost bit of z and there is no carry. If not, there is a carry of 1. Next we add the carry and the two following bits of x, y. If the sum is less then 2 we have no carry. If not, we have a carry of 1. And so on. Thus, during the process, a sequence of carries, each of which is either 0 or 1, is generated.

If the values of the bits of the summands' are (left to right) $x_m = i$, $y_m = j$ ($m = 0, 1, \cdots, N-1$) and those of the sum are $z_m = k$, then we have the relations: $i + j + C_{m+1} = k + 2C_m$, ($m = 0, 1, \cdots N-1$) where C_m is the carry sequence. We always have $i, j, k \in \{0, 1\}$. Usually we have also $C_m \in \{0, 1\}$. However, a rounding up process may add an additional 1 to the carry and produce an effective carry of 2 (see subsections 5.1, 5.2 below). For this reason it is convenient to assume that $C_m \in \{0, 1, 2\}$. Note that C_0 and C_N are boundary values which, usually, have both the value zero. However, we need a slightly more general generator, so we allow $C_0 \neq 0$, $C_N \neq 0$

The input to the fixed point generator includes masks of length N: M_x, M_y, M_z, of the form described earlier, for the numbers x, y, z. It includes also a mask, M_C, of length $N + 1$, which corresponds to the sequence of carries. This last mask is composed of the characters $'0'$, $'1'$, $'2'$, $'x'$, where $'0'$, $'1'$, $'2'$ determine

MCN	0	1	2	3	4	5	6	7
i	n	n	n	n	x	x	x	x
j	n	n	x	x	n	n	x	x
k	n	x	n	x	n	x	n	x

Table 2: Mask Combination Numbers

the corresponding values of the carry and an $'x'$ leaves the corresponding carry undetermined. We may now state the function of the fixed point generator as follows:

Fixed point generator: *Given three masks of length N, for binary numbers, 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 generates three binary numbers x, y, z and a carry sequence which are compatible with their masks, respectively, or states that no solution exists.*

4 Analysis of the fixed point generator problem

4.1 Mask Combination Numbers and Case Numbers

The basic relations which control the construction of the sequences $x_m = i$, $y_m = j$, $z_m = k$, $(m = 0, 1, \cdots, N-1)$ and C_m, $(m = 0, 1, \cdots, N)$ are the condition of compatibility with the masks and the relations, $i + j + C_{m+1} = k + 2C_m$, $(m = 0, 1, \cdots N-1)$ where $i, j, k \in \{0, 1\}$ and $C_m, C_{m+1} \in \{0, 1, 2\}$. Clearly this set of conditions might be self contradictory. Such contradictions should be identified by the generator which should state, if they exist, that there is no solution.

Given an index m, each of the bits i, j, k corresponds to a character in the appropriate mask. This character may be either an $'x'$ or a number ($'0'$ or $'1'$). With such a classification of the characters of the mask each triplet of masks elements is one of eight possible types of triplets (e.g. all three of the characters are $'x'$ or i corresponds to a number and j, k both correspond to an $'x'$ etc.). Each of the eight types of triplets may be assigned a number, which we denote by MCN (Mask Combination Number). The numbers are given in Table 2. In this table n means a number character in the mask and x means an $'x'$ character in the mask.

The main idea in solving the fixed point generator problem is to construct first the sequence C_m and only later construct the bits of x, y, z.

Given the masks M_x, M_y, M_z and a numerical value for the index, m, we have an MCN value and the numerical values of some of the variables i, j, k. In view of this fact, for MCN=0, 1, 2, 3, 4, 5, 6 we have sufficient information to compute, respectively, a *Case Number* CN=$i+j-k$, $i+j$, $i-k$, i, $j-k$, j, $-k$.

Let us list the pairs (C_m, C_{m+1}) which are possible for each MCN, CN pair:

- MCN=0, CN=$i + j - k$

  CN=-1: (0,1)
  CN= 0: (0,0),(1,2)
  CN= 1: (1,1)
  CN= 2: (1,0),(2,2)

- MCN=1, CN=$i + j$

  CN= 0: (0,0),(0,1),(1,2)
  CN= 1: (0,0),(1,1),(1,2)
  CN= 2: (1,0),(1,1),(2,2)

- MCN=2, CN=$i - k$ or MCN=4, CN=$j - k$

  CN=-1: (0,0),(0,1),(1,2)
  CN= 0: (0,0),(1,1),(1,2)
  CN= 1: (1,0),(1,1),(2,2)

- MCN=3, CN=i or MCN=5, CN=j

  CN= 0: (0,0),(0,1),(1,1),(1,2)
  CN= 1: (0,0),(1,0),(1,1),(1,2),(2,2)

- MCN=6, CN=$-k$

  CN=-1: (0,0),(0,1),(1,1),(1,2)
  CN= 0: (0,0),(1,0),(1,1),(1,2),(2,2)

- MCN=7, CN=0

  CN= 0: (0,0),(1,0),(0,1),(1,1),(1,2),(2,2)

This list is exhaustive in the sense that for each pair of MCN, CN values it includes all of the possible pairs (C_m, C_{m+1}). It is a basis for the construction of a feasible sequence C_m, $(m = 0, 1, \cdots, N)$. By *feasible* we mean such a sequence, C_m, compatible with M_C, for which there exists at least one triplet of corresponding numbers x, y, z, compatible with M_x, M_y, M_z, respectively. Since the list is exhaustive it will enable us to construct potentially every feasible sequence C_m which then will be used to construct potentially every solving triplet x, y, z.

