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

David Defour

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

David Defour*
Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, 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. Among those errors, the loss of significance, or cancellation, 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, unsignificant trailing bit 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-754 standard.**

I. Introduction

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

When numerical noises become dominant, for example during dramatic cancellation, there are no more useful bit of information in the representation numbers. Unfortunately, in this situation there is no way to figure out that such situation occurred by just looking at the representation. This is due to the fact that with the widely used IEEE-754 representation format, there is no way to distinguish useful numerical information from noises. This problem is not new and has been identified and addressed since the late 1950s with significance arithmetic [2]. Significance arithmetic addressed this issues by tailoring the number of digits to the needs.

Significance arithmetics is gaining in popularity again directly through unum [3] or indirectly through numerous problem encountered with exascale computer and the lack of confidence in numerical results [4]. If 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 side, indirect solutions based on software solution to detect cancellation [6], [7], or avoid rounding error [4] are not meant to be efficient and for real time execution.

In this article, we propose a new way to keep track of unsignificant information. The solution consists in an altered IEEE-754 representation format of the mantissa. That information is stored using a simple pattern that replace unsignificant digits. This makes such representation numbers almost as accurate as original IEEE-754 numbers. In addition, operations on that format can be easily implemented in hardware at no cost. Therefore, the proposed solution corresponds to an efficient implementation of significance arithmetic that is simple, efficient and compatible with IEEE-754 format.

This article is organized as follows: Section II gives some background on rounding error management. Section III details the proposed format named FP-ANR to embed cancellation information within the representation format of floating-point numbers. Section IV presents how to implement the proposed approach both in software and hardware. Then, Section V presents some experimental results and comparison with others solutions, before concluding in Section VI.

II. Preliminaries

Floating-point numbers consist in approximations of real numbers. Associated with the concept of approximation is the concept of errors. Therefore, digits of a floating-point representation numbers can be split in two parts; a significant and an unsignificant part. To make the paper self-complete, we recall in this section some background on IEEE-754 floating-point arithmetic, errors and significance arithmetic.

A. The IEEE-754 standard

The current version of the floating-point standard, the IEEE 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-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

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

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

In 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 the explicit 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-754 double precision format (or binary64) are represented with 64 bits which are split in 1 sign bit, $p = 52$ bits of mantissa and $e = 11$ bits of exponent whereas single precision format (or binary32) are represented with 32 bits split in 1 sign bit, $p = 23$ bit of mantissa and $e = 8$ bits of exponent.

B. Floating-Point Errors

Representation format of floating-point numbers differs by their radix and the number of bits used for the encoding. The 2008 revision of the IEEE-754 includes new formats to better adapt to the real need of the computation ranging from 16 to 128 bits. However, those formats rely on fixed numbers of bit, which implies that it does not exactly matches the real needs. Therefore, numbers must be rounded or filled with zeros in the least significand digits of the mantissa when the used format is respectively undersized or oversized. This means that by constructions FP numbers embed errors in their representation. These errors can be separated into three groups: data uncertainty, rounding and cancellation.

1) Uncertainty: Uncertainty in data are linked with initial input values, especially when data come from measurements or experimentations using physical sensor. It may also come from the model or the algorithm used to model real phenomena such as polynomial approximation [8].

For example, due to uncertainty, data produced by industrial measurement are accurate to a few digits (thermal sensor [9], voltage sensor [10]). This uncertainty can be given in percentage such as in [11]. This mean that for example the twenty digits measure $x = 12345.678901234567890$ obtained with a process exhibiting an uncertainty of $U = 10^{-3}\%$ correspond 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 corresponding solely to noise or unsignificant digits. Due to the lack of knowledge, or simply because floating-point number are over-dimensioned, this noise is kept in manipulated numbers during computation. However, those extra digits do not bring any benefit for the numerical quality of the final results.

2) Rounding: Because floating-point numbers have a limited number of digits, there cannot represent real numbers accurately. When there are more digits than the format allows, the number is rounded and the leftover ones are omitted.

