Sipe: a Mini-Library for Very Low Precision Computations with Correct Rounding

Vincent Lefèvre

To cite this version:

Vincent Lefèvre. Sipe: a Mini-Library for Very Low Precision Computations with Correct Rounding. 2013. <hal-00864580>

HAL Id: hal-00864580

https://hal.inria.fr/hal-00864580

Submitted on 22 Sep 2013

SIPE: a Mini-Library for Very Low Precision Computations with Correct Rounding

Vincent Lefèvre INRIA, LIP / CNRS / ENS Lyon / Université de Lyon
Lyon, France
Email: vincent@vinc17.net

***Abstract*—SIPE is a mini-library in the form of a C header file, to perform radix-2 floating-point computations in very low precisions with correct rounding, either to nearest or toward zero. The goal of such a tool is to do proofs of algorithms/properties or computations of tight error bounds in these precisions by exhaustive tests, in order to try to generalize them to higher precisions. The currently supported operations are addition, subtraction, multiplication (possibly with the error term), fused multiply-add/subtract (FMA/FMS), and miscellaneous comparisons and conversions. SIPE provides two implementations of these operations, with the same API and the same behavior: one based on integer arithmetic, and a new one based on floating-point arithmetic. Timing comparisons have been done with hardware IEEE-754 floating point and with GNU MPFR.**

***Index Terms*—low precision; arithmetic operations; correct rounding;**

I. INTRODUCTION

Numerical calculations on computers are most often done in floating-point arithmetic, as specified by the IEEE 754 standard, first published in 1985 [1] and revised in 2008 [2].

This standard first defines the floating-point formats. Given a radix β and a precision p, a finite floating-point number x has the form:

$$x = s \cdot m \cdot \beta^e$$

where $s = \pm 1$ is the *sign*, $m = x_0.x_1x_2 \ldots x_{p-1}$ (with $0 \leq x_i \leq \beta - 1$) is a p-digit radix β fixed-point number called the *significand*, and e is a bounded integer called the *exponent*. If x is non-zero, one can require that $x_0 \neq 0$, except when this would make the exponent smaller than the minimum exponent[1]. If x has the mathematical value zero, the sign s matters in the floating-point format, but s has a visible effect only for particular operations, like $1/0$. As this paper will not consider such operations and we will focus on the values from $\mathbb{R}$ represented by the floating-point numbers, we will disregard the sign of zero.

Most hardware floating-point implementations use binary formats ($\beta = 2$), as specified by the first IEEE 754 standard in 1985. So, for the sake of simplicity, we will assume $\beta = 2$. But future work may consider $\beta = 10$ (as decimal formats have been introduced in the IEEE 754-2008 revision), and possibly other radices.

The IEEE 754 standard also specifies that the result of an operation done in a supported floating-point format be *correctly rounded* according to one of the *rounding-direction attributes* [2, §4.3] (a.k.a. *rounding modes*). The most common one, and the default one for binary formats [2, §4.3.3], is round-to-nearest with the *even-rounding rule* if the exact value to be rounded is the middle of two consecutive machine numbers; it is called roundTiesToEven in IEEE 754-2008. Various floating-point algorithms were designed for it in particular. For these reasons, we focused on this rounding mode in SIPE. However some support for the round-toward-zero mode has recently been added: the value is rounded to the closest machine number in the direction of the value zero, i.e. the exact significand is truncated to the precision of the floating-point system.

The most common and most often implemented binary formats are the two formats entirely specified by the initial IEEE 754 standard:

- binary32, a.k.a. single precision: precision $p = 24$;
- binary64, a.k.a. double precision: precision $p = 53$.

But for the following reasons, one may want to perform computations in much lower precisions than 24 bits:

- One purpose is to perform exhaustive tests of algorithms (such as determining the exact error bound in the floating-point system). Since the number of possible values per input is proportional to 2^p, such tests will be much faster with small values of p and may still be significant to deduce or conjecture results for larger values of p, such as the usual precisions $p = 24$ and $p = 53$.
- Similar tests can be done to get a computer proof specific to these precisions, where larger precisions can be handled in a different way. This is what was done to prove that the TwoSum algorithm in radix 2 is minimal among algorithms only based on additions and subtractions in the round-to-nearest mode. [3], [4][2]

For this purpose, it is absolutely necessary to have correct rounding in the target floating-point system. Only one library was known to provide it in non-standard precisions: GNU MPFR [5], which guarantees correct rounding in any precision larger than or equal to 2, in particular the small precisions mentioned above. However the main goals of MPFR are efficiency in large precision and full specification as in the IEEE 754 standard (e.g. support of special numbers and exceptions)[3], while our main concern here is the performance

[1]Such numbers that must have $x_0 = 0$ are called *subnormals*, but we will ignore them in this paper, as they do not often occur in computations, and if they do, they need specific attention in the algorithms, the proofs and so on.

[2]Only [3] has the complete proof.

[3]MPFR has also been optimized to be efficient in low precision, but the overhead due to its generic precision and full specification cannot currently be avoided.

in a low precision, which may be fixed at compile time for even more efficiency. That is why the SIPE library, presented in this paper, has been written.

Let us also mention GCC's *sreal* internal library (`sreal.c` and `sreal.h` files in the GCC source), which provides a similar arithmetic; but because this library was written for another purpose, there are major differences:

- *sreal* does not support negative numbers;
- with *sreal*, the rounding-direction attribute corresponds to roundTiesToAway (rounding to nearest, halfway cases being rounded away from zero), while SIPE needed at least the support of the conventional round-to-nearest mode with the even-rounding rule;
- with *sreal*, the precision is more or less hard-coded;
- *sreal* detects the overflows and returns the maximum floating-point number in such a case, while overflow detection is not necessary and would lower the performance in the context of SIPE (see Section II);
- contrary to SIPE, *sreal* supports division, but not FMA;
- the *sreal* library does not seem to be very optimized.

Section II presents the basic choices to implement SIPE, which lead to two different implementations. Section III describes some of the algorithms used in the initial implementation, based on integer arithmetic. Section IV presents a newer implementation, based on native floating-point arithmetic (usually implemented mostly in hardware). We give results and timings in Section V and conclude in Section VI.

II. Basic Choices

Let us recall the criteria we want to focus on:

- The SIPE library must implement a binary low-precision floating-point arithmetic. A datum in this system will be called a SIPE number.
- The results must be correctly rounded in the chosen system. We need to support the roundTiesToEven rounding mode. Other rounding modes may be supported as an option; actually, after the first versions of SIPE were written, there has been a request for roundTowardZero.
- The library must be as fast as possible, since it may be used for exhaustive tests on a huge number of inputs.
- We only need to deal with finite numbers, representing real values, i.e. we do not need to consider special numbers (NaN, infinities, the sign of zero) and exceptions from the IEEE 754 standard. It is up to the user of the library to make sure that underflows and overflows cannot occur, e.g. with a proof or by adding tests[4]; since the only available operations are currently based on addition, subtraction and multiplication, and since the exponent range that will be implied by the representation is very large, this is not even a problem in practice. Moreover, concerning the other IEEE 754 exceptions, division by zero is impossible, and all the operations are mathematically valid (but this may change if other operations are implemented in the future).

