FP-ANR: A representation format to handle floating-point cancellation at run-time

David Defour*

LAMPS, Univ. Perpignan Via Domitia, F-66860, Perpignan, France
* *david.defour@univ-perp.fr*

***Abstract*—When dealing with floating-point numbers, there are several sources of error which can drastically reduce the numerical quality of computed results. One of those error sources is the loss of significance or cancellation, which occurs during for example, the subtraction of two nearly equal numbers. In this article, we propose a representation format named Floating-Point Adaptive Noise Reduction (*FP-ANR*). This format embeds cancellation information directly into the floating-point representation format thanks to a dedicated pattern. With this format, insignificant trailing bits lost during cancellation are removed from every manipulated floating-point number. The immediate consequence is that it increases the numerical confidence of computed values. The proposed representation format corresponds to a simple and efficient implementation of significance arithmetic based and compatible with the IEEE Standard 754 standard.**

1. Introduction

Floating-point numbers, which are normalized by the IEEE Standard 754 standard [1], correspond to a bounded discretization of real numbers. Therefore, a floating-point number corresponds to the representation of an exact number combined with errors due to discretization, accumulation of rounding errors or cancellation. In other words, a floating-point number embeds useful information along with noise linked to those errors.

When numerical noise becomes dominant, for example during catastrophic cancellation, there are no more useful bits of information in the representated numbers. Unfortunately, the occurence of this situation is undetectable just by looking at the representation. This is due to the fact that with the widely used IEEE Standard 754 representation format, there is no way of distinguishing useful numerical information from noise. This problem has been identified and addressed since the late 1950s with significance arithmetic [2]. Significance arithmetic addressed these issues by tailoring the number of digits to their needs.

Significance arithmetics is regaining interest thanks to the Unum proposal [3] or indirectly through numerous problems encountered with exascale computers and the lack of confidence in numerical results [4]. If the Unum system is based on real problems, the proposed solution is subject to criticism for numerous reasons as pointed out by W. Kahan [5]. On the other hand, indirect solutions based on software solutions to detect cancellation [6], [7], or avoiding rounding errors [4] are not meant to be efficient nor effective for real time execution.

This article proposes a new way to represent significant information in floating-point numbers. The solution consists of an altered IEEE Standard 754 representation format of the mantissa. That information is stored using a simple pattern that replaces insignificant digits. This makes such number representation almost as accurate as original IEEE Standard 754 numbers. Therefore, the proposed solution corresponds to a simple, efficient and IEEE Standard 754 compliant implementation of significance arithmetic.

2. Preliminaries

Floating-point numbers are approximations of real numbers. The concept of approximation is associated with the concept of errors. Digits of a floating-point representation number can be split into two parts; a significant and an insignificant part. This section provides some background on IEEE Standard 754 floating-point arithmetic, errors and significance arithmetic.

2.1. The IEEE Standard 754 standard

The current version of the floating-point standard, the IEEE Standard 754[-2008] [1] published in August 2008, includes the original binary formats along with three new basic formats (one binary and two decimal).

Definition 1 (Floating-Point Numbers)**.** *A IEEE Standard 754 representation format is a "set of representations of numerical values and symbols" made of finite numbers, two infinities and two kinds of NaN (Not A Number). The set of finite numbers are described by a set of three integers (s,m,e) corresponding respectively to the sign, the mantissa and the exponent. The numerical value associated with this representation is*

$$(-1)^s \times m \times b^e.$$

Values that can be represented are determined by the base or radix b (2 or 10), the number (p) of digits in the mantissa and the exponent parameter $emax$ such that:

$$0 \leq m \leq b^p - 1$$

and

$$1 - emax \leq e + p - 1 \leq emax$$

It should be pointed out that the number $e+p-1$ is called the "exponent" in some literature.

The value Zero is represented with a 0 mantissa and a sign bit specifying a positive or negative zero.

In the case of binary formats, representation of finite numbers is made unique by choosing the smallest representable exponent. Numbers with an exponent in the normal range have the leading bit set to 1. It corresponds to an implicit bit as it is not present in the memory encoding, allowing the memory format to have one more bit of precision. This extra bit is not present for subnormal numbers which have an exponent outside the normal exponent range.

For example, the IEEE Standard 754 double precision format (or binary64) is represented with 64 bits which are split into 1 sign bit, $p = 52$ bits of mantissa and $e = 11$ bits of exponent, whereas single precision format (or binary32) is represented with 32 bits split into 1 sign bit, $p = 23$ bit of mantissa and $e = 8$ bits of exponent.

2.2. Floating-Point Errors

Floating-point numbers representation format differs by their radix and the number of bits used for their encoding. The 2008 revision of the IEEE Standard 754 defines formats for radix 2, ranging from 16 to 128 bits. For each of these formats, the number of bits that represent the exponent and the mantissa is fixed. Therefore, the floating-point representation of numerical value have to be either rounded or padded with zeros in the least significant digits of the mantissa. This means that by construction, FP numbers embed errors in their representation. These errors can be separated into three groups: data uncertainty, rounding and cancellation.