4.2 Preliminary list of n-values

It is possible to draw some useful conclusions from the list. First we note that if, for some index $m \in \{0, \cdots, N-1\}$, we have $C_m = 2$ then it is necessary that $C_{m+1} = 2$ too, and if we have $C_{m+1} = 2$ then it is necessary that $C_m \neq 0$ (i.e. $C_m \in \{1, 2\}$). This implies that one of the following is true:

1. There exists a boundary index $n \in \{1, \cdots, N\}$ such that $C_m = 2$ for all $m \geq n$, $C_m = 1$ for $m = n-1$ and $C_m \in \{0, 1\}$ for all $m < n$.
2. All of the carries are 2 (we set $n = 0$ in this case).
3. All of the carries are in $\{0, 1\}$ (we set $n = N+1$ in this case).

A feasible $n \in \{0, 1, \cdots, N+1\}$ is generally not unique and there might exist several possible values for it. We would like to construct a *list of n-values* which includes all of the values of n that correspond to solutions and no other values of n. Clearly, for all $n \leq m < N$ we must have $C_m = C_{m+1} = 2$. So, looking in the list of carry pairs we find that for all such m the pair (MCN,CN) must be one of: $(0, 2)$, $(1, 2)$, $(2, 1)$, $(3, 1)$, $(4, 1)$, $(5, 1)$, $(6, 0)$, $(7, 0)$. Since $C_{n-1} = 1$, $C_n = 2$ for $n \in \{1, \cdots, N\}$ we infer, from the list, for such n, that it is necessary for $m = n-1$ that the pair (MCN,CN) is one of: $(0, 0)$, $(1, \neq 2)$, $(2, \neq 1)$, $(3, x)$, $(4, \neq 1)$, $(5, x)$, $(6, x)$, $(7, x)$, where $\neq 1$ means CN$\neq 1$, $\neq 2$ means CN$\neq 2$ and x means that CN might have any value. Additional restrictions on n are imposed by the mask M_C. It is necessary that C_{n-1}, C_n, ..., C_N are all compatible with this mask.

Given the masks, this discussion enables one to construct a preliminary list of possible values of n. As we shall see below, this list is often too large and some of its terms must be erased.

4.3 Given n, look for contradictions and construct a feasible sequence of carries

Let us discuss now the completion of the sequence C_m, given a value for n, by setting values to C_0, C_1, ..., C_{n-2}. These missing values of carries must all be in $\{0, 1\}$. Hence, we may take the above list of pairs, (C_m, C_{m+1}), and erase from it all of the pairs which include 2. The remaining list, which is relevant to the construction of the missing carries, may be replaced by the following equivalent list of inference rules:

- MCN=0, CN=$i + j - k$

  CN=-1: $C_m = 0$, $C_{m+1} = 1$

  CN= 0: $C_m = C_{m+1} = 0$

  CN= 1: $C_m = C_{m+1} = 1$

CN= 2: $C_m = 1$, $C_{m+1} = 0$

- MCN=1, CN=$i + j$

  CN= 0: $C_m = 0$

  CN= 1: $C_m = C_{m+1}$

  CN= 2: $C_m = 1$

- MCN=2, CN=$i - k$ or MCN=4, CN=$j - k$

  CN=-1: $C_m = 0$

  CN= 0: $C_m = C_{m+1}$

  CN= 1: $C_m = 1$

- MCN=3, CN=i or MCN=5, CN=j

  CN= 0: $C_m = 1 \Rightarrow C_{m+1} = 1$
  $C_{m+1} = 0 \Rightarrow C_m = 0$

  CN= 1: $C_m = 0 \Rightarrow C_{m+1} = 0$
  $C_{m+1} = 1 \Rightarrow C_m = 1$

- MCN=6, CN=$-k$

  CN=-1: $C_m = 1 \Rightarrow C_{m+1} = 1$
  $C_{m+1} = 0 \Rightarrow C_m = 0$

  CN= 0: $C_m = 0 \Rightarrow C_{m+1} = 0$
  $C_{m+1} = 1 \Rightarrow C_m = 1$

- MCN=7, CN=0

  CN= 0: No restrictions

Like the list of pairs from which it was derived, this set of rules is exhaustive in the sense that each feasible sequence $C_0, \cdots, C_{n-1}$, of $\{0, 1\}$ terms, must be compatible with it and each such sequence, which is compatible with it and with M_C, is feasible.

Let us discuss now the problem of setting values to the carries C_0, ..., C_{n-1}. Their values are constrained by the mask M_C and by the inference rules, listed above. In addition we know that $C_{n-1} = 1$ if $n \in \{1, 2, \cdots, N\}$. The mask M_C uniquely defines those terms of C_m which correspond to non $'x'$ characters (Note, though, that a $'2'$ character in M_C is permitted only for $m \geq n$. Otherwise n should be erased from the list of n-values). The set of inference rules may be divided into three (not disjoint) groups:

1. Assignment rules: E.g. $C_m = C_{m+1} = 0$ or $C_m = 1$ etc.

2. Right continuation rules: E.g. $C_m = 1 \Rightarrow C_{m+1} = 1$ or $C_m = C_{m+1}$ etc.

3. Left continuation rules: E.g. $C_{m+1} = 0 \Rightarrow C_m = 0$ or $C_m = C_{m+1}$ etc.

Applying first only the assignment rules we may assign values to some of the carry terms. One should be aware of the fact that we described here several ways to deduce a definite value for a C_m (mask, assignment rules, $C_{n-1} = 1$). It may happen that we run into some contradictions. So, each time we deduce a definite value for a given C_m we must check whether it was assigned a different value earlier. Clearly, a contradiction means that n should be erased from the list of n-values.