For portability and performance, the library is written in C (with the generated assembly code in mind, when designing the algorithms), and for the implementation based on integer arithmetic, we require GCC or a compatible compiler in order to benefit from better semantics of the bitwise shift operators on negative values and from some GCC extensions. However it would still be possible to write a more portable version if need be, at least for testing purpose. We did not want to include assembler, which would depend on the processor; instead we rely on the compiler optimizations. More will be said about this later, but first, let us describe how the precisions are handled and how SIPE numbers are encoded.

Contrary to GNU MPFR, where each MPFR object (a multiple-precision floating-point number) has its own precision and operations between several objects (input and output numbers) can mix different precisions, the precision is here assumed to be common to each number. For performance reasons, SIPE does not check that the user follows this requirement (an assertion mechanism, where assertion checking could be enabled or disabled, could be added in the future) and the precision is not encoded in the numbers. Allowing one to mix precisions could also be considered in the future (without degrading the performance of the case of a common precision). The precision is passed as an argument to each function, but since these functions are declared as inline, if the precision is known at compile time, then the compiler will be able to generate code that should be as fast as if the precision were hard-coded.

Concerning the encoding of SIPE numbers, several possibilities had initially been considered:

- A structure consisting of two native signed integers (typically corresponding to registers of the processor): an integer M representing a signed significand and an integer E representing an exponent. This is what had been chosen for the first version of SIPE (which had only this integer based implementation), and this gave the name of the library: *Small Integer Plus Exponent* (SIPE), inspired by the name DPE[5] (meaning *Double Plus Exponent*). Though the integer M can hold values allowing one to represent numbers for up to precision $p = 32$ or 64 in practice, the algorithms described in Section III are valid only for much smaller values of p; the maximum allowed value of p will depend on these algorithms.
- The same representation by a (significand,exponent) pair, but packed in a single integer. This could have been possible, even with 32-bit integers, since the precision is low and the exponent range does not need to be very wide here. However such a choice would have required splittings, with potential portability problems in C related to signed integers. It could be interesting to try, though. The choice that has been done here in the integer version of SIPE is closer to the semantics (with no hacks). Anyway one cannot really control what the compiler will do, so that the performance greatly depends on the C

[4]In the SIPE implementation, such exceptions will correspond to integer overflows in the integer based version, and to floating-point overflows and underflows in the floating-point based version. Thus there may be some detection support from the language implementation and/or the processor, e.g. in a LIA-1 [6] context for the integer based version and in a IEEE 754 context for the floating-point based version.

[5]https://gforge.inria.fr/projects/dpe/

implementation; this could be observed just by changing the compiler version, see Section V.

- A separate significand sign. This would have made the rounding code (a macro) simpler, but the arithmetic operation code more complex, with more memory transfers.
- An integer representing the value scaled by a fixed power of two, i.e. a fixed-point representation (but let us recall that we still want the semantics of a floating-point system). The exponent range would have been too limited, and such an encoding would have also been unpractical with correct rounding.
- A native floating-point format, e.g. via the `float` or `double` C type. Such a choice was initially thought to be impractical due to the well-known *double-rounding problem*, but in our context, such problems do not occur for most operations implemented in SIPE. This will be detailed in Section IV.

Still for performance reasons, SIPE is not implemented as a usual library. Like with DPE, only a C header file is provided, consisting of inline function definitions. Some of these functions are described in the following sections (III and IV).

Most of the integer based implementation with rounding to nearest was written in April/May 2008. Some functions were added in November 2009 in the context of [4] (though MPFR was initially used for the tests performed for this paper), and several bugs were fixed in 2011 and 2012 (in addition to minor changes); the first article on SIPE [7] concerns this version. Both the floating-point based implementation and the support for round-toward-zero in the integer based implementation were added in 2013; a few comparisons of generated x86_64 code have been done to check that this support did not introduce any regression.

III. IMPLEMENTATION: INTEGER VERSION

This was the original SIPE implementation. First let us say a bit more about the representation of SIPE numbers.

There exist several conventions to define the (significand,exponent) pair. The usual one was given at the beginning of Section I, where the component M would represent a p-bit fixed-point number. But since M is an integer, the following convention is better here: we can define

$$x = M \cdot \beta^E$$

where M is an integer such that $|M| < \beta^p$, and E (denoted q in the IEEE 754-2008 standard [2, §3.3]) is a bounded integer, respectively called *integral significand* and *quantum exponent* in [8]. One has: $E = e - p + 1$. If $x \neq 0$, we require its representation to be *normalized*, i.e. $\beta^{p-1} \leq |M| \leq \beta^p - 1$. The value β^E is the *ulp* (Unit in the Last Place)[6] of x. The benefit of normalization in SIPE will be discussed in Section III-E.

Moreover, for $x = 0$, we necessarily have $M = 0$ and the value of the exponent E does not matter. But we will require E to be 0 in order to avoid undefined behavior due to potential integer overflow in some cases, in particular with the multiplication, as said later; other values for E could have been chosen (not the intuitive minimum value representable in the type of E, though, since adding two such values would directly trigger an integer overflow), but 0 happens to be the most practical value in the C code. Even though the results of an integer overflow would not really be used, the undefined behavior could have unwanted side effects in practice: an integer overflow may generate an exception or the code may be transformed in an uncontrolled manner by the compiler, due to optimizations based on the fact that undefined behavior is forbidden.

The main idea behind the implementation is that there are three classes of operations, possibly depending on the order of magnitude of the inputs:

1) simple operations that can be performed exactly in a straight way, without the need to take care of the precision and the need to round the result;
2) operations that can be performed exactly (or "almost" exactly, as in Section III-C) thanks to higher internal precision (the bit-width of the integer variables being larger than the maximum allowed precision of the system), and whose result needs to be rounded;
3) operations that would need too much internal precision for an exact computation, but whose result can easily be deduced from the sign and the exponent of the inputs.

A. Addition and Subtraction

We consider the addition or subtraction of two SIPE numbers x and y: $x \pm y$. Let $\delta = E_x - E_y$ when $x \neq 0$ and $y \neq 0$. Let $\nu = 1$ in the round-to-nearest mode, $\nu = 0$ in the round-toward-zero mode; `SIPE_NEAREST` is ν in Code 1, and `SIPE_TRUNC` is the complement $1 - \nu$. The algorithm of the addition (`sipe_add`) and subtraction (`sipe_sub`) operations distinguishes several cases, taken in the following order:

1) If $x = 0$, we return y for addition, $-y$ for subtraction. This corresponds to class 1.
2) If $y = 0$, we return x. This corresponds to class 1.
3) If $\delta > p + \nu$ (corresponding to class 3), then $|y|$ is so small compared to $|x|$ that $x \pm y$ rounds to x in the round-to-nearest mode; thus one returns x in this case. Ditto in the round-toward-zero mode, except that one may need to return the next SIPE number in the direction of zero, depending on the operation and on the sign of the inputs. Indeed, $|y| = |M_y| \cdot 2^{E_y} < 2^{E_y+p} \leq 2^{E_x-\nu-1}$, and if E_r denotes the exponent of the exact result $r = x \pm y$, then $|E_r - E_x| \leq 1$, so that $|r - x| = |y| < 2^{E_r-\nu} = 2^{-\nu}\,\mathrm{ulp}(r)$, because x is normalized.
4) If $-\delta > p + \nu$ (corresponding to class 3), then $|x|$ is so small compared to $|y|$ that $x \pm y$ rounds to $\pm y$ or the next SIPE number in the direction of zero. The explanations are the same as above.
5) Otherwise $|\delta| \leq p+\nu \leq p+1$, so that with a requirement on the precision $p \leq \lfloor (S-2)/2 \rfloor$, S being the bit-width of the type `sipe_int_t` of the significand M, we can

[6]In the IEEE 754-2008 standard, it is called *quantum*, which has a more general definition for numbers that are not normalized. So, we prefer here the conventional term *ulp*.

compute $x \pm y$ exactly without an integer overflow by $(M_x \pm M_y \cdot 2^{-\delta}, E_x)$ or $(M_x \cdot 2^{\delta} \pm M_y, E_y)$ depending on the sign of δ, then round and normalize the result (see Section III-E). This corresponds to class 2.