2.2.1. Uncertainty. Uncertainty [8] in data is linked to initial input values produced by measurements, experimentations using physical sensors, or numerical model such as polynomial approximation [9]. For example, a physical sensor producing the twenty digit value $x = 12345.678901234567890$ with a process exhibiting an uncertainty of $U = 10^{-5}$, corresponds to a real value in the interval $[x \cdot (1 - U); x \cdot (1 + U)] = [12345.555; 12345.802]$. This translates into 5 significant digits, the rest of the information corresponds solely to noise or insignificant digits. As floating-point numbers are of a fixed size, noise is kept in the representation of those numbers and remains present in all computation that follows. As those extra digits do not carry any numerical meaning, it may lead to an overconfidence in the numerical quality of the result.

2.2.2. Rounding. Because floating-point numbers have a limited number of digits, they cannot represent real numbers accurately. When there are more digits than the format allows, the number is rounded and the leftovers are omitted. The standard defines five rounding rules, two rounding to the nearest (ties to even, ties away from zero) and three directed rounding (toward $0, -\infty, +\infty$). Floating-point operations in IEEE Standard 754 satisfy:

$$fl(a \circ b) = (a \circ b) \cdot (1 + \epsilon) \qquad |\epsilon| \leq u \qquad \circ \in \{+, -, \times, /\}$$

Where $u = b/2 \cdot b^{-p}$ depends on the radix b and the precision p , and $fl()$ denote the result of a floating-point computation.

2.2.3. Cancellation. Cancellation occurs when two nearby quantities are subtracted and the most significant digits cancel each other. Cancellations are very common but when many digits are lost, the effect can be severe as the number of informative digits is reduced. In that case, this results in catastrophic cancellation that has a dramatical impact on the sequel of the computation.

For example, let $x = 1.5 \times 2^0$ and $y = 1.0 \times 2^{26}$ be two floating-point numbers stored in binary32 format. Then the sequence of operations $r = fl(fl(x + y) - y)$ produces the result $r = 0.0$ which has no correct digits, as the correct real result should be 1.5. This is due to the catastrophic cancellation which occurred during the subtraction. Such cancellations cannot be detected without additional examination of the source and destination of data elements, leaving no trace of the fact that $r = 0.0$ was completely incorrect.

Such sequences are used for example in numerical algorithms that compute errors such as the 2sum algorithm [10], [11].

2.3. Significance arithmetic

Significance arithmetic [2], [12], [13] brings a solution to the problem of representing an approximation of the error along with floating-point numbers. It relies on the concept of significant and insignificant digits.

Definition 2 (Significant and insignificant digits)**.** *Significant digits of a number are digits that carry meaning contributing to a number. The number of significant digits for a p-digits number X is represented by α_X and the number of insignificant digits $p - \alpha_X$.*

Significance arithmetic sets two methods to calculate a bound for the propagated and generated error called *normalized significance* and *unnormalized significance*. The normalized significance always keeps the floating-point number normalized and provides an index of significance. The unnormalized significance does not normalize floating-point numbers and uses the count of digits remaining after leading zeros as an indication of their significance.

The normalized method allows as many digits as possible of a number to be retained. This requires an added index that defines the number of significant digits. There exists software implementations of significance arithmetic such as for the FORSIG [14] library written in Fortran, or Python [15]. With the unnormalized method [16], only digits considered significant are retained.

The integration of a specific pattern in the mantissa to categorize significant and insignificant digit has already been proposed for decimal computer in the BCD format [17]. It relies upon unused bit patterns in the BCD format which are bit-field 1010 and 1011 corresponding to respectively digits 10 and 11. More recently, Gustafson [3] extended significance arithmetic by proposing the Unum representation format which is able to represent exact and approximate numbers with varying mantissa and exponent field length.

Even though significance arithmetic offers an approximation of the error, it is not suitable for every numerical problem related to the management of error. In particular, significance arithmetic is not meant for self-correcting numerical algorithm.

3. A format to embed cancellation information

The proposed representation format: *Floating-Point Adaptive Noise Reduction* (*FP-ANR*) is detailed in this section. It allows the user to split the mantissa in two: the significant and insignificant part. Insignificant digits, or noise, can come from initial uncertainty, or cancellation generated during computation. This format corresponds to an implementation of significance arithmetics based on the existing IEEE Standard 754 format. In this article, we will consider the radix-2 arithmetic, where bit or digit will refer to the same notion.

3.1. The representation format

Our goal is to propose a non-intrusive solution while being able to keep track of uncertainty due to cancellation. By non-intrusive, we mean that the proposed solution must be compatible with existing floating-point representation format without exhibiting a large overhead. This discards any solutions relying on shadow memory, or extra fields.