Let $fl()$ denote the result of a floating-point computation, which has to be rounded according to the relative rounding error u. We have

$$fl(1 + u) = 1$$

with u that depends on the radix b and the precision p as follow:

$$u = b/2 \cdot b^{-p}$$

Let $\mathbb{F} \in \mathbb{R}$ be the set of floating-point numbers, if $x \in \mathbb{R}$ belongs to the range of representable floating-point numbers, then

$$fl(x) = x \cdot (1 + \epsilon) \text{ with } |\epsilon| < u$$

Floating-point operations in IEEE-754 satisfy

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

The standard defines five rounding rules, two rounding to nearest (ties to even, ties away from zero) and three directed rounding (toward $0, -\infty, +\infty$).

3) Cancellation: Cancellation occurs when two nearby quantities are subtracted such that the most significant digits cancel each other. Cancellations are very common but when many digits are lost, the effect can be severed as the number of informative digits is reduced. In that case, it corresponds to catastrophic cancellation as the impact on the sequel of the computation can be dramatic.

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 digit as the real result should be 1.5. This is due to the catastrophic cancellation which occurred during the subtraction. Such cancellations cannot be detected, meaning there is no way to know that $r = 0.0$ was completely incorrect, except through dedicated sequence of operations.

Such sequence are used for example in numerical algorithms that computes the error such as the 2sum algorithm [12], [13]. This corresponds to specific pattern of computation with correlation between variables.

C. Significance arithmetic

Significance arithmetic [2], [14], [15] 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 unsignificant digits.

Definition 2 (Significant and Unsignificant digits)**.** *Let α be the number of significant digits of a p-digits number X represented in radix b. Then, the error e in X is such that $|e| \leq X \cdot b^{-\alpha}$ and the number of unsignificant digits is $p - \alpha$.*

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 provide 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.

With the normalized method, as many digits as possible of a number are retained and an added index defines the

number of significant digits. Such arithmetic is implemented in software using set of numbers in FORTRAN with FORSIG [16], or Python [17]. With the unnormalized method [18], only digits considered significant are retained.

The integration of specific pattern in the mantissa to categorize significant and unsignificant digit has already been proposed for decimal computer in BCD format [19]. It relies upon unused bit pattern in BCD format which are bit-field 1010 and 1011 corresponding to respectively digits 10 and 11. More recently, Gustafson 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 [3].

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 such as Newton's algorithm.

III. A FORMAT TO EMBED CANCELLATION INFORMATION

In this section we described the proposed representation format named *Floating-Point Adaptive Noise Reduction* (*FP-ANR*) as it allows the user to distinguish significant from unsignificant digit. Unsignificant digit, or noise, can come from initial uncertainty, or cancellation generated during computation. This format corresponds to an implementation of significance arithmetics which can be easily done in hardware or software based on existings IEEE-754 format. To simplify the article, in the sequel we will consider radix-2 arithmetic, where bit or digit will refer to the same notion.

A. The representation format

Our goal is to propose a non-intrusive solution while being able to keep track of uncertainty and 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 to embed cancellation information. The modification of the mantissa consists in replacing uninformative bit, for example bit of noise lost during cancellation, by a given pattern. This pattern must be self-detectable to avoid using extra fields as in Unum. There are two candidates for such pattern. We can use a 0 followed by as many 1 as needed (or a 1 followed by as many 0 as needed). With such patterns, one can easily deduce the number of cancelled bit by scanning from right to left the mantissa to detect the first 1 (or 0 respectively). The assembly instruction that perform this operation is usually named *Count Trailing Zero/One*. For the sake of simplicity, in the sequel of this article we will focus on the pattern made of a 1 followed by 0. We call that first 1 encountered from right to left in the mantissa the *significant flag*.

With FP-ANR, one bit of the mantissa is used to represent the *significant flag*. It means that a p-bit mantissa number will have at most $p-1$ informative bits which is 1 bit less than the corresponding IEEE-754 representation format which FP-ANR is built upon. For example, the value 1.0 which corresponds to the binary32 IEEE-754 representation number 0 01111111 00000000000000000000000 will be represented in the FP-ANR format by

0 01111111 0000000000000000000001

The rightmost bit equal to 1 and corresponding to the *significant flag*, indicates the position between significant and unsignificant 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 00000000000*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-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.

In other words, the major difference between the IEEE-754 representation format and the FP-ANR format is that IEEE-754 can manipulate exact values such as 1.0 whereas FP-ANR deals only with approximation (except for 0). This could be a drawback as discussed in section II-C, but we believe that exact values have to be handled with fixed-point arithmetic which are meant for that purpose. On the contrary, floating-point values are by essence finite representation and therefore approximation of real number and should integrate that information.