The code for both functions `sipe_add` and `sipe_sub` is implemented via a single macro, which is invoked twice for the actual function definitions. It is given as an example (Code 1).

Code 1 Code of addition and subtraction, slightly edited for this paper.

```
#define SIPE_DEFADDSUB(OP,ADD,OPS,CN,CP)      \
  static inline sipe_t                        \
  sipe_##OP (sipe_t x, sipe_t y, int prec)    \
  {                                           \
    sipe_exp_t d = x.e - y.e;                 \
    sipe_t r;                                 \
                                              \
    if (SIPE_UNLIKELY (x.i == 0))             \
      return (ADD) ? y :                      \
        (sipe_t) { - y.i, y.e };              \
    if (SIPE_UNLIKELY (y.i == 0) ||           \
        d > prec + SIPE_NEAREST)              \
      return SIPE_TRUNC &&                    \
        ((y.i < 0 && x.i CN 0) ||             \
         (y.i > 0 && x.i CP 0)) ?             \
        sipe_nexttozero (x, prec) : x;        \
    if (d < - (prec + SIPE_NEAREST))          \
      return (r = (ADD) ? y :                 \
              (sipe_t) { - y.i, y.e }),       \
        SIPE_TRUNC && (y.i < 0 ? x.i CN 0     \
                               : x.i CP 0)    \
        ? sipe_nexttozero (r, prec) : r;      \
    r = d < 0 ?                               \
      ((sipe_t) {(x.i)OPS(y.i<<-d),x.e}) :    \
      ((sipe_t) {(x.i<<d)OPS(y.i), y.e});     \
    SIPE_ROUND (r, prec);                     \
    return r;                                 \
  }

SIPE_DEFADDSUB(add,1,+,>,<)
SIPE_DEFADDSUB(sub,0,-,<,>)
```

B. Multiplication

The algorithm of the multiplication function `sipe_mul` corresponds to class 2: we compute $(M_x \cdot M_y, E_x + E_y)$, then round and normalize the result (see Section III-E). The C code is given as an example (Code 2), for its simplicity.

Code 2 Code of multiplication.

```
static inline sipe_t
sipe_mul (sipe_t x, sipe_t y, int prec)
{
  sipe_t r;
  r.i = x.i * y.i;
  r.e = x.e + y.e;
  SIPE_ROUND (r, prec);
  return r;
}
```

However, as already mentioned earlier, we need to be careful in the normalization step, as the addition of the exponents could yield an integer overflow. Indeed, consider the sequence $x_{i+1} = x_i^2$, with $x_0 = 0$ represented by $(M_0, E_0) = (0, 1)$. If normalization left the obtained representation of 0 untouched, then one would get $M_i = 0$ and $E_i = 2^i$, thus an integer overflow on E_i after several iterations. That is why E will be forced to 0 if $M = 0$.

Note: alternatively, we could detect whether $M_x \cdot M_y = 0$ before adding the exponents, but even in this case, the component E of the result must still get some arbitrary value fixed in the code,[7] such as 0, so that this alternative code should be equivalent to the current one after optimizations.

SIPE also provides an "error-free transformation" macro `SIPE_2MUL`, which computes a rounded product `R` and the corresponding error term `S`. The best way to compute the error term is to do this in the rounding process, via another macro, as shown in Code 3. Usual rounding and normalization is just a simplified version of this macro (see Section III-E).

Code 3 Code of multiplication with error term.

```
#define SIPE_2MUL(R,S,X,Y,PREC)                 \
  do                                            \
    {                                           \
      (R) = (sipe_t)                            \
        { (X).i * (Y).i, (X).e + (Y).e };       \
      SIPE_ROUND_ERR (R, 1, (S) =, PREC);       \
    }                                           \
  while (0)
```

C. FMA and FMS

The functions `sipe_fma` and `sipe_fms` respectively compute fused multiply-add $xy+z$ (FMA) and fused multiply-subtract $xy - z$ (FMS), i.e. with a single rounding.

In short, they are implemented by doing an exact multiplication xy (where the xy significand fits on $2p$ bits), then an addition or subtraction similar to `sipe_add` and `sipe_sub`. The main difference is that the first term of the addition/subtraction has a $2p$-bit significand instead of a p-bit one, so that the case where xy is larger in magnitude than z is a bit more difficult because the direction of rounding can come *both* from the lower p-bit part of xy and from z.

In detail: Let $s = 1$ for FMA, $s = -1$ for FMS. Like for addition and subtraction, let $\nu = 1$ in the round-to-nearest mode, $\nu = 0$ in the round-toward-zero mode; in Code 4, `SIPE_NEAREST` is ν and `SIPE_TRUNC` is the complement $1 - \nu$. If $x = 0$ and/or $y = 0$, then $xy = 0$, so that we return $s \cdot z$. Otherwise we compute $t = xy$ exactly. If $z = 0$, we return the rounding of xy (as done with `sipe_mul`). Otherwise we compute the difference $\delta = E_t - E_z$, where $t = M_t \cdot 2^{E_t}$ with $M_t = M_x \cdot M_y$ and $E_t = E_x + E_y$.

- If $\delta > p$, then $|z| = |M_z| \cdot 2^{E_z} < 2^{E_z + p} \leq 2^{E_t - 1}$, i.e. $|z|$ is less than half the quantum of t (actually the representation of t), with $|M_t| \geq 2^{2p-2} \geq 2^p$. Therefore the exact result $t + s \cdot z$ (which we want to round correctly) and the simplified value $t + s \cdot \operatorname{sign}(z) \cdot 2^{E_t - 1}$ have the same

[7]This is a limitation of the ISO C language: it is not possible to just say that the value does not matter while reading it will not trigger an undefined behavior; if a function/macro returning an unspecified value (which cannot be a trap representation, by definition) existed, it could be used here.

rounding (here, since $z \neq 0$, we have $\operatorname{sign}(z) = \pm 1$). The advantage of considering the simplified value is that it has only one more bit than t, so that we can compute it exactly, then round it correctly to get the wanted result.
- If $\delta < -(2p+\nu)$, then $|t| = |M_x| \cdot |M_y| \cdot 2^{E_t} < 2^{E_t+2p} \leq 2^{E_z-\nu-1}$. The following is the same as the proof done for `sipe_add`.