The proposed format, named FP-ANR, is based on a modification of the mantissa that integrates information on cancellation. The modification of the mantissa consists in replacing uninformative bits, or bits lost during cancellation, by a given pattern. This pattern must be self-detectable to avoid using extra fields as in the Unum. There are two possible patterns: First, a 1 followed by as many 0s as needed, second a 0 followed by as many 1s as needed. With any one of these solutions, one can easily deduce the number of cancelled bits by scanning the mantissa from right to left to detect the first 1 (or 0 respectively). The assembly instruction that performs this operation is usually named *Count Trailing Zero/One*. The rest of this article will focus on the first pattern (1 followed by 0s). The first 1 encountered from right to left in the mantissa will be called the *significant flag*.

With FP-ANR, one bit of the mantissa is used to represent the *significant flag*. It means that a number with a p-bit mantissa will have at most $p - 1$ informative bits which is 1 bit less than the corresponding IEEE Standard 754 representation format which FP-ANR is built upon. For example, the value 1.0 which corresponds to the binary32 IEEE Standard 754 representation number

Table 1. Binary representation of the value 1234.56 with an uncertainty $10^{-3}\%$

binary32	0 10001001 00110100101000111101100
FP-ANR	0 10001001 0011010010100011*0000000*

0 01111111 00000000000000000000000 will be represented in the FP-ANR format by

0 01111111 0000000000000000000000*1*

The rightmost bit equal to 1 and corresponding to the *significant flag*, indicates the position between significant and insignificant bit in the mantissa. In other words, this representation corresponds to the floating-point number 1.0 accurate up to 23 bits. Alternatively, the FP-ANR representation string

0 01111111 000000000000*10000000000*

corresponds to the floating-point number 1.0 as well, but accurate to 13 bits.

This slight modification affects the set of finite numbers as defined by the IEEE Standard 754 standard including normal and subnormal numbers. The representation format of special values which includes infinities, NaN and 0 remains unchanged as no *significant flag* is embedded.

The major difference between the IEEE Standard 754 representation format and the FP-ANR format is that IEEE Standard 754 can manipulate exact values such as 1.0 whereas FP-ANR deals solely with approximation (except for 0). This is a drawback as discussed in section 2.3, which is why the proposed format cannot be considered as a universal format.

3.2. Managing uncertainties

With FP-ANR, uncertainty is integrated directly in the mantissa. For example, let us consider a physical process which produces the value $x = 1234.56$ with an uncertainty $U = 10^{-5}$ and its representations (Table 1). With the IEEE Standard 754, there is no direct solution to integrate the information on uncertainty in the representation number. It is still possible to circumvent this problem by using interval arithmetic, but this will requires at least 2 numbers. With FPANR, the information on uncertainty is integrated by evaluating the number of significant digit, which corresponds to $\lfloor |log_2(U)| \rfloor = 16$ bits.

As we can observe, with the IEEE Standard 754 format the value will be translated directly into its binary format where the last 8 insignificant bits correspond to noise. Whereas with FP-ANR we can distinguish significant and insignificant bits.

When all signicant bits are lost we can keep track of that information, which is not the case with the other representation format. In that case we have information on the order of magnitude of the insignificance. This concept is similar to the concept of informatical zero represented by *@.0* in CADNA [7]. For example, let us consider the following number where all bits of the mantissa are set to 0 and only the implicit bit is set to 1.

0 01111111 *00000000000000000000000*

This representation number means that there are no significant bits in the mantissa. However, there is still useful embedded information which is the order of magnitude of the error stored in the exponent. This information can be used in further computation involving such a number: For example, in an addition to discard bits of weight less than the one corresponding to insignificant bit. It can potentially avoid a division by zero resulting from an unwanted catastrophic cancellation where all bits are lost.

3.3. Addition of FP-ANR

As we have seen in section 2.2.3, the least significant bits of the mantissa are usually uninformative as they corresponds to noise due to cancelation or discretization. The information on insignificant bit has to be propagated during operations. This can be done by updating the position of the *significant flag* found in the result of an addition between two FP-ANR as follow.

Let A, B and R be three FP-ANR numbers with respectively α_A, α_B and α_R significant bits. The number of significant bits α_R of the results $R = A \circ B$ with $\circ \in \{+, -\}$ is determined by:

$$\alpha_R = \exp_R - MAX((\exp_A - \alpha_A), (\exp_B - \alpha_B))$$

where $\exp_X$ corresponds to the exponents of the FP-ANR number X with $X \in \{A, B, R\}$. One can notice that the quantities $(\exp_A - \alpha_A)$ and $(\exp_B - \alpha_B)$ correspond to the absolute error.