Suppose that all of the methods, described above, to deduce a definite value for a C_m, were used and no contradiction was found. Some of the defined carries may be neighbors (like C_m, C_{m+1}). For each such pair of neighbors we must find the MCN and CN, which correspond to the index m, and test for a contradiction by the corresponding inference rule. Finding a contradiction means that we must erase n from the n-values list. If all of the pairs of neighbors were tested and no contradiction was found we apply the continuation rules, one at a time. This process will create chains of consecutive defined carries, separated by chains of consecutive (yet) undefined carries. As the process continues the chains of undefined carries shrink and it might happen that one of them disappears completely. Namely, the right end of one chain of defined carries becomes a neighbor of the left end of the following chain of defined carries. Such neighbors must be tested for contradiction by the inference rules. If any contradiction is found then n should be erased from the list of n-values. If the process ends and cannot be continued any further and no contradiction was found, then either all of the carries are defined and we have a complete, feasible, sequence of carries, or some chains of undefined carries were left over. In the last case we shall see, that a point was reached, where no more contradictions are expected. Actually we may choose one end of an undefined carries chain and choose for it a value of either 0 or 1, at random. No contradiction can arise from this operation, because, as was mentioned above, the set of inference rules is exhaustive. The new carry becomes a left or a right end of a chain of defined carries. We try to apply continouation rules to this new end, again and again, until the end of the chain meets an end of another chain or until no further continuation rule can be applied, and then we choose again an undefined carry at random. This process is repeated until all of the carries are assigned definite values. Note that if the new end meets another chain of defined carries, namely if the new end becomes a neighbor of another end there cannot arise a contradiction because the other end could not be continued at an earlier stage and this means that its new neighbor may have the value 1 or the value 0 without causing any contradiction.

As was mentioned above, if any contradiction was found then the value of n must be erased from the list of n-values. If we get contradictions for all values of

n, i.e. if, at the end, the list of n-values is empty then the fixed point generator must state that there exists no solution and stop.

This process enables one to discover whether no feasible sequence of carries exists and to produce, in potential, every feasible sequence of carries, otherwise. If we have a feasible sequence of carries we can use it to construct every triplet of solving numbers x, y, z, as described below.

4.4 Construction of the numbers, given a feasible carry sequence

We assume that the whole sequence C_m is known and that it is feasible. For each value of $m \in \{0, 1, \cdots, N-1\}$ we have, then, numerical values for C_m, C_{m+1}, MCN, CN and some of the variables i, j, k and we may write the equation $i+j+C_{m+1} = k+2C_m$. Transfering to the right hand side, of this equation, all of the known variables we get an equation of the type $\alpha =$RHS where the right hand side, RHS, is of a known numerical value and α is an expression which depends on those of the variables i, j, k which are unknown. Actually, it is not difficult to see that RHS$= 2C_m - C_{m+1}-$CN and that α depends on MCN and satisfies, $\alpha + CN = i + j - k$. For instance, if MCN=3 then CN$= i$, $\alpha = j - k$ and the equation is $j - k =$RHS. If MCN=2 then CN$= i - k$, $\alpha = j$ and the equation is $j =$RHS and so on. We can summarise the equations and all of their solutions, in all of the possible cases, in the following list:

- MCN=0, α is an empty expression

  RHS= 0: i, j, k are all known

- MCN=1, $\alpha = -k$

  RHS=-1 or 0: $k = -$RHS

- MCN=2, $\alpha = j$

  RHS= 0 or 1: $j =$RHS

- MCN=3, $\alpha = j - k$

  RHS=-1: $(j, k) = (0, 1)$
  RHS= 0: $(j, k) = (0, 0), (1, 1)$
  RHS= 1: $(j, k) = (1, 0)$

- MCN=4, $\alpha = i$

  RHS= 0 or 1: $i =$RHS

- MCN=5, $\alpha = i - k$

RHS=-1: $(i,k) = (0,1)$
RHS= 0: $(i,k) = (0,0),(1,1)$
RHS= 1: $(i,k) = (1,0)$

- MCN=6, $\alpha = i + j$

  RHS= 0: $(i,j) = (0,0)$
  RHS= 1: $(i,j) = (0,1),(1,0)$
  RHS= 2: $(i,j) = (1,1)$

- MCN=7, $\alpha = i + j - k$

  RHS=-1: $(i,j,k) = (0,0,1)$
  RHS= 0: $(i,j,k) = (0,0,0),(0,1,1),(1,0,1)$
  RHS= 1: $(i,j,k) = (0,1,0),(1,0,0),(1,1,1)$
  RHS= 2: $(i,j,k) = (1,1,0)$

Knowing the numerical values of MCN and RHS for every $m \in \{0, 1, \cdots, N-1\}$, we may choose from the last list, a solution which completes the triplet i, j, k, for every m. If the list includes several solutions for some combination of MCN, RHS, then we choose one of them at random. Making such choices for all values $m \in \{0, 1, \cdots, N-1\}$ we complete the construction of x, y, z.

4.5 Structure of the algorithm of the fixed point generator

Based on the discussion above, we outline the structure of an algorithm that realizes the fixed point generator:

1. Construct a preliminary list of possible n-values.

2. Choose a value of n out of the list of n-values, at random. If the list is empty state that there is no solution and stop.

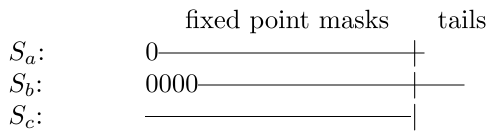
3. Try to construct the missing terms of C_m. In case you find at least one contradiction erase n from the list of n-values and go back to step 2. If no contradiction was found you have a feasible sequence C_m.

4. Using the sequence C_m constructed, and the masks M_x, M_y, M_z, set values for i, j, k for each $m \in \{0, 1, \cdots, N-1\}$. Whenever there is more than one possibility to choose i, j, k make a random choice. When the construction is completed return the resulting x, y, z, C_m and stop.