In the remaining cases, $-(2p+1) \leq -(2p+\nu) \leq \delta \leq p$. If $\delta < 0$, we compute $M = M_t + s \cdot M_z \cdot 2^{-\delta}$, and we have: $|M| < 2^{2p} + (2^p - 1) \cdot 2^{2p+1} < 2^{3p+1}$. If $\delta \geq 0$, we compute $M = M_t \cdot 2^{\delta} + s \cdot M_z$, and we have: $|M| < 2^{2p} \cdot 2^p + 2^p < 2^{3p+1}$. Then we round and normalize M (see Section III-E). Since any integer whose absolute value is strictly less than 2^{S-1} is representable in a `sipe_int_t`, the mathematical value M fits in a `sipe_int_t` (no integer overflows) for any precision p such that $3p+1 \leq S-1$. Thus these functions `sipe_fma` and `sipe_fms` are correct for any precision p up to $p_{\max} = \lfloor (S-2)/3 \rfloor$.

Like for addition and subtraction, the code for both functions `sipe_fma` and `sipe_fms` is implemented via a single macro, which is invoked twice for the actual function definitions. It is given as an example (Code 4).

D. Simple Operations (Class 1)

SIPE supports the usual Boolean valued comparisons of numbers `sipe_eq` (=), `sipe_ne` (≠), `sipe_le` (≤), `sipe_lt` (<), `sipe_ge` (≥), `sipe_gt` (>), the minimum and maximum functions `sipe_min` and `sipe_max`, and the magnitude minimum and maximum functions `sipe_minmag` and `sipe_maxmag`, corresponding to the IEEE 754-2008 minNumMag and maxNumMag operations.

Their implementation does not present much difficulty: in short, the signs are compared first (except for the magnitude functions), then in case of non-zero numbers of the same sign, the exponents are compared (this is correct because the representations are normalized), and in case of identical exponents, the significands (or their absolute values) are compared. No roundings are involved.

E. Rounding and Normalization

At the end of operations of class 2, after computing the exact result or a result that would have the same rounding as the exact one, a rounding-and-normalization step is necessary. It is implemented by a `SIPE_ROUND` macro, which takes two arguments: (1) a variable `X` holding a `sipe_t` value to round and normalize; (2) the precision. Let us denote by (M, E) the initial values of the significand and exponent components `X.i` and `X.e` of the variable `X`. The only assumption is that $|M| < 2^{S-1}$.

This `SIPE_ROUND` macro is actually a simplified form of the `SIPE_ROUND_ERR` macro, which can also return the rounding error if $|M| < 2^{2p}$, e.g. after an exact multiplication, as done by `SIPE_2MUL`. Both macros are given in Code 5. In `SIPE_ROUND_ERR`, `X` is the value to be rounded, `C` is a flag set to 1 to return the rounding error, `ERR` is where this rounding error should go (see the `SIPE_2MUL` macro as an example), and `PREC` is the precision.

Code 4 Code of FMA/FMS, slightly edited for this paper.

```
#define SIPE_DEFFMAFMS(OP,FMA,OPS,CN,CP)       \
  static inline sipe_t                          \
  sipe_##OP (sipe_t x, sipe_t y, sipe_t z,      \
             int prec)                          \
  {                                             \
    sipe_t t, r;                                \
    sipe_exp_t d;                               \
                                                \
    t.i = x.i * y.i;                            \
    if (SIPE_UNLIKELY (t.i == 0))               \
      return (FMA) ? z :                        \
        (sipe_t) { - z.i, z.e };                \
    t.e = x.e + y.e;                            \
    if (SIPE_UNLIKELY (z.i == 0))               \
      {                                         \
        SIPE_ROUND (t, prec);                   \
        return t;                               \
      }                                         \
    d = t.e - z.e;                              \
    if (d > prec)                               \
      {                                         \
        r = (sipe_t) {                          \
          2 * t.i OPS (z.i < 0 ? -1 : 1),       \
          t.e - 1 };                            \
        SIPE_ROUND (r, prec);                   \
        return r;                               \
      }                                         \
    if (d < - (2 * prec + SIPE_NEAREST))        \
      return (r = (FMA) ? z :                   \
              (sipe_t) { - z.i, z.e }),         \
        SIPE_TRUNC && (z.i < 0 ? t.i CN 0       \
                               : t.i CP 0)      \
        ? sipe_nexttozero (r, prec) : r;        \
    r = d < 0 ?                                 \
      ((sipe_t) {(t.i)OPS(z.i<<-d),t.e}) :      \
      ((sipe_t) {(t.i<<d)OPS(z.i), z.e });      \
    SIPE_ROUND (r, prec);                       \
    return r;                                   \
  }

SIPE_DEFFMAFMS(fma,1,+,>,<)
SIPE_DEFFMAFMS(fms,0,-,<,>)
```

The `SIPE_ROUND_ERR` macro works in the following way. First, if $M = 0$, we just need to set the exponent field `X.e` to 0 and the error term to 0. Now assume that $M \neq 0$. We will work mainly on its absolute value $|M|$. Since $|M| < 2^{S-1}$, it is representable in the `sipe_int_t` type. Then we compute the difference d between the precision p of the SIPE floating-point system and the size (in bits) of $|M|$. We distinguish the following three cases:

- Difference $d = 0$, i.e. $2^{p-1} \leq |M| \leq 2^p - 1$. The result does not need to be rounded and it is already normalized: there is nothing to do for the variable `X`. We just set the error term to 0.
- Difference $d > 0$, i.e. $|M| \leq 2^{p-1} - 1$. We just need to normalize M and set the error term to 0.
  Normalization is done with two basic operations: shift the significand and correct the exponent.
  One may wonder whether one should do the normalization step here in this macro (which is called at the *end* of an operation) or choose to let results possibly

Code 5 Rounding and normalization code (integer based version), slightly edited for this paper.

```
#define SIPE_ROUND_ERR(X,C,ERR,PREC) do     \
  if (SIPE_LIKELY ((X).i != 0)) {           \
    sipe_int_t _i, _j;                      \
    int _s, _ns;                            \
                                            \
    _i = SIPE_ABSINT ((X).i);               \
    _s = (PREC) - SIPE_SIZE + sipe_clz(_i); \
    if ((SIPE_ROUND_ZOPT) ? _s>=0 : _s>0)   \
      {                                     \
        (X).i <<= _s;                       \
        (X).e -= _s;                        \
        ERR (sipe_t) { 0, 0 };              \
      }                                     \
    else if (!(SIPE_ROUND_ZOPT) && _s == 0) \
      {                                     \
        ERR (sipe_t) { 0, 0 };              \
      }                                     \
    else                                    \
      {                                     \
        _ns = - SIPE_NEAREST - _s;          \
        _j = _i >> _ns;                     \
        if (SIPE_NEAREST)                   \
          {                                 \
            if ((_j&2) | (_i - (_j<<_ns)))  \
              _j++;                         \
            _j >>= 1;                       \
            if (SIPE_UNLIKELY               \
                (_j == SIPE_TWO_TO (PREC))) \
              {                             \
                _j >>= 1;                   \
                _ns++;                      \
              }                             \
          }                                 \
        if (C)                              \
          {                                 \
            _i -= _j << (_ns+SIPE_NEAREST); \
            if (_i == 0)                    \
              ERR (sipe_t) { 0, 0 };        \
            else                            \
              {                             \
                _s = (PREC) - SIPE_SIZE +   \
                  sipe_clz(SIPE_ABSINT(_i));\
                SIPE_ASSERT (_s >= 0);      \
                ERR (sipe_t)                \
                  { ((X).i >= 0 ? _i : -_i) \
                      << _s, (X).e - _s };  \
              }                             \
          }                                 \
        (X).i = (X).i >= 0 ? _j : - _j;     \
        (X).e += _ns + SIPE_NEAREST;        \
      }                                     \
  } else {                                  \
    (X).e = 0;                              \
    ERR (sipe_t) { 0, 0 };                  \
  }                                         \
while (0)

#define SIPE_ROUND(X,PREC) \
  SIPE_ROUND_ERR(X,0,(void),PREC)
```