3.4. Multiplication and division of FP-ANR

Propagation of significant information during multiplication corresponds to the simplest case. The number of significant bits resulting from a multiplication between two FP-ANR numbers is estimated as follows:

Let X with $X \in A, B, R$ be a FP-ANR representation of the number x with $x \in \{a, b, r\}$ respectively, with α_X significant bits. The number α_R of significant bits in the results $R = A \cdot B$ can be approximated by $\alpha_R = MIN(\alpha_A, \alpha_B)$. This approximation corresponds to the minimal number of bit lost during cancellation and does not consider the accumulation of it.

We chose to not consider the accumulation of uncertainty as its estimation would grow too fast during uncertainty propagation. This differs from interval arithmetic that will always overestimate the error. For comparison purposes, lets consider the case of accumulation of uncertainty. The number of significant bit α_X corresponds to an error e_X in X is such that $X = x \cdot (1 + e_X)$ with $|e_X| \leq 2^{-\alpha_X}$. The error for the multiplication $R = A \cdot B$ corresponds to $R = (a \cdot b) \cdot (1 + e_a + e_b + e_a \cdot e_b)$ and the error term $e_r = e_a + e_b + e_a \cdot e_b$ is such that $|e_r| \leq 2^{-\alpha_A} + 2^{-\alpha_B} + 2^{-\alpha_A - \alpha_B}$. The error e_r is maximal when $\alpha_A = \alpha_B$. A more accurate approximation that considers the accumulation of uncertainty for the multiplication is

$$\alpha_R = \begin{cases} MIN(\alpha_A, \alpha_B) - 2 \text{ when } \alpha_A = \alpha_B \\ MIN(\alpha_A, \alpha_B) - 1 \text{ when } \alpha_A \neq \alpha_B \end{cases} \quad (1)$$

For similar reasons, we decided to not consider the accumulation of uncertainty for the division $R = A/B$. The number α_R of significant bits in the results $R = A/B$ is set to $\alpha_R = MIN(\alpha_A, \alpha_B)$. This differs from a solution considering the accumulation of uncertainty as follows.

The error for the division can be expressed as $R = \frac{a}{b} \cdot \frac{1+e_a}{1+e_b}$. This formula can be rewritten by expressing the denominator term for the error as an infinite series $R = \frac{a}{b} \cdot (1 + e_a) \cdot (1 - e_b + e_b^2 + ...)$. Since the error e_b is required by the number format to be less than 1, e_b^2 and all the higher order terms can be neglected. The error term for the division $e_r = e_a - e_b - e_a \cdot e_b$ is such that $|e_r| \leq 2^{-\alpha_A} + 2^{-\alpha_B} + 2^{-\alpha_A - \alpha_B}$. Therefore, equation 1 corresponds to a more accurate approximation that consider the accumulation of uncertainty for the division.

3.5. Other operations using FP-ANR

We can consider the propagation of uncertainty in the case of more complex operations as well (e.g. exponential, logarithms or trigonometric function). Such functions have already been considered in previous work on significant arithmetic [2].

Let R and X be FP-ANR representations of the number r and x respectively, with α_R and α_X significant bits. We would like to estimate the number of significant bits α_R when $R = f(X)$ with f a function of X.

The number of significant digits can be approximated using results on the propagation of uncertainty. It can be done by looking at the extremum on the interval of values corresponding to the initial uncertainty interval $[X \cdot (1 - e_X); X \cdot (1 + e_X)]$ with e_X the error in X such that $|e_X| \leq 2^{-\alpha_X}$.

This uncertainty can be estimated using a first-order Taylor series expansion. It consists in replacing the function f by its local tangent:

$$f(X \cdot (1 + e_X)) = f(X) + f'(X) \cdot X \cdot e_X + o(X \cdot e_X)$$

with $o(x)$ a function which quickly tends toward 0. Therefore, the uncertainty in the result R can be estimated by:

$$e_R \approx |f'(X) \cdot X \cdot e_X|$$

This estimation is valid only if the function is considered quasi-linear and quasi-Gaussian on the interval $[X \cdot (1 - e_X); X \cdot (1 + e_X)]$. This corresponds to an estimation of the number of significant bits of the result α_R:

$$\alpha_R = log_2 \left| \frac{f(X)}{f'(X) \cdot X \cdot e_X} \right|$$

Table 2. APPROXIMATION OF THE NUMBER OF SIGNIFICANT DIGITS α_R FOR SOME FUNCTIONS $f(X)$

$R = f(X)$	$f'(X)$	$\alpha_R \approx$
$\sqrt{X}$	$-\frac{1}{2\cdot\sqrt{X}}$	$\alpha_X + log_2\|2\| = \alpha_X + 1$
$exp(X)$	$exp(X)$	$\alpha_X + log_2\|1/X\| = \alpha_X - log_2\|X\|$
$ln(X)$	$1/X$	$\alpha_X + log_2\|ln(X)\|$
$sin(X)$	$cos(X)$	$\alpha_X + log_2 \left\| \frac{sin(X)}{X\cdot cos(X)} \right\|$
$cos(X)$	$-sin(x)$	$\alpha_X + log_2 \left\| \frac{cos(X)}{X\cdot sin(X)} \right\|$

Combining this equation with the estimation of the number of significant digits $\alpha_X = -log_2\,|e_X|$, we get:

$$\alpha_R - \alpha_X \approx log_2 \left| \frac{f(X)}{f'(X)\cdot X} \right|$$

where log_2 is approximated using the exponent part of its floating-point representation format. Table 2 summarizes some of these approximation of the number of significant digits α_R for some functions $f(X)$.