B. Managing uncertainties

With FP-ANR, uncertainty is integrated directly in the mantissa. For example, let consider a physical process which produces a value 1234.56 with an uncertainty $U = 10^{-3}\%$. This measure will be converted in:

binary32	0 10001001 00110100101000111101100
FP-ANR	0 10001001 00110100101001000000000

As we can observe, with the IEEE-754 format the value will be translated directly in its binary format where the last 10 unsignificant bits correspond to noise. Whereas with FP-ANR we can distinguish significant and unsignificant bits.

To the extreme, when all bits of information are lost we can keep track of that information, which is not the case with other representation format. It means that FP-ANR offers a new concept over other representation format including IEEE-754 which is the concept of error.

For example, let 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. It is therefore difficult to produce valid bit out of such numbers, except that there is another useful embedded information. It 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 order of the error (unsignificant bit). It can potentially avoid a division by zero resulting from an unwanted catastrophic cancellation where all bits are lost.

C. Addition of FP-ANR

As we have seen in section II-B3, least significant bit of the mantissa are usually uninformative as they corresponds to noise due to cancelation or discretization. The information on unsignificant bit has to be propagated during operations. This can be done by updating the position of the *significant flag* of 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.

D. Multiplication and division of FP-ANR

Error propagation during multiplication corresponds to the simplest case. The number of significant bits resulting from a multiplication between two FP-ANR numbers is computed as follow:

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 error e_X in X is such that $X = x \cdot (1 + e_X)$ with $|e_X| \leq 2^{-\alpha_X}$.

The error in the multiplication $R = A \cdot B$ is computed as follow:

$$R = (a \cdot b) \cdot (1 + e_a + e_b + e_a \cdot e_b)$$

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 number of significant bits in the results R is approximated using

$$\alpha_R = MIN(\alpha_A, \alpha_B) \tag{1}$$

The number of significant bits in the results $R = A/B$ for the division can be computed as follow:

$$R = \frac{a}{b} \cdot \frac{1 + e_a}{1 + e_b}$$

This formula can be rewritten by expressing the denominator term for the error as an infinite series as follow:

$$R = \frac{a}{b} \cdot (1 + e_a) \cdot (1 - e_b + e_b^2 + ...)$$

Since the error e_b is assumed 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

$$|e_r| \leq 2^{-\alpha_A} + 2^{-\alpha_B} + 2^{-\alpha_A - \alpha_B}$$

and the number of significant bits in the results R of the division can be approximated using Equation 1 as well.

E. Other operations in FP-ANR

To be complete, we must consider the propagation of uncertainty in the case of more complex operation such as exponential, logarithms or trigonometric function. 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 digit is determined similarly as the propagation of uncertainty. It can be done by looking at the extremum on the interval of values corresponding to the initial uncertainty interval $[X - e_X; X + e_X]$ with e_X the error in X such that $|e_X| \leq X \cdot 2^{-\alpha_X}$.

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

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

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

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

This estimation is valid only if the function is considered quasi-linear and quasi-gaussian on the interval $[X - e_X; X + e_X]$. This corresponds to an estimation of the number of significant bit of the result α_R:

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

Combining this formulae with the following estimation of the number of significant digit in x:

$$\alpha_X = log_2 \left| \frac{X}{e_X} \right|$$

we get

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

where for any number y, $log_2(y)$ can be approximated using the exponent part of its floating-point representation format. In particular:

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

F. Rounding in FP-ANR

We should mention that the presence of the *significant flag* is independent of the rounding problem. Therefore, we propose to use similar rounding strategy with FP-ANR than with IEEE-754 format. The only difference is 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 IEEE-754 representation format.

G. 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 [20]. Table Maker's Dilemma corresponds to the problem of computing approximation of elementary functions with enough bit to ensure correct rounding. This problem is known to be difficult with IEEE-754 representation format since there are no bound on the number of bit mandatory for every function and every format.

With FP-ANR, the Table Maker's Dilemma is circumvented as follow. One can set a target accuracy t function of the number of significant bit 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 not, which corresponds to a hard to round case, it means that we are not sure of the last bit in the significant part. This corresponds to an uncertainty due to the Table Maker's Dilemma, and this uncertainty can be integrated in the FP-ANR format by left-shifting by one position the *significant flag*. This way reproducibility and portability of results provided by correct rounding is preserved.