unnormalized, in which case a normalization step at the beginning of some operations or a more complex implementation would be needed. The advantage of the latter choice is to avoid unnecessary normalization, but if it needs a costly handling of unnormalized operands just to avoid the two basic operations mentioned above, it may turn into a drawback. Let us note that if it can be detected at compile time that the next operation does not need normalized operands (e.g., in the case of a multiplication), then the normalization step here could be avoided without any run-time test, assuming that the functions are inlined. It has not been found yet how to do this in a clean and simple way.

Now, what can we expect to gain by avoiding unnecessary normalization? This depends on the program, but it is believed that in most applications, at least those using SIPE, the computed significands (before rounding) do not fit on $p-1$ bits in general, thus tend to become normalized automatically from the case $d < 0$ below; thus these two additional basic operations will be needed only during initialization (but in case of constants, the compiler can do them at compile time) and after a cancellation, so that the gain should be very low.

- Difference $d < 0$, i.e. $|M| \geq 2^p$. We need to round the value according to the chosen rounding mode and compute the error term, which is done in the following way.

  Both supported rounding modes are symmetrical, so that we can just round the absolute value $|M|$ of the significand, without taking its sign into account.

  - In the round-toward-zero mode, we set j to $|M|$ shifted $-d$ bit positions to the right: this truncates the value to p bits, which is exactly what we want.
  - Handling the round-to-nearest mode is more difficult: we use the formula $j = \lfloor (j_0 + u)/2 \rfloor$, where j_0 is $|M|$ truncated on $p + 1$ bits (i.e. right-shifted $-1 - d$ bit positions), and $u = 1$ except when the truncated significand on p bits is even and the exact significand fits on $p+1$ bits (said otherwise, the *sticky bit* is zero), in which case $u = 0$. Note: without this particular case $u = 0$, one would obtain the value in roundTiesToAway (halfway cases rounded away from zero) instead of roundTiesToEven (even-rounding rule). If $|M|$ has been rounded up to 2^p, i.e. falls in the next binade, then we change j to 2^{p-1}, implying an increment of the exponent.

  Then `X.i` is set to $\pm j$ with the correct sign, and the quantum exponent `X.e` is corrected.

  In the case the macro call asks for the error term: The code assumes that $|M| < 2^{2p}$, so that the error term is exactly representable; this term is computed before the exponent correction. All the significands mentioned here will be associated with the quantum exponent E. Let M' be the rounded significand. Then the error term has the significand $M_e = M - M'$. Since M and M' are both integers, M_e is also an integer. Moreover $|M'| < \mathrm{ulp}(M) \leq 2^p$, so that the error term is exactly representable (as said above). $|M'|$ is computed by left-shifting j, and since M and M' have the same sign, we can compute $M_e \cdot \mathrm{sign}(M) = |M| - |M'|$. If we obtain 0, then the error term is set to 0. Otherwise we take the result with the correct sign and normalize it.

The case $d = 0$ can actually be regarded as a particular case of $d \geq 0$, where the normalization leaves the values unchanged: a shift by 0 and an addition with 0. SIPE has an option (by setting `SIPE_ROUND_ZOPT` to 1) to merge these cases, yielding two additional useless operations in some cases, but avoiding a test and a branch to distinguish these two cases. Tests on several machines showed that, depending on the context, this option could make the code faster or slower.

IV. Implementation: Floating-Point Version

A. About Rounding

The main idea for this implementation is that the SIPE numbers are represented in a native floating-point format in precision q, typically with $q = 24$ (binary32, a.k.a. single precision), $q = 53$ (binary64, a.k.a. double precision) or $q = 64$ (x86 extended precision, when available), and that an operation on SIPE numbers would be done in two steps:

1) the operation in the native floating-point format, thus with a rounding in precision q, this step being very fast, as entirely done in hardware;
2) a rounding to the target precision p, for instance implemented with Veltkamp's splitting algorithm [9], [10] (and [8, §4.4.1]), as shown on Code 6.

Code 6 Rounding and normalization code (floating-point based version), slightly edited for this paper.

```
#define SIPE_POWD(PREC) \
  (1UL << (SIPE_NATIVE_PREC - PREC))

#define SIPE_ROUND_ERR(X,C,ERR,PREC)          \
  do                                          \
    {                                         \
      sipe_t _y, _z;                          \
      _y = (X) * (SIPE_POWD(PREC) + 1);       \
      _z = (X) - _y;                          \
      if (C)                                  \
        {                                     \
          sipe_t _r = _z + _y;                \
          ERR ((X) - _r);                     \
          (X) = _r;                           \
        }                                     \
      else                                    \
        (X) = _z + _y;                        \
    }                                         \
  while (0)

#define SIPE_ROUND(X,PREC) \
  SIPE_ROUND_ERR(X,0,(void),PREC)
```

It can be proved that such a process always gives the correct rounding when the three roundings involved (the specified rounding and both roundings of the process) are in some fixed directed rounding mode; but in the case of rounding to nearest, primarily considered in SIPE, the rounding may be done in the wrong direction for some particular operations and inputs. Indeed, when the result of the first rounding is the middle of two consecutive machine numbers of the target system in precision p (i.e. its significand is a $p+1$-bit number that does not fit on p bits), the correctly-rounded result not only depends on the result of the first rounding, but also on the sign of its error; unfortunately, the rule used for the second rounding does not usually depend on this second information, with the consequence that the final rounding may be incorrect. This is the well-known *double-rounding problem*. This means that one would need to detect when double rounding can have a side effect, i.e. detect halfway cases for precision p, and write special code for this. We could consider alternative possibilities, such as modifying one of the two roundings (or the combination of both) without using branching and special code:

- For the first rounding, using a special rounding mode called *rounding to odd*, proposed by Boldo and Melquiond in [11], would avoid the side effect of double rounding. However such a rounding mode is not standard and not available in hardware, and emulating it would be slower than trying to solve the double-rounding problem in a more direct way.
- The second rounding is implemented in software (though based on common operations implemented in hardware), thus could be modified. Unfortunately, contrary to MPFR, IEEE 754 floating point does not give access to the sign of the error of an operation, so that there are no generic methods for this point.
- Going back to the first rounding, one could also use a directed rounding mode. The same kind of problem occurs, but with a major difference: due to directed rounding instead of rounding to nearest, the sign of the error no longer matters, just whether the error is zero or not, and such an information is provided by the IEEE 754 *inexact* flag. Using rounding toward zero and the *inexact* flag is actually one of the methods proposed by Boldo and Melquiond to implement rounding to odd. It might be interesting on processors with static rounding modes and fast access to the inexact flag, and good compiler support.