3.6. Rounding in FP-ANR

It should be noticed that the presence of the *significant flag* is independent of the rounding problem. Therefore, we propose to use similar rounding strategies with FP-ANR as done with the IEEE Standard 754 format. The only difference being the bit position, where rounding will be done. With FP-ANR, rounding is operated on the last bit of the significant part, whereas it is done on the last bit of the mantissa for the IEEE Standard 754 representation format. However, the exact impact of rounding remains to be evaluated.

3.7. FP-ANR and the Table Maker's Dilemma

In addition to the propagation of the *significant flag*, there is another problem regarding elementary functions: the Table Maker's Dilemma [18]. The Table Maker's Dilemma corresponds to the problem of computing approximations of elementary functions with enough bits to ensure correct rounding. This problem is known to be difficult with the IEEE Standard 754 representation format since there is no bound on the number of bits required for every function and every format.

With FP-ANR, the Table Maker's Dilemma is circumvented as follows. One can set a target accuracy t function of the number of significant bits α_X of the input number X mandatory to evaluate the results of an elementary function. For example, one can set $t = 2\cdot\alpha_X$. If rounding can be done, then the process ends. If rouding is not possible, this corresponds to a hard to round case meaning that we are not sure of the last bit in the significant part. This uncertainty due to the Table Maker's Dilemma can be integrated in the FP-ANR format by left-shifting the *significant flag* one position. This way reproducibility and portability of the results provided by correct rounding is preserved.

3.8. Interaction between FP-ANR and IEEE Standard 754

One major advantage of FP-ANR is that it is compatible with the IEEE Standard 754 representation format. As with any format, compatibility can be assured thanks to conversion. Conversion between those two formats is straightforward as only the mantissa must be modified. From FP-ANR to IEEE Standard 754 format, this can be done by replacing the *significant flag* with a 0. From IEEE Standard 754 to FP-ANR format, this can be done by replacing the right-most bit of the mantissa by the *significant flag* (a 1 in the last position).

In addition to conversion, one can notice that the IEEE Standard 754 operators can process FP-ANR numbers. This will not lead to a crash or irrelevant results: it merely modifies the meaning of the insignificant bits. Nevertheless, it should not be considered as a serious issue as those bits correspond to noise. However when FP-ANR operators process IEEE Standard 754 numbers, the situation becomes more problematic, as the meaning of the resulting number depends on the position of the last bit set to 1.

4. Implementations

4.1. Software implementation

In this section, we describe a simplified software emulation of the proposed format. This section focuses on basic operations on the FP-ANR format related to the IEEE Standard 754 binary32 format.

Two C++ classes have been implemented to deal with single and double precision formats. These two classes are based on the header file of the CADNA library [7], where the code related to stochastic arithmetic is replaced with operations on significance arithmetic. This library can advantageously replace the IEEE Standard 754 formats (float, double) and major operations for those formats. It is available for download at http://perso.univ-perp.fr/david.defour/

One can notice that the biggest advantage of the FP-ANR over other solutions that require extra memory (shadow memory or extra fields), is that it could be easily integrated in a compiler pass. Indeed, memory allocation, bit manipulations (such as extraction of exponent, sign,...), tricky pointer manipulation are straightforward with the proposed format. However, such implementations is out of the scope for this article and will be developed in future work.

4.1.1. Conversion. Programs in Listing 1 rely on the *ieee754.h* header file provided by many Linux distributions. This header file defines the type *ieee754_float* that eases access to the bitfield of floating-point numbers. The two functions convert a number between binary32 and the FP-ANR format by managing the *significant flag* according to the rules defined in section 3.8.

Listing 1. Functions to convert between FP-ANR and binary32 format

```
#include <ieee754.h>

// Convert a binary32 number f
// to a p bits FP-ANR number
float Float2FpAnr(float f, int p){
  union ieee754_float d;
  d.f = f;

  prec = MIN(22, p);

  d.ieee.significand &= (0x7FFFFF<<(23-p));
  // Set the significant flag
  d.ieee.significand |= 1<<(22-p);

  return  d.f;
}

// Convert a p bits FP-ANR number
// to a binary32 number
float FpAnr2Float(float f, int *p){
  union ieee754_float d;
  int c;

  d.f = this->value;

  if (d.ieee.significand!=0){
    c = count_trailing_zeros(d.ieee.significand);
    // Remove the significant flag
    d.ieee.significand ^= 1<<c;
  }

  *p = 22-c;
  return(d.f);
}
```

4.1.2. Operations. We wrote a set of operations over FP-ANR numbers. Listing 2 describes how information on cancellation is propagated during addition and multiplication.