5 Analysis of the significands generator for addition

5.1 The general idea

The present generator applies the fixed point generator always with $N = p$. The idea goes as follows: Almost always the exponent of $\bar{a} + \bar{b}$ is equal to the exponent of $\bar{c}$. The only exceptional case is the post normalization case. It occurs when $\bar{a}+\bar{b}$ rounds upward to produce $S_c = 1.00\cdots0$ exactly. In this case the exponent of $\bar{c}$ is larger by 1 than that of $\bar{a}+\bar{b}$. Ignoring, for the moment, the exceptional case (it will be further discussed later) we note that if we align the significands S_a, S_b, S_c according to the values of Q_a, Q_b, some of the trailing bits of S_a, S_b are positioned to the right of the least sinificant bit of S_c and form *tails*:

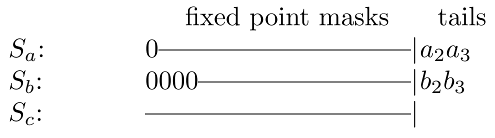


We may present to the fixed point generator the shifted and truncated masks of S_a, S_b. In addition we present to the fixed point generator a carries mask $M_C =$"$0xx\cdots xC_p$". Here $C_p \in \{0, 1, 2\}$ is the contribution of the tails, which is the combined effect of carry and of rounding.

The idea is to choose a numerical value, of 0, 1 or 2, for C_p, and then generate the tails, on the one hand, and S_c and the left parts of S_a, S_b, on the other hand (by the fixed point generator).

5.2 The tails triplet

In order to see, in more detail, how we do that, let us denote the leftmost bits of the tails by a_2, b_2, respectively, and the remainders of the tails by a_3, b_3:



Clearly, the value of C_p is determined by a_2, b_2, a_3, b_3.

Since either $Q_a \in \{0, 1\}$ or $Q_b \in \{0, 1\}$, we have either $a_3 = 0$ or $b_3 = 0$. So, if we denote $c_3 = a_3 + b_3$ we have $c_3 = a_3$ if $Q_b \in \{0, 1\}$ and $c_3 = b_3$ if $Q_a \in \{0, 1\}$. C_p depends on a_2, b_2, c_3.

Actually we do not need to know the whole sequence of bits of c_3, but only the result of an OR operation over all of its bits, $OR(c_3)$. We call the triplet of bits $(a_2, b_2, OR(c_3))$, the *tails triplet*. Thus, for instance, if the tails triplet is $(1,1,0)$ and the rounding mode is *up* then $C_p = 1$. If it is $(1,0,1)$ and the rounding mode is *to nearest* then $C_p = 1$ etc.

Let us list all of the tails triplets possible, for each of the rounding modes and for each of the possible values of C_p:

- round *down*

  $C_p = 0$**:** $(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1)$
  $C_p = 1$**:** $(1,1,0), (1,1,1)$

- round *up*

  $C_p = 0$**:** $(0,0,0)$
  $C_p = 1$**:** $(0,0,1)^0, (0,1,0)^0, (0,1,1)^1, (1,0,0)^0, (1,0,1)^1, (1,1,0)$
  $C_p = 2$**:** $(1,1,1)^0$

- round *to nearest/even*

  $C_p = 0$**:** $(0,0,0), (0,0,1), (0,1,0)_0, (1,0,0)_0$
  $C_p = 1$**:** $(0,1,0)_0^0, (0,1,1)^2, (1,0,0)_0^0, (1,0,1)^2, (1,1,0), (1,1,1)$

Some of the listed triplets have a numerical subscript and/or superscript: The subscript 0 means that the corresponding triplet implies round to even case. So, it is possible, with the indicated C_p value, only if the last character of S_c is forced to be $'0'$. The existence of a superscript indicates that the rounding component of the contribution of the tails is 1, which means that a result with $S_c = 1.00\cdots0$ is post normalized and is potentially wrong (This will be further discussed later).

The generation of S_a, S_b, S_c will start by constructing a three character mask for the tails triplet. The elements, of this mask, which correspond to a_2 and b_2 are simply copied from M_{Sa}, M_{Sb} or they are set to be $'0'$ if a_2 and/or b_2 fall outside of the range of the corresponding shifted mask. The element which corresponds to c_3 is set to be $'1'$ if the corresponding part of M_{Sa} or M_{Sb} includes at least one $'1'$ character. If not, the element will be $'0'$ if no $'x'$ exists in the appropriate part of M_{Sa} or M_{Sb} and $'x'$ otherwise.

After the mask of the tails triplet is ready we choose one tails triplet, from the complete list above, which is compatible with this mask. Note that in the case of round *to nearest/even* each of the triplets $(0,1,0)$, $(1,0,0)$ appears twice. Once with $C_p = 0$ and once with $C_p = 1$. These appearances are considered to be distinct. Namely, a choice like $(0,1,0)$ with $C_p = 0$ is different from the choice $(0,1,0)$ with $C_p = 1$.

Now, that we have chosen a tails triplet, the construction of the two tails is straight forward. The construction of S_c and of the left hand parts of S_a, S_b is performed by the fixed point generator.

5.3 The exceptional cases

Let us discuss now the exceptional case where the exponent of $\bar{a}+\bar{b}$ is smaller by 1 than that of $\bar{c}$:

If the tails triplet, chosen from the list, does not have any superscript it means that even if the generated S_c is $1.00\cdots 0$, we do not have post normalization. So, the result is correct and acceptable.

If the tails triplet has a superscript and the fixed point generator produces $S_c = 1.00\cdots 0$ then it means that some thing may be wrong with the result and we cannot be sure that the produced S_a, S_b, S_c satisfy $S_c = round(2^{-Q_a}S_a + 2^{-Q_b}S_b)$, as it should. In more details:

If the superscript is 0 then the result is definitely wrong and cannot be corrected, so it should be discarded and a repeated significands construction should be tried.