Therefore such a representation was initially thought to lead to a slow implementation. However, we noticed later that since the precision is low enough in the context of SIPE, the double-rounding problem does not occur here for the addition, subtraction and multiplication operations, i.e. the second rounding yields the correctly-rounded result, thus eventually making this representation particularly interesting.

B. Addition, Subtraction and Multiplication

Let us now detail why the double-rounding problem does not occur for addition, subtraction and multiplication.

- For addition and subtraction $x \pm y$: If x and/or y is zero, then the native operation is exact, so that there will be only one rounding, and the result will be correct. Otherwise, without loss of generality, let us assume that $|x| \geq |y|$. If $|y| < \frac{1}{4}\,\mathrm{ulp}(x)$, then the result of the native operation cannot be a halfway number in precision p;

there may really be two roundings, but the final result will be correct. If $|y| \geq \frac{1}{4}\,\mathrm{ulp}(x)$, then $x \pm y$ fits on $2p+1$ bits, and if $q \geq 2p+1$, then the native operation (in precision q) is exact.

- For multiplication, if $q \geq 2p$, then the native operation is exact.

Therefore, under the condition $p \leq (q-1)/2$, the addition, subtraction and multiplication operations are correctly-rounded by using a native operation followed by rounding to precision p. This property makes the code for these common operations rather simple, as shown on Code 7.

Code 7 Code of addition, subtraction and multiplication, slightly edited for this paper.

```
#define SIPE_DEFOP(OP,OPS)                      \
  static inline sipe_t sipe_##OP                \
    (sipe_t x, sipe_t y, int prec)              \
  {                                             \
    sipe_t r = x OPS y;                         \
    SIPE_ROUND (r, prec);                       \
    return r;                                   \
  }

SIPE_DEFOP(add,+)
SIPE_DEFOP(sub,-)
SIPE_DEFOP(mul,*)
```

C. FMA and FMS

Unfortunately double rounding can occur for FMA/FMS ($xy \pm z$). If $q \geq 3p-1$, i.e. $p \leq (q+1)/3$, it can occur only when xy is the middle of two consecutive machine numbers of the target system (in precision p) and the native operation $xy \pm z$ rounds to xy (because $|z|$ is small enough compared to $|xy|$). Actually this cannot happen in precision 2, since the only inexact product of integral significands is, in binary, $11 \times 11 = 1001$, which is not a 3-bit odd integer. But for simplicity, we do not regard precision 2 as a special case (even when the precision is the constant 2): only hypothetical applications computing only in precision 2 would benefit from such a specific optimization, and we do not believe that it would be worth code bloat.

So, for FMA/FMS, the current SIPE code uses Veltkamp's splitting to detect whether the significand of the rounded result r in the native precision fits on $p+1$ bits. If it does, z is non-zero and $r+z$ rounds to r (meaning that $r = xy \neq 0$), then r is slightly modified in the direction given by the sign of z, by using a `nextafter*` function (a multiplication by a constant close to 1 was also tried instead of `nextafter*`, but this was not faster).

Concerning the implementation of the native operation, since xy is exact, we have the choice between using a real FMA and using separate multiplication and addition. The behavior will be the same (even the sign when the result is zero, though this does not matter here), but the use of a hardware FMA will normally yield a faster executable. Thus we try to use it if available, via standard C code, by testing one of the `FP_FAST_FMA*` macros, depending on the native floating-point type, and using one of the `fma*` functions if the corresponding macro is defined. If the macro is not defined, we try an alternate way: set the `STDC FP_CONTRACT` pragma to on and compute the operation with the `x * y + z` expression; due to the pragma, the compiler is allowed to evaluate this expression with a FMA unconditionally. If a hardware FMA is not available or not supported by the C implementation, then separate multiplication and addition will be performed.

V. RESULTS AND TIMINGS

In [3], [4], the TwoSum algorithm in radix 2 (due to Knuth [12] and Møller [13]) was proved to be minimal among algorithms only based on additions and subtractions in the round-to-nearest mode. The initial proof was done with GNU MPFR, but it has later been checked with SIPE. The programs are provided on http://hal.inria.fr/inria-00475279 (see attached files in annex, in the detailed view). Only `minasm.c` can be compiled against SIPE (as an alternative to MPFR); this is the program used to prove several minimality properties of the TwoSum algorithm by testing all possible algorithms on a few well-chosen inputs, thus eliminating all the incorrect algorithms. Note: the `sipe.h` file provided at this URL is an old version from 2009 and contains bugs (`minasm.c` is not affected by these bugs, though).

In order to evaluate the performance of SIPE, we chose to compare the execution times of `minasm`, built against:

- the `double` native floating-point type (IEEE 754 double-precision, i.e. 53 bits, in hardware);
- MPFR in precision 12 (denoted MPFR in the tables);
- the integer based version of SIPE in precision 12 (chosen at compile time), with `SIPE_ROUND_ZOPT` being 0 (denoted SIPE/0) and 1 (denoted SIPE/1);
- the floating-point based version of SIPE in precision 12 (chosen at compile time), with the `double` (denoted SIPE/D) and `long double` (denoted SIPE/L) C types, except on PowerPC, where `long double` is not used (on such platforms, this type corresponds to double-double arithmetic [14], [8, §14.1.1], while SIPE requires a floating-point format with correct rounding in the floating-point system).

For better comparison, the same precision should have been chosen for each implementation, but this is not possible. However, as shown in [3], the choice of precision 12 leads to operations that are similar for any precision $p \geq 12$ (this property is the base of the proof of the minimality of TwoSum).

Moreover, not all SIPE functions are called by `minasm`, but this program mainly uses addition, subtraction, and rounding, which more or less correspond to the trickiest part of the integer based version of SIPE (together with FMA). Functions involving multiplication and FMA are currently not tested at all for the timings, and this could be part of future work.[8]

[8]For best timing results with floating point, tests should be carried out on a machine with a hardware FMA, as most machines will have one in the near future. However this will probably have very little effect here.

Now that the examples have been chosen, it is important to get meaningful and accurate timings. Ideally everyone should always use the "best" compiler, but it may not be available, and the quality of generated code can depend on the compiler and other factors. Here we used GCC, as it is widely available, required by the integer version of SIPE, and known to be a good compiler; all the GCC versions installed on the machines were tested. After choosing the compiler, it is important to choose good optimization options. Indeed it is meaningless to compare performance if code is poorly optimized. On x86_64 machines, we chose the `-O3 -march=native -std=c99` options as this should be the best for GCC (without additional knowledge and without testing every combination), while keeping the generated code strictly correct (by default, GCC does not necessarily follow the ISO C standard). On a PowerPC machine, the `-march` option is not supported, and with the `-O2` and `-O3` options, we got incorrect results with the floating-point version of SIPE (due to a compiler bug?); thus the `-O` option was used. Different profiles have also been tested, thanks to the `-fprofile-generate` and `-fprofile-use` GCC options, with the condition that profile generation must be fast compared to the actual test. The timings below show that the choice of how a code is compiled is important: we could get up to a factor 2 on the same machine between two GCC versions!