Listing 2. Functions to perform addition and multiplication over FP-ANR format

```
float FpAnrAdd(float a1, float a2){
  float res;
  int e1, e2, er;
  int p1, p2;

  res = FpAnr2Float(a1, &p1) + FpAnr2Float(a2, &p2);

  frexp(a1, &e1);
  frexp(a2, &e2);
  frexp(res, &er);

  return Float2FpAnr(res , er-MAX((e1-p1),(e2-p2)));
}

float FpAnrMul(float a1, float a2){
  float res;
  int p1, p2;

  res = FpAnr2Float(a1, &p1) * FpAnr2Float(a2, &p2);

  return Float2FpAnr(res , MIN(p1,p2));
}
```

One can notice that this simplified version implements truncation as the rounding mode. There are two solutions to implement other roundings. The first and easiest solution consists of adding a given quantity to the mantissa followed by a truncation. However, this solution is subject to the double rounding problem [19]. The second solution consists of allowing the hardware to perform the rounding at the right position in the mantissa. It can be done by right shifting the mantissa so that the least significant bit of the

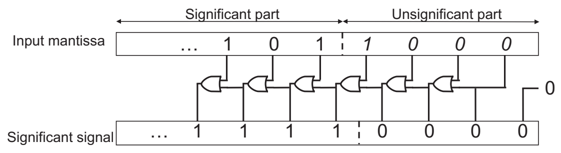

Figure 1. Generation of the significant flag from a mantissa in FP-ANR format based on a tree of OR gate.

significant part of the FP-ANR format corresponds with the least significant bit of the mantissa of the IEEE Standard 754 representation format. Providing those rounding modes is achieved by adding extra shifting instructions.

4.2. Hardware implementation

Hardware implementation of the FP-ANR is more straightforward and simpler than the software solution. As the FP-ANR and the IEEE Standard 754 format are similar, FP-ANR can rely on the existing IEEE Standard 754 hardware implementation. The only difference is the introduction of the necessary hardware to manage the position of the *significant flag* and rounding. This requires the introduction of a trailing zero count at the input and mantissa shifters for the rounding. This operation can be done with a priority enforcer/encoder corresponding to a chain of elements with a ripple signal, scanning bit of the mantissa from right to left. The ripple signal signifies that "nothing before it" is valid and it could be replaced with a tree of OR gates to split the mantissa between its significant and insignificant part. This could be done using a carry lookahead implementation. Figure 4.2 exhibits a simple implementation of this operation based on a tree of OR gates.

5. Example

Cancellation can affect the convergence and accuracy of iterative numerical algorithms. As an example, let us consider Archimedes formulae to compute the approximation of π. The iteration is given by:

$$t_0 = \frac{1}{\sqrt{3}} \ , \ t_{i+1} = \frac{\sqrt{t_i^2+1}-1}{t_i} \ , \ \pi \approx 6 \times 2^i \times t_i$$

We have implemented this equation using IEEE-754 double precision and the 64 bits FP-ANR format. Results for iteration up to $i = 27$ are given in Table 3. One can notice that the accuracy of the approximations based on the IEEE-754 format is increasing up to the 13th iteration, and is slowly decreasing till the 26th iteration. Moreover, starting from the 27th iteration, a problematic *Not-a-Number* appears. The rightmost columns of Table 3 are reporting for the FP-ANR format both the value and the number of bits which are considered. This valuable information helps to avoid invalid operation resulting in a NaN.

On this example, one can notice that with the IEEE-754, there is no way to determine when the result is wrong and

Table 3. COMPARISON BETWEEN THE IEEE-754 DOUBLE PRECISION FORMAT AND THE FP-ANR FORMAT FOR THE COMPUTATION OF π DECIMALS USING ARCHIMEDES FORMULAE. BOLD NUMBERS CORRESPOND TO VALID DECIMALS.