If the superscript is 1, it means that the result is correct, after all, and should be accepted.

If the superscript is 2, it means that the c_3 part of the tails should be constructed with care: Note that there are only two tails triplets with a superscript 2, in the list. They are $(0,1,1)$, $(1,0,1)$ in the round *to nearest/even* mode, with $C_p = 1$. Since the resulting S_c was $1.00\cdots 0$ it means that the $p+1$ first bits of the exact sum were $011\cdots 1$ and that c_3 should be concatenated to the right of this, in order to produce the complete exact sum. This means that, in order for the result, generated by the fixed point generator to be usable the leftmost bit of c_3 should be 1. The rest of the bits of c_3 are not important. If the leftmost bit of c_3 cannot be chosen to be 1, because of mask constraints, it means that the solution should be discarded and repeated construction of a solution should be tried.

Note that no possible solution of the significands generator problem is excluded by the method of generation described above. Not even those with $S_c = 1.00\cdots 0$ and exact sum $011\cdots 1c_3$.

5.4 Structure of the algorithm for the significands generator in the addition of non-negative machine numbers

1. Using Q_a, Q_b and M_{Sa}, M_{Sb}, produce a three characters mask for the tails triplet.

2. Construct a sublist of all tails triplets, from the complete list of tails triplets, which are compatible with the mask.

3. Choose one tails triplet out of the sublist, at random. However, if the sublist is empty state that there is no solution and stop.

4. Invoke the fixed point generator in order to construct S_c and the left hand parts of S_a, S_b. If the fixed point generator states that there is no solution,

erase the chosen tails triplet, from the sublist, and go back to step 3.

5. Use the chosen tails triplet to construct the tails and by this complete the construction of S_a, S_b. Return S_a, S_b, S_c and stop.

6 Analysis of the significands generator for subtraction

Let us denote $c = \bar{a} - \bar{b}$. Then, $\bar{c} = round(c)$. We also denote the rounding error by $\varepsilon = |\bar{c} - c|$. In case c is rounded down we have $\bar{c} + \varepsilon = \bar{a} - \bar{b}$ or $\bar{b} + (\bar{c} + \varepsilon) = \bar{a}$. In case c is rounded upward we have $\bar{c} - \varepsilon = \bar{a} - \bar{b}$ or $\bar{b} + \bar{c} = (\bar{a} + \varepsilon)$. In either case we have an exact identity which includes only one addition of non-negative numbers. We consider these numbers to be fixed point numbers. Had we have masks for these three fixed point numbers, we could use the fixed point generator to generate the three numbers. Let us see that this is exactly what we have: Note that the non-zero bits of ε always lie to the right of the least significant bit of $\bar{c}$. Also, $\bar{a} \geq c \Rightarrow \bar{a} \geq \bar{c}$ so, the non-zero bits of ε lie to the right of the least significant bit of $\bar{a}$ as well. This means that the bits of $\bar{c} + \varepsilon$ are composed of the bits of $\bar{c}$ and the bits of ε written in sequence, one after the other. A similar thing is true for $\bar{a} + \varepsilon$. We may describe the implication of this fact graphically: Assume, for instance, that the rounding mode is *down*, that $Q_b = 3$ and that $Q_c = 1$. Then the masks for $\bar{a}$, $\bar{b}$, $\bar{c} + \varepsilon$ may be chosen to be,

		S_b		
$\bar{b}$:	000	————————		
		S_c	ε	
$\bar{c} + \varepsilon$:	0	————————	xx	
		S_a		
$\bar{a}$:		————————	000	

We see that these masks are composed of the masks for S_a, S_b, S_c, padding 0 characters and padding x characters, which represent the bits of ε. We may use the fixed point generator to generate the three numbers and then extract from their binary representations the bits of S_a, S_b, S_c. In this particular case we have for the fixed point generator $N = p + 3$ and the mask for the sequence of carries is "$0xx \cdots xx0$". It turns out that the maximal value for N necessary is in the order of magnitude of $2p$, because if Q_b is larger than p we have to know only if $\bar{b} > 0$ or not and the details of its bits are of no significance. A similar treatment may be used if the rounding mode is *up*. In such a case the bits of ε must be added to those of $\bar{a}$ instead of $\bar{c}$.

Q_b	0	0	1	1	0	0	$\cdots$	0	0	1	1	$\cdots$	1	1	2	3	$\cdots$	$p-1$	$\geq p$	2	3	$\cdots$	p	$> p$
Q_c	0	1	0	1	2	3	$\cdots$	$p-1$	$\geq p$	2	3	$\cdots$	p	$> p$	0	0	$\cdots$	0	0	1	1	$\cdots$	1	1

Table 3: Pairs Q_b, Q_c for subtraction

6.1 More details for the *down* rounding mode

The identity that must be used, in this case, is $\bar{b}+(\bar{c}+\varepsilon) = \bar{a}$. The combinations of Q_b, Q_c which should be considered are listed in Table 3.

The common length of the masks that must be presented to the fixed point generator, for most of the combinations, is $N = p+max(Q_b, Q_c)$. The mask for S_b should be padded to the left by Q_b $'0'$ characters (if $Q_b > 0$) and to the right by $Q_c - Q_b$ $'0'$ characters (if $Q_b < Q_c$). The mask for S_c should be padded to the left by Q_c $'0'$ characters (if $Q_c > 0$) and by $Q_b - Q_c$ $'x'$ characters to the right (if $Q_c < Q_b$). The mask for S_a never has to be padded to the left (because $\bar{a} \geq \bar{b}$, $\bar{a} \geq \bar{c}$) and should be padded to the right by $max(Q_b, Q_c)$ $'0'$ characters (unless $Q_b = Q_c = 0$). The mask of the carries must always be of the form "$0xx \cdots xx0$".