For most tests, we did not recompile the libraries (in particular GMP and MPFR) and we linked against them dynamically, as this is usually done in practice, for good reasons. However, in cases where computations can run for more than a few days, it may be interesting to spend time to get linkage related optimizations. One of the techniques is to use static linking (e.g., with GCC's `-static` option); the speedup *without combinations of other techniques* should remain limited, though. Another technique is to enable link-time optimizations (LTO), introduced in the most recent GCC versions;[9] however this requires to recompile the libraries with the same compiler and LTO specific options (SIPE, thanks to its design using a header file, does not have this drawback). Various tests have shown that one can really benefit from LTO only with static linking, which is not surprising. On some tests on an Intel Xeon based machine, it has been seen that the global LTO speedup could be up to 37%, which is quite important (hoping that this is not due to wrong code, see below); this speedup was due to three factors: the rebuild of the GMP and MPFR libraries for the target processor (instead of generic x86_64), static linking, and LTO itself. A drawback, like with aggressive optimizations in general, is that when software has not been carefully tested (if not proved), LTO may make new bugs visible. For instance, during the tests done for this paper, a GMP 5.1.2 bug (an integer overflow) was triggered with a GCC snapshot and LTO, making the GMP testsuite fail. Further analysis with Clang's sanitizer revealed undefined behavior in several other places in GMP. As such bugs might affect the generated code with LTO, in doubt, we do not give LTO timings here. But according to Tables III and IV of [7], the trend seems to be the following: as expected, the tests with MPFR were always significantly faster; and with some profiles, the other tests could also be faster, but we do not have any explanation on the precise influence of the profile.

Tables I, II, III, IV, V, VI, VII, and VIII present `minasm` timings on two different machines with x86_64 processors and a machine with PowerPC processors. The programs (including SIPE itself) used to generate these tables can be downloaded via the following archive: https://www.vinc17.net/software/sipe-timings-201309.tar.xz. Compilation has been done with the following profile modes (g-column in the tables):

- no profiles (−);
- profile generation on `minasm 0 2 6` (2);
- profile generation on `minasm 0 4 5` (4);
- profile generation on `minasm 0 6 5` (6).

In order to detect potential bugs, the exit status of each run of `minasm` is tested, and the outputs are compared. For instance, running `minasm` built against the `double` type quickly fails on 32-bit x86 machines due to the use of extended precision, unless the GCC `-mfpmath=sse` option is provided. This problem is detected automatically. The incorrect results with `-O3` on PowerPC were also detected automatically thanks to output comparison.

These timings include the overhead for the input data generation (here, computation DAG's) and the tests of the results; thus the real ratios are probably significantly higher. But these are timings on a real-world program used as a part of a proof, not just a raw, theoretical benchmark using synthetic tests, which would not necessarily be representative in practice; other programs should be tested in the future.

From these timings, we can do the following remarks:

- GCC versions 4.5 and 4.6 generate code that is twice as slow for the integer version of SIPE. With these versions, adding round-toward-zero support resulted in even slower code, though this should have not changed anything with constant propagation analysis. There is probably an optimization bug in these GCC versions.
- If we ignore these bad GCC versions 4.5 and 4.6, the use of the integer version of SIPE, in these cases, is between 1.2 and 6 times as slow as the use of `double` (but the test on `double` does not allow one to deduce TwoSum minimality results for precisions up to 11, so that an arbitrarily-low precision library is really needed). And the use of the integer version of SIPE is between 3 and 6 times as fast as the use of MPFR for precision 12.
- The floating-point version of SIPE is much faster in these cases, and just a bit slower than the direct use of `double`. However it must be used with care, as wrong code generation (failing or giving incorrect results) has been detected on some machines, as said above. In practice, comparisons with the integer based version of SIPE on a subset of the problem to solve are recommended. This is easy to do, as it generally suffices to recompile the program without defining the `SIPE_FLOAT` macro.
- Some timings look surprising, such as the "1 4 6 2" lines on AMD Opteron (Tables VI and VII) for floating point (`double`, SIPE/D and SIPE/L), but they are reproducible. This is not a measurement error.

[9]http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
http://gcc.gnu.org/onlinedocs/gccint/LTO.html

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.44	8.64	2.26	2.24	0.50	0.85
1 2 6	2	0.39	8.72	1.95	1.93	0.52	0.82
1 2 6	4	0.38	8.80	2.12	2.06	0.50	0.82
1 2 6	6	0.42	8.67	2.14	2.02	0.52	0.85
1 4 6	–	5.13	64.53	16.14	16.50	5.38	10.68
1 4 6	2	6.10	67.58	15.43	15.90	6.73	12.28
1 4 6	4	5.15	63.25	15.90	15.72	5.57	10.98
1 4 6	6	5.33	64.62	16.27	15.76	5.81	11.11
1 6 5	–	0.20	1.70	0.43	0.44	0.19	0.35
1 6 5	2	0.23	1.81	0.47	0.48	0.32	0.44
1 6 5	4	0.22	1.78	0.49	0.50	0.23	0.37
1 6 5	6	0.24	1.73	0.48	0.48	0.25	0.39

TABLE I
TIMINGS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX), WITH GCC 4.4.7 (DEBIAN 4.4.7-4).

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.42	8.76	2.73	2.69	0.50	0.90
1 2 6	2	0.41	8.73	2.39	2.40	0.53	0.83
1 2 6	4	0.39	8.73	2.44	2.42	0.52	0.81
1 2 6	6	0.39	8.76	2.48	2.44	0.50	0.86
1 4 6	–	5.16	64.81	31.33	31.02	5.45	10.93
1 4 6	2	8.80	66.96	30.01	30.04	8.89	11.18
1 4 6	4	7.06	64.19	29.04	28.93	7.34	10.01
1 4 6	6	5.72	64.50	29.56	29.13	6.96	10.84
1 6 5	–	0.19	1.75	0.91	0.90	0.19	0.38
1 6 5	2	0.33	1.80	0.88	0.88	0.32	0.39
1 6 5	4	0.30	1.78	0.87	0.88	0.30	0.37
1 6 5	6	0.25	1.74	0.86	0.86	0.26	0.39

TABLE II
TIMINGS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX), WITH GCC 4.6.4 (DEBIAN 4.6.4-4).

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.54	8.78	2.03	2.04	0.53	0.92
1 2 6	2	0.38	8.83	1.69	1.69	0.54	0.83
1 2 6	4	0.38	8.81	1.79	1.88	0.50	0.84
1 2 6	6	0.43	8.75	1.89	1.88	0.48	0.88
1 4 6	–	5.24	64.09	14.95	14.71	5.64	12.20
1 4 6	2	7.88	67.34	14.61	14.89	8.42	12.48
1 4 6	4	6.56	64.99	15.41	16.02	7.20	11.80
1 4 6	6	6.66	66.01	15.62	15.83	6.98	13.02
1 6 5	–	0.19	1.78	0.42	0.40	0.22	0.40
1 6 5	2	0.32	1.90	0.43	0.43	0.31	0.42
1 6 5	4	0.28	1.81	0.48	0.50	0.29	0.40
1 6 5	6	0.26	1.75	0.45	0.46	0.26	0.44

TABLE III
TIMINGS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX), WITH GCC 4.7.3 (DEBIAN 4.7.3-7).