H. Interaction between FP-ANR and IEEE-754

One major advantage of FP-ANR is that it is compatible with IEEE-754 representation format. As with any formats, 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-754 format, this can be done by replacing the *significant flag* with a 0. From IEEE-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 IEEE-754 operators can process FP-ANR numbers. It won't lead to crash or irrelevant results, only the meaning of the unsignificant bits will differs. However, it should not be considered as a serious issue as those bits correspond to noise. The opposite, which corresponds to the case where FP-ANR operators process IEEE-754 numbers is more problematic, as this depends on the position of the last bit set to 1.

IV. Implementations

A. Software implementation

In this section, we describe a simplified software emulation of the proposed format. For the sake of simplicity, we will only describe basic operations on FP-ANR format related to the IEEE-754 binary32 format.

We should mention that we have developed two C++ class to deal with single and double precision format. These two class are based on the header file of the CADNA library [7], where we replaced the code related to stochastic arithmetic with operations on significance arithmetic. This library can advantageously replace IEEE-754 double and float format and major operations over those formats. It is available for download at http://perso.univ-perp.fr/david.defour/

One can notice that the biggest advantage of FP-ANR over other solutions that requires 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 of this article and is kept for future work.

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

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

```
#include <ieee754.h>

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

  prec = MIN(22, p);

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

  return  d.f;
}

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

  d.f = this->value;

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

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

2) Operations: We wrote a set of operations over FP-ANR numbers. Listing 2 describes how information on cancellation gets 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);
```