The cases $(Q_b, Q_c) \in \{(0, \geq p), (1, > p), (\geq p, 0), (> p, 1)\}$ should be treated in a slightly different way:

In the two cases where $Q_c \geq p$ or $Q_c > p$ we must have $S_c = 0$, unless this is not compatible with its mask, in which case there is no solution. Generating a solution, then, is straightforward.

In the two cases where $Q_b \geq p$ or $Q_b > p$, S_b may be any bit string which is compatible with its mask. The three number masks that must be presented to the fixed point generator are of length $N = p + Q_c + 1$ each, where the mask for $\bar{b}$ is composed only of $'0'$ characters except for the rightmost one, which is $'1'$ if $\bar{b} \neq 0$ and is 0 otherwise. The mask for S_c is padded by Q_c $'0'$ characters to the left and by a single $'x'$ to the right. The mask for S_a is padded by $Q_c + 1$ $'0'$ characters to the right. After we determine a solution to the fixed point problem we may easily construct the bit strings for S_a, S_b, S_c.

6.2 More details for the *up* rounding mode

The identity which must be used in this case is $\bar{b}+\bar{c} = (\bar{a}+\varepsilon)$. The combinations of Q_b, Q_c which should be considered are again those listed in Table 3. Again, in most cases, the length of the masks, presented to the fixed point generator is $N = p + max(Q_b, Q_c)$. The mask for S_b is padded by Q_b $'0'$ characters to the left (if $Q_b > 0$) and by $Q_c - Q_b$ $'0'$ characters to the right (if $Q_b < Q_c$). The mask for S_c should be padded by Q_c $'0'$ characters to the left (if $Q_c > 0$) and by $Q_b - Q_c$ $'0'$ characters to the right (if $Q_c < Q_b$). The mask for S_a does not have to be padded on its left side. On its right side it must be padded by Q_c $'0'$

characters (if $Q_c > 0$) and by $Q_b - Q_c$ $'x'$ characters to their right (if $Q_c < Q_b$). The mask for the carries is, again, of the form "$0xx \cdots xx0$".

In the four cases where either $Q_c \geq p$ or $Q_b \geq p$ the treatment is, again, slightly different:

In the two cases where $Q_c \geq p$, S_c must be zero, unless this is not compatible with its mask, in which case there exists no solution.

In the two cases where $Q_b \geq p$ we clearly have $\varepsilon = \bar{b}$ and we must have $\bar{c} = \bar{a}$ ($Q_c = 1$ is impossible, then). So, S_b may be chosen to be any number which is compatible with its mask and $S_c = S_a$ may be chosen to be any positive number which is compatible with both the masks of S_a and of S_c.

In the discussion above we assumed implicitely that the exponents of c and of $\bar{c}$ are the same. This is always so in the case of round *down*. However, if *round*() is round *up* there exists one exceptional case: If the significand of $\bar{c}$ is $1.00 \cdots 0$ and $\varepsilon > 0$ then this implicit assumption is not satisfied. Unless the leftmost bit of ε (the one which corresponds to the leftmost $'x'$ character in the right padding of S_a) is 0 this leads to an error. So, solutions, returned by the fixed point generator, in which $S_c = 1.00 \cdots 0$ and the leftmost bit of ε is 1, should be rejected and an additional attempt to produce a solution should be made.

6.3 More details for the *nearest/even* rounding mode

In this case the rounding is sometimes *up* and sometimes *down*. So, the algorithm for this type of rounding is a kind of a mixture of the algorithms for round *up* and for round *down*. We consider, again, the combinations of Q_b, Q_c listed in Table 3. In the same way that we extended and padded the masks for S_a, S_b, S_c for round *down* and round *up* we extend and pad those masks also in the round to nearest case. The new element is division of the dicussion of each combination of Q_b, Q_c into four subcases:

1. *Case of round to nearest/down*, in which we use the identity $\bar{b} + (\bar{c} + \varepsilon) = \bar{a}$ and pad the extended masks of S_a, S_b, S_c in the same way it is done in the round *down* case, except that the "$xx \cdots x$" padding that corresponds to ε is replaced by "$0xx \cdots x$" padding.

2. *Case of round to nearest/up*, in which we use the identity $\bar{b} + \bar{c} = (\bar{a} + \varepsilon)$ and pad the extended masks of S_a, S_b, S_c in the same way it is done in the round *up* case, except that the "$xx \cdots x$" padding, that corresponds to ε, is replaced, again, by "$0xx \cdots x$" padding. Similar to the round *up* case we should reject a solution with $S_c = 1.00 \cdots 0$, unless the leftmost x (of ε) is replaced in the solution by 0 (i.e. the bits of ε are compatible with "$00xx \cdots x$") or the bits of ε in the solution are $0100 \cdots 0$ exactly.

3. *Case of round to even/down*, which is exactly like case 1 above, except that the "$0xx \cdots x$" padding is replaced by "$100 \cdots 0$" padding and the

last character of the mask of S_c is replaced by $'0'$ (this can be done only if the original last character of M_{Sc} is $'0'$ or $'x'$).

4. *Case of round to even/up*, which is exactly like case 2, except that the "$0xx\cdots x$" padding is replaced by "$100\cdots 0$" padding and the last character of the mask of S_c is replaced by a $'0'$ (again, this can be done only if the original last character of M_{Sc} is $'0'$ or $'x'$). If the solution returned is such that $S_c = 1.00\cdots 0$ it must be rejected. So, a solution with $S_c = 1.00\cdots 0$ cannot be produced in this case. Note, though, that such a solution may be produced in case 2 (see the end of case 2, where the bits of ε are $0100\cdots 0$).