Iter.	IEEE-754	FP-ANR (value ; α_R)	
0	**3.**464101615137755e+00	3.464101615137754e+00	52
1	**3.**215390309173475e+00	3.215390309173465e+00	49
2	**3.1**59659942097510e+00	3.159659942097420e+00	47
3	**3.14**6086215131467e+00	3.146086215131277e+00	45
4	**3.14**2714599645573e+00	3.142714599644023e+00	43
5	**3.141**873049979866e+00	3.141873049977221e+00	41
6	**3.141**662747055068e+00	3.141662747046212e+00	39
7	**3.141**610176599522e+00	3.141610176535323e+00	37
8	**3.14159**7034323337e+00	3.141597034060396e+00	35
9	**3.14159**3748816856e+00	3.141593747306615e+00	33
10	**3.141592**927873633e+00	3.141592921689153e+00	31
11	**3.141592**725622592e+00	3.141592703759670e+00	29
12	**3.14159267**1741545e+00	3.141592592000961e+00	27
13	**3.1415926**18900886e+00	3.141592383384705e+00	25
14	**3.14159267**1741545e+00	3.141592025756836e+00	23
15	**3.14159**1935881973e+00	3.141588211059570e+00	21
16	**3.14159267**1741545e+00	3.141571044921875e+00	19
17	**3.1415**81007579364e+00	3.141540527343750e+00	17
18	**3.14159267**1741545e+00	3.141357421875000e+00	15
19	**3.141**406154737622e+00	3.140625000000000e+00	13
20	**3.14**0543492401100e+00	3.136718750000000e+00	11
21	**3.14**0006864690968e+00	3.125000000000000e+00	9
22	**3.1**34945375658852e+00	3.093750000000000e+00	7
23	**3.14**0006864690968e+00	3.000000000000000e+00	5
24	**3.**224515243534819e+00	3.000000000000000e+00	3
25	2.791117213058638e+00	2.000000000000000e+00	1
26	0.000000000000000e+00	ERR(2ˆ1)	0
27	NaN	ERR(2ˆ1)	0

how wrong it is. Whereas with FP-ANR, the number of bit that could potentially be considered valid is known at each iterations and invalid operations can be avoided.

6. Comparisons with Other Methods

6.1. Performance

We have tested the overhead for the addition, multiplication and division of the proposed format compared to hardcoded IEEE Standard 754 operations and CADNA [20] operations on an 2,4 Ghz Intel Core i5, with LLVM version 8.1.0. Results are reported in table 4. These results correspond to the implementation of the prototype library available at http://perso.univ-perp.fr/david.defour/. One can notice that the overhead of the FP-ANR over hardcoded operations range between 8.5 for the multiplication and 21 for the addition. If this overhead is higher than the one of CADNA, we should recall that the FP-ANR is intended to be implemented in hardware and therefore available at no cost.

6.2. Comparison with Unum

Recently [3], Gustafson proposed a modified version of significance arithmetic with an extra field (unum field) which indicates if a number is exact. However, according to

Table 4. EXECUTION TIME OF COMMON OPERATIONS IN THE FP-ANR AND THE CADNA FORMAT NORMALIZED WITH THE IEEE STANDARD 754 OPERATIONS.

	double		float	
Operations	FP-ANR	CADNA	FP-ANR	CADNA
Addition	11	7.5	11.3	15
Multiplication	4.7	3.5	3.96	4.0
Division	6.1	5.0	7.8	14.2

William Kahan, the principal architect of IEEE 754-1985, this format presents several drawbacks [5]. Among them, he states:

- The Unum computation does not always deliver correct results.
- The Unums can be expensive in terms of time and power consumption.
- The bit length of Unum format can change during computation, which make its hardware implementation harder than with fixed-size format especially regarding memory allocation, de-allocation and accesses.

The last two points are serious issues that the FP-ANR format does not exhibit. However, the Unum possesses some properties that the FP-ANR does not, such as being able to handle exact numbers.

6.3. Comparison with Stochastic arithmetic

Stochastic arithmetic provides an estimation of the numerical confidence of computed results. The CESTAC method formalizes a simplified version of discrete stochastic arithmetic using randomized rounding for each floating-point operation. This method is implemented using C++ overloaded operators in the CADNA library [7]. This library detects the number of significant digits with a high degree of confidence. It also detects instability such as cancellation, branching instability and mathematical instability. It consists of replacing each floating-point number by a set of 3 floating-point numbers plus an integer, on which stochastic operation are performed. Thanks to those extra fields, such systems provide a tighter bound than the FP-ANR format. However, similarly to the Unum format, those extra fields manipulated with the CADNA hinder memory management and performance.

6.4. Comparison with Monte-Carlo arithmetic

Another alternative to estimate numerical quality of computed result can be achieved by using the Monte-Carlo arithmetic suggested by Parker [21]. Monte-Carlo arithmetic gathers rounding and catastrophic cancellation errors by applying randomization on input and output operands at a given virtual precision. A recent implementation of this solution has been proposed with Verificarlo [6]. Verificarlo implement a LLVM pass which replaces every floating-point operation automatically with the Monte Carlo Arithmetic.

Even though Verificarlo is implemented directly as a compiler pass, which makes it very efficient, the large number of execution samples necessary to collect qualitative results remains a major drawback. The solution proposed by the authors consists of running those numerous execution in parallel. Although this solution reduces the global execution time, it does not reduce the total amount of work to gather this information.

7. Conclusions and Perspectives

New representation formats for floating-point numbers were introduced in the IEEE Standard 754[-2008] revision. This is an attempt to adapt the format to the real need of applications. However, dealing with various formats require a numerical analysis of the program, which is a tedious task that can be solely executed by the expert. Some recent work has been proposed to automate this analysis and/or the benefit of formats changes.