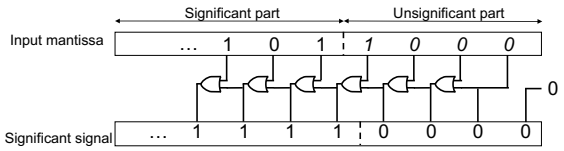

Figure 1. Generation of the significant flag from a mantissa in FP-ANR format based on a tree of OR gate.

```
  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 implement truncation as rounding mode. There are two solutions to implement other rounding. The first and easiest solution consists in adding a given quantity to the mantissa followed by a truncation. However, this solution is subject to the double rounding problem [21]. The second solution consists in letting the hardware perform the rounding at the right position in the mantissa. It can be done by shifting the mantissa by some quantity so that the least significant bit of the significant part of the FP-ANR format corresponds with the least significant bit of the mantissa of the IEEE-754 representation format. Providing those rounding modes just consists in adding extra shifting instructions. Therefore, there are kept out of this article.

B. Hardware implementation

Hardware implementation of FP-ANR is more straightforward and simpler than the software solution. As FP-ANR and IEEE-754 format are similar, FP-ANR can rely on existing IEEE-754 hardware implementation. The only difference is the introduction of the necessary hardware to manage the position of the *significant flag*. This position dictates the position of rounding. It can be determined by implementing in hardware a trailing zero count operation. 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 means "nobody before it" is valid and it could be replaced with a tree of OR gates to split mantissa between its significant and unsignificant part. This could be done using a carry lookahead implementation. Figure IV-B exhibit a simple implementation of this operation based on a tree of OR gates.

V. Comparisons with Other Methods

A. Performance

We have tested the overhead for the addition, multiplication and division of the proposed format compared to hardcoded IEEE-754 operations and CADNA [22] operations on an 2,4 Ghz Intel Core i5, with LLVM version 8.1.0.

	double		float	
Operations	FP-ANR	CADNA	FP-ANR	CADNA
Addition	21	7.5	18.5	15
Multiplication	8.7	3.5	8.5	4.0
Division	8.9	5.0	15	14.2

Table I. Execution time of common operations in FP-ANR and CADNA format normalized with IEEE-754 operations.

Results are reported in table I. Those results corresponds to the implementation of the prototype library available at http://perso.univ-perp.fr/david.defour/. One can notice that the overhead of 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 FP-ANR is intended to be implemented in hardware and therefore available at no cost.

B. Comparison with Unum

Recently, Gustafson proposed in [3] a modified version of significance arithmetic with an extra field (unum field) which indicate if a number is exact. However, according to William Kahan, the principal architect of IEEE 754-1985, this format presents several drawbacks [5]. Among them, he is stating that:

- 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 FP-ANR does not exhibits.However, Unum has some properties that FP-ANR don't, such as being able to handle exact numbers.

C. Comparison with Stochastic arithmetic

Stochastic arithmetic provides an estimation of the numerical confidence of computed results. CESTAC method formalizes a simplified version of discrete stochastic arithmetic using randomized rounding for each floating-point operation. This method is implemented using C++ operator overloading in CADNA library [7]. This library detects with a high degree of confidence the number of significant digits, and instability such as cancellation, branching instability and mathematical instability. It consists in 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 system provides tighter bound than FP-ANR. However, similarly to the Unum format, those extra fields manipulated with CADNA hinder memory management and performance.

D. Comparison with Monte-Carlo arithmetic

Another alternative to estimate numerical quality of computed result consists in using Monte-Carlo arithmetic as suggested by Parker [23]. 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 an LLVM pass which replace every floating-point operation to automatically use Monte Carlo Arithmetic.

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

VI. Conclusions and Perspectives

IEEE-754 2008 revision have seen the introduction of new format which reflect the trend to better adapt the representation format used in software to the real need of the application. However, dealing with various format requires to bound at each step numerical quality of results, which is a tedious task that can be done only by the expert. Some recent work has been proposed to automate the estimation of the numerical quality of computed results produced by software and/or the benefit of formats changes.

In this article, we have presented a solution that bring up to date significance arithmetic and make it compatible with the IEEE-754 standard. Significance arithmetic is a concept that add information on significant digits on each floating-point number. It can provide information on cancellation errors, and if accurate enough, on rounding error. It consists in a representation format with rules for the propagation of error.

The proposed solution consists in a simple pattern embedded in the mantissa of floating-point numbers. This pattern is self-sufficient and does not requires extra fields or memory. This solution presents numerous advantages as it is a simple concept to understand, simple to implement and memory efficient. Tests on a preliminary version shows that the cost for the detection in software of the proposed pattern is high compared to other solution. However, the simplicity of this solution suggests that performance could be improved thanks to hardware support. Support for FP-ANR can be achieve through specific instructions or execution flag similarly to the management of rounding mode.

If implemented in hardware, this solution can definitely help developers gain confidence in their code by providing an estimation on 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 suffer from the same problem of interval arithmetic such as loss of correlation between variables, and produces over pessimistic bound as results. For example, iterative scheme such as newton iteration works perfectly with IEEE-754 floating point arithmetic which is not the case with significance arithmetic. That is why we advocate that FP-ANR should only come in support to traditional IEEE-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. [Online]. Available: http://doi.acm.org/10.1145/366274.366339

[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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115001155

[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. [Online]. Available: http://dx.doi.org/10.1109/ARITH.2016.31

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

[9] T. Huynh, "Fundamentals of thermal sensors," in *Thermal Sensors*. Springer, 2015, pp. 5–42.

[10] N. Femia, G. Petrone, G. Spagnuolo, and M. Vitelli, *Power electronics and control techniques for maximum energy harvesting in photovoltaic systems*. CRC press, 2012.

[11] Fluke, *Understanding specifications for precision multimeters*, 12 2006. [Online]. Available: http://support.fluke.com/calibration-sales/Download/Asset/2547797_6200_ENG_A_W.PDF

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

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

[14] C. H. Jr. and H. L. Gray, "Normalized floating-point arithmetic with an index of significance," *Managing Requirements Knowledge, International Workshop on*, vol. 00, p. 244, 1899.

[15] E. A. Bond, "Significant digits in computation with approximate numbers," *The Mathematics Teacher*, vol. 24, no. 4, pp. 208–212, 1931. [Online]. Available: http://www.jstor.org/stable/27951340

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

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

[18] R. L. Ashenhurst and N. Metropolis, "Unnormalized floating point arithmetic," *J. ACM*, vol. 6, no. 3, pp. 415–428, Jul. 1959. [Online]. Available: http://doi.acm.org/10.1145/320986.320996

[19] G. Langdon, "Method and means for tracking digit significance in arithmetic operations executed on decimal computers," Aug. 29 1978, uS Patent 4,110,831. [Online]. Available: https://www.google.com/patents/US4110831

[20] 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.

[21] 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.

[22] 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.

[23] 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. [Online]. Available: http://www.cs.ucla.edu/~stott/mca/CSD-970002.ps.gz