This discussion may be easily completed to include also the cases where $Q_b \geq p$ or $Q_c \geq p$, which were not mentioned here.

6.4 Structure of the algorithm for the significand generator in the subtraction of positive numbers case

Given numerical values for Q_b, Q_c we use Table 3 to classify this pair so we can use the appropriate procedure, out of those described in the previous section, to generate three significands. If the construction was successful return the solution and stop. If the construction failed state that there is no solution and stop.

7 Analysis of the biased exponents generators problem

In both generators I and II we have $q_1 \in \{0, 1\}$, $e_3 = e_1 + q_1$. There are then two possible cases, $e_3 = e_1$ and $e_3 = e_1 + 1$.

7.1 Generator I

In the case $e_3 = e_1$, the common value of e_1 and e_3 must be compatible with both the masks M_1 and M_3. If the two masks have different number characters in the same position then they are contradictory and no pair e_1, e_3 exists. Otherwise, it is very easy to produce their intersection, M_{13}. Hence we are left with the problem of producing e_2 and e_3 which satisfy $e_3 = e_2 + q_2$. We have masks for e_2, e_3 (i.e. M_2, M_{13}) and we may construct a mask (composed of numerical characters only) for q_2. Hence, clearly, the problem may be solved by the fixed point generator.

In the case $e_3 = e_1 + 1$, we note that the right hand end of the string of bits e_1, must be one of the following: $0, 01, 011, 0111, \ldots, 011\cdots 1$ (the last string is of length w). Since $e_3 = e_1 + 1$, the right hand end of e_3 must be, respectively:

$1, 10, 100, 1000, \ldots, 100\cdots0$ (here also the last string is of length w). Comparing the possible right ends of e_1, e_3 with the masks M_1, M_3 we usually may erase some of the possibilities and are left with a reduced list of pairs of right hand ends. Choosing one of these pairs, the left ends of e_1, e_3 must be identical. This means that the masks of e_1 and e_3 may be chosen to be composed of known numerical characters in the right ends and of the intersection of the left hand ends of M_1 and M_3 (if the left ends of M_1 and M_3 are contradictory then the corresponding pair of right ends, should be erased from the list).

Thus, every time we choose a pair of right ends of e_1, e_3, from the list, we are in a position similar to the one we have in the case $e_1 = e_3$: We have masks, M_2, M_{13}, for e_2, e_3 and a mask for q_2 and must find e_2, e_3 from the relation $e_3 = e_2 + q_2$. This, again, can be solved by the fixed point generator. If the generator states that there is no solution it means that the chosen pair of right ends should be erased from the list of pairs and another pair should be tried. If the list of pairs becomes empty it means that there is no solution which satisfies $e_3 = e_2 + 1$.

7.2 Generator II

The analysis of generator II is similar to the analysis of generator I, up to the point where we have a new mask, M_{13}, for e_3 (this applies to the case $e_3 = e_1$ as well as to the case $e_3 = e_1 + 1$). Thus, the problem we face now is to generate q_2, e_2, e_3 where we have masks M_2, M_{13} for e_2, e_3, respectively, and we must satisfy the relation $e_3 - e_2 = q_2 > q$.

The smallest e_2 which is consistent with M_2 is obtained by replacing each $'x'$ in M_2 by $'0'$. Denote the result by $\underline{e}_2$. The largest e_3 which is consistent with M_{13} is obtained by replacing each $'x'$ in M_{13} by a $'1'$. Denote the result $\bar{e}_3$. There exists a solution, to the generation problem II, if and only if $\bar{e}_3 - \underline{e}_2 > q$.

In case there exists a solution we want to choose, at random, a pair $\tilde{e}_2$, $\tilde{e}_3$ for which $\tilde{e}_3 - \tilde{e}_2 > q$, and $\tilde{e}_2$, $\tilde{e}_3$ are compatible with M_2, M_{13}, respectively. We describe a way to do that below:

Erasing from M_2 all of the $'0'$ and $'1'$ characters, we are left with a submask which is composed of $'x'$ characters only. The numbers $\dot{e}_2$ which are compatible with this submask, are in a natural 1-1 correspondence with the numbers e_2, which are compatible with M_2. Clearly e_2 is increasing as a function of $\dot{e}_2$ and vice versa. Similar relations exist between e_3 and $\dot{e}_3$ via the mask M_{13}.

The construction of random $\tilde{e}_2$, $\tilde{e}_3$, which are compatible with the masks M_2, M_{13}, respectively, and satisfy $\tilde{e}_3 - \tilde{e}_2 > q$, may go as follows (In this decription, $\dot{e}_2$, $\underline{\dot{e}}_2$, $\dot{\bar{e}}_2$, $\dot{\tilde{e}}_2$ correspond to e_2, $\underline{e}_2$, $\bar{e}_2$, $\tilde{e}_2$, respectively, by the 1-1 correspondence, via M_2, and $\dot{e}_3$, $\dot{\bar{e}}_3$, $\dot{\tilde{e}}_3$, $\underline{\dot{e}}_3$ correspond to e_3, $\bar{e}_3$, $\tilde{e}_3$, $\underline{e}_3$, respectively, via M_{13}):

1. Compute $\underline{e}_2$, $\bar{e}_3$ with $\underline{\dot{e}}_2 = 00\cdots0$, $\dot{\bar{e}}_3 = 11\cdots1$, respectively.