In this article, we have presented a solution that brings the significance arithmetic up-to-date, and makes it compatible with the IEEE Standard 754[-2008]. Significance arithmetic is a concept that adds information on significant digits to each floating-point number. It can provide information on cancellation errors, and if sufficiently accurate, on rounding error. It consists of a representation format with rules for the propagation of error.

The proposed solution is a simple pattern embedded in the mantissa of floating-point numbers. This pattern is self-sufficient and does not require extra fields or memory. This solution presents numerous advantages as it is a simple concept to understand, simple to implement and proves to be memory efficient. Tests on a preliminary version shows that the cost for the detection in software of the proposed pattern is higher compared to other solutions. However, the simplicity of the solution suggests that the performance could be improved using hardware support similar to the management of the rounding modes (e.g. specific instructions or execution flag).

If implemented in hardware, this solution can definitely help developers gain confidence in their code by providing an estimation of the number of significance digits at no cost or help achieve reproducibility.

However, it is not meant to solve all problems related to floating-point arithmetic. Significance arithmetics is similar to the interval arithmetic, produces over-pessimistic bound as results and is unable to solve the loss of correlation between variables. For example, error computation as used in compensated algorithm cannot be evaluated with the significance arithmetic, whereas it works perfectly with the IEEE Standard 754 floating-point arithmetic. For these reasons, we suggest that the FP-ANR format should be used as a complement to the traditional IEEE Standard 754 floating-point arithmetic.

References

[1] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[2] M. Goldstein, "Significance arithmetic on a digital computer," *Commun. ACM*, vol. 6, no. 3, pp. 111–117, Mar. 1963.

[3] J. Gustafson, "The end of numerical error." in *ARITH*, 2015, p. 74.

[4] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Numerical reproducibility for the parallel reduction on multi- and many-core architectures," *Parallel Computing*, vol. 49, pp. 83 – 97, 2015.

[5] W. M. Kahan. (2016, July) A critique of John L. Gustafson's. the end of error — Unum computation and his a radical approach to computation with real numbers. [Online]. Available: http://people.eecs.berkeley.edu/∼wkahan/UnumSORN.pdf

[6] C. Denis, P. de Oliveira Castro, and E. Petit, "Verificarlo: Checking floating point accuracy through monte carlo arithmetic," in *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, 2016, pp. 55–62.

[7] F. Jézéquel and J.-M. Chesneaux, "CADNA: a library for estimating round-off error propagation," *Computer Physics Communications*, vol. 178, no. 12, pp. 933–955, 2008.

[8] J. Denker. (2018, 04) Uncertainty as applied to measurements and calculations. [Online]. Available: http://www.av8n.com/physics/uncertainty.htm

[9] D. Funaro, *Polynomial approximation of differential equations*. Springer Science & Business Media, 2008, vol. 8.

[10] O. Møller, "Quasi double-precision in floating point addition," *BIT Numerical Mathematics*, vol. 5, no. 1, pp. 37–50, 1965.

[11] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[12] H. L. Gray and C. J. Harrison, "Normalized floating-point arithmetic with an index of significance," *Managing Requirements Knowledge, International Workshop on*, vol. 00, p. 244, 1959.

[13] E. A. Bond, "Significant digits in computation with approximate numbers," *The Mathematics Teacher*, vol. 24, no. 4, pp. 208–212, 1931.

[14] J. M. Hyman, "Forsig: an extension of fortran with significance arithmetic," Los Alamos National Lab., NM (USA), Tech. Rep., 1982.

[15] F. Johansson. (2008, 06) Basic implementation of significance arithmetic. [Online]. Available: http://fredrik-j.blogspot.fr/2008/06/basic-implementation-of-significance.html

[16] R. L. Ashenhurst and N. Metropolis, "Unnormalized floating point arithmetic," *J. ACM*, vol. 6, no. 3, pp. 415–428, Jul. 1959.

[17] G. Langdon, "Method and means for tracking digit significance in arithmetic operations executed on decimal computers," Aug. 29 1978, uS Patent 4,110,831.

[18] V. Lefèvre, J.-M. Muller, and A. Tisserand, "Towards correctly rounded transcendentals," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, USA, 1997*. Los Alamitos, CA: IEEE Computer Society Press, 1997.

[19] É. Martin-Dorel, G. Melquiond, and J.-M. Muller, "Some issues related to double rounding," *BIT Numerical Mathematics*, vol. 53, no. 4, pp. 897–924, 2013.

[20] P. Eberhart, J. Brajard, P. Fortin, and F. Jézéquel, "High performance numerical validation using stochastic arithmetic." in *Reliable Computing*, vol. 21, 2015, pp. 35–52.

[21] D. S. Parker, "Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic," Department of Computer Science, University of California, Los Angeles, Los Angeles, CA, USA, Tech. Rep. CSD 970002, 1997.