2. Compute $\dot{\bar{e}}_2 = max\{\dot{e}_2 | \bar{e}_3 - e_2 > q\}$ by a binary search (a bisection-like algorithm).

3. Choose an integer $\dot{\tilde{e}}_2 \in [\underline{\dot{e}}_2, \dot{\bar{e}}_2]$, at random.
4. Compute $\underline{\dot{\tilde{e}}}_3 = min\{\dot{e}_3 | e_3 - \tilde{e}_2 > q\}$ by a binary search.
5. Choose an integer $\dot{\tilde{e}}_3 \in [\underline{\dot{\tilde{e}}}_3, \dot{\bar{e}}_3]$ at random.

Knowing $\dot{\tilde{e}}_2$, $\dot{\tilde{e}}_3$ we know also $\tilde{e}_2$, $\tilde{e}_3$. We set $e_2 = \tilde{e}_2$, $e_3 = \tilde{e}_3$. As for e_1, its right end is known and its left end may be copied from e_3. Also, we may take $q_2 = \tilde{e}_3 - \tilde{e}_2$.

7.3 Structure of the algorithms for the biased exponents generators

The algorithms for the case $q_1 = 0$ are outlined below. In the case $q_1 = 1$ the algorithms are similar, apart from obviouse complications, due to the possibility of several M_{13} masks.

7.3.1 Generator I

1. If M_1 and M_3 are contradictory, state that there is no solution and stop. Else, construct M_{13}.
2. Present M_2, M_{13}, q_2 to the fixed point generator to generate e_2, e_3 satisfying $e_3 = e_2 + q_2$.
3. If the fixed point generator states that there is no solution, state that there is no solution and stop. Else, construct e_1 from e_3.
4. Return e_1, e_2, e_3 and stop.

7.3.2 Generator II

1. If M_1 and M_3 are contradictory, state that there is no solution and stop. Else, construct M_{13}.
2. Produce random $\tilde{e}_2$, $\tilde{e}_3$ which are compatible with M_2, M_{13}, respectively, and satisfy $\tilde{e}_3 - \tilde{e}_2 > q$. If such $\tilde{e}_2$, $\tilde{e}_3$ do not exist, state that there is no solution and stop.
3. Set $e_2 = \tilde{e}_2$, $e_3 = \tilde{e}_3$, $q_2 = \tilde{e}_3 - \tilde{e}_2$ and construct e_1 from e_3
4. Return e_1, e_2, e_3, q_2 and stop.

8 Conclusions

We described an algorithm that generates floating point numbers $\bar{a}$, $\bar{b}$, $\bar{c}$ which are compatible with three corresponding arbitrary masks and satisfy $\bar{c} = round(\bar{a} \pm \bar{b})$. The solution is general enough to be applicable to all floating point data types and all rounding modes, mentioned in the IEEE standard 754. Although the target data types are floating point data types, the main engine is a fixed point generator. We expect this engine to be useful in other applications as well. The complexity of the algorithm is polynomial and tests indeed confirm that it is fast: solutions are found almost immediately for a large variety of instances.

9 Acknowledgement

Thanks are due to Sigal Asaf, Anatoly Koyfman and Shy Tzadok who suggested the method of applying the fixed point generator to the significands generator for subtraction.

10 Bibliography

[**1**] IEEE standard for binary floating point arithmetic. An American national standard, ANSI/IEEE Std 754-1985.

[**2**] B. Beizer: "Software Testing Techniques". Van Nostrand Reinhold, 1990.

[**3**] Y. Lichtenstein, Y. Malka and A. Aharon, "Model-Based Test Generation For Processor Design Verification". Innovative Applications of Artificial Intelligence (IAAI), AAAI Press, 1994.

[**4**] L. Fournier, D. Lewin, M. Levinger, E. Roytman and Gil Shurek: "Constraint Satisfaction for Test Program Generation". Int. Phoenix Conference on Computers and Communications, March 1995.

[**5**] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho and G. Shurek: "Test Program Generation for Functional Verification of PowerPC Processors in IBM". 32nd Design Automation Conference, San Francisco, June 1995, pp. 279-285.

[**6**] B. Marick: "The Craft of Software Testing, Subsystem Testing Including Object-Based and Object-Oriented Testing". Prentice-Hall, 1995.

[**7**] C. Kaner: "Software negligence and testing coverage". In proceedings of STAR 96: the Fifth International Conference, Software Testing, Analysis and Review, pages 299-327, June 1996.

[**8**] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv: "User defined coverage - a tool supported methodology for design verification". In the Proceedings of the 35th Design Automation Conference (DAC), pages 158-163, June 1998.

[**9**] Edmund M.Clarke, Steven M.German and Xudong Zhao: "Verifying the SRT Division Algorithm Using Theorem Proving Techniques". Formal Methods in system Design, volume 14, number 1, January 1999, pages 7-44.

[**10**] David M.Russinoff: "A Mechnically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode". Formal Methods in System Design, volume 14, number 1, January 1999, pages 75-125.

[**11**] J. o'Leary, X.Zhao, R.Gerth, and C.-J.H. Seger: "Formally verifying IEEE compliance of floating-point hardware". Intel Technology Journal, Vol 1999-Q1, pp 1-14. Available on the Web as

http://developer.intel.com/technology/itj/q11999/articles/art_5.htm

[**12**] L. Fournier, Y. Arbetman, M. Levinger: "Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator". Application to the x86 Microprocessors Family. DATE99, Munchen, 1999.

[**13**] Mark D.Aagaard, Robert B.Jones, Roope Kaivola, Katherine R.Kohastsu, Carl-Johan, J.Seger: "Formal Verification of Iterative Algorithms in Microprocessors". Interl Corporation, Hillsboro, Oregon, USA, DAC 2000.

[**14**] Silvia M. Mueller and Wolfgang J. Paul: "Computer Architecture, Complexity and Correctness", Springer-Verlag Berlin Heidelberg 2000.