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.54	9.28	2.26	2.12	0.55	0.91
1 2 6	2	0.38	8.75	2.24	2.24	0.53	0.82
1 2 6	4	0.38	8.74	2.10	2.14	0.53	0.83
1 2 6	6	0.45	8.73	2.22	2.11	0.56	0.88
1 4 6	–	5.37	69.07	16.05	15.90	5.57	12.10
1 4 6	2	7.64	67.44	17.53	17.98	8.49	12.89
1 4 6	4	6.55	66.10	15.95	16.03	7.81	10.80
1 4 6	6	5.80	65.61	16.88	16.22	7.68	12.18
1 6 5	–	0.20	1.86	0.42	0.42	0.20	0.39
1 6 5	2	0.32	1.85	0.49	0.50	0.32	0.43
1 6 5	4	0.29	1.80	0.46	0.46	0.32	0.41
1 6 5	6	0.25	1.79	0.46	0.46	0.28	0.42

TABLE IV
TIMINGS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX), WITH GCC 4.8.1 (DEBIAN 4.8.1-10).

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.50	9.25	2.39	2.15	0.55	0.91
1 2 6	2	0.41	8.70	2.27	2.26	0.54	0.82
1 2 6	4	0.44	8.76	2.20	2.18	0.54	0.82
1 2 6	6	0.40	8.87	2.26	2.17	0.54	0.89
1 4 6	–	5.14	69.10	16.60	15.61	5.54	12.24
1 4 6	2	6.66	67.15	17.92	17.71	9.00	12.60
1 4 6	4	6.22	65.56	16.99	16.67	7.47	11.01
1 4 6	6	5.87	66.37	17.06	17.10	7.35	12.31
1 6 5	–	0.20	1.86	0.43	0.41	0.19	0.39
1 6 5	2	0.28	1.82	0.50	0.50	0.33	0.41
1 6 5	4	0.27	1.80	0.48	0.48	0.30	0.38
1 6 5	6	0.27	1.79	0.47	0.47	0.28	0.44

TABLE V
TIMINGS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX), WITH A GCC 4.9 (TRUNK) SNAPSHOT (DEBIAN 20130917-1).

VI. CONCLUSION

We presented a small library whose purpose is to do simple operations in binary floating-point systems in very low precisions with correct rounding to nearest or toward zero, in order to test the behavior of simple floating-point algorithms (correctness, error bounds, etc.) on a huge number of inputs (numbers and/or computation trees, for instance). For that, we sought to be as fast as possible, thus did not want to handle special numbers and exceptions.

We dealt with the main difficulties of SIPE, hoping nothing has been forgotten; [15] includes an old version of the full SIPE source and gives a more detailed proof of the implementation, which was only integer based with rounding to nearest at that time. To go further, one would need to write a formal proof, where the ISO C language (including the preprocessor, since SIPE quite heavily relies on it) and GCC features would

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.53	8.37	2.54	2.45	0.68	1.65
1 2 6	2	0.58	8.02	2.62	2.40	0.78	1.66
1 2 6	4	0.58	8.75	2.59	2.42	0.78	1.69
1 2 6	6	0.55	8.81	2.60	2.51	0.78	1.69
1 4 6	–	9.32	60.06	18.98	18.55	10.08	17.61
1 4 6	2	12.98	62.28	19.51	18.46	13.16	18.19
1 4 6	4	9.96	68.74	18.84	18.09	10.50	18.02
1 4 6	6	9.90	70.19	19.45	18.54	11.08	18.02
1 6 5	–	0.33	1.82	0.55	0.54	0.34	0.57
1 6 5	2	0.46	2.01	0.57	0.57	0.42	0.59
1 6 5	4	0.35	2.03	0.55	0.55	0.35	0.58
1 6 5	6	0.33	1.98	0.55	0.53	0.34	0.57

TABLE VI
TIMINGS OBTAINED ON A 64-BIT 2.4GHZ AMD OPTERON 8378 (DEBIAN/6.0.7, A.K.A. SQUEEZE, GNU/LINUX), WITH GCC 4.3.5 (DEBIAN 4.3.5-4).

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.57	9.96	3.70	3.64	0.67	n/a
1 2 6	2	0.43	9.76	3.27	2.97	0.56	n/a
1 2 6	4	0.46	9.88	3.03	2.98	0.54	n/a
1 2 6	6	0.42	9.88	3.00	2.98	0.54	n/a
1 4 6	–	8.49	76.38	30.67	30.26	8.34	n/a
1 4 6	2	7.35	75.71	25.79	26.94	7.33	n/a
1 4 6	4	6.91	77.07	23.82	23.36	7.17	n/a
1 4 6	6	6.18	76.93	23.41	23.59	7.07	n/a
1 6 5	–	0.29	2.12	0.89	0.87	0.27	n/a
1 6 5	2	0.25	2.13	0.79	0.80	0.25	n/a
1 6 5	4	0.24	2.18	0.79	0.71	0.24	n/a
1 6 5	6	0.20	2.11	0.68	0.69	0.23	n/a

TABLE VIII
TIMINGS OBTAINED ON A 64-BIT 3.55GHZ POWER7 MACHINE OF THE GCC COMPILE FARM, WITH GCC 4.7.2 (RED HAT 4.7.2-8) AND THE `-O` COMPILER OPTION INSTEAD OF `-O3`.

		timings (in seconds)					
args	g	`double`	MPFR	SIPE/0	SIPE/1	SIPE/D	SIPE/L
1 2 6	–	0.55	8.82	2.94	2.86	0.70	1.60
1 2 6	2	0.47	8.73	2.72	2.57	0.68	1.59
1 2 6	4	0.50	8.63	2.73	2.67	0.64	1.60
1 2 6	6	0.53	8.73	2.81	2.64	0.68	1.61
1 4 6	–	9.39	67.17	21.15	20.79	10.41	17.49
1 4 6	2	10.80	69.70	20.38	19.26	12.62	24.85
1 4 6	4	8.89	67.51	20.06	19.80	10.05	17.68
1 4 6	6	9.80	69.60	20.94	19.51	10.73	17.61
1 6 5	–	0.33	1.92	0.59	0.58	0.32	0.58
1 6 5	2	0.35	1.96	0.60	0.58	0.42	0.75
1 6 5	4	0.35	2.01	0.60	0.59	0.34	0.58
1 6 5	6	0.34	1.96	0.58	0.57	0.35	0.56

TABLE VII
TIMINGS OBTAINED ON A 64-BIT 2.4GHZ AMD OPTERON 8378 (DEBIAN/6.0.7, A.K.A. SQUEEZE, GNU/LINUX), WITH GCC 4.4.5 (DEBIAN 4.4.5-8).

also need to be formalized. However we have also done almost-exhaustive tests of some functions in some precisions on an x86_64 platform, namely the following functions have been tested against GNU MPFR on zero and *all* (normalized) `sipe_t` values having an exponent between -15 and $13+p$, i.e. $1+(29+p)\cdot 2^p$ values per argument for precision p. For the integer version, both rounding modes, and both `SIPE_ROUND_ZOPT = 0` and `SIPE_ROUND_ZOPT = 1` have been tested. For the floating-point version, the three native C types `float` (24-bit precision), `double` (53-bit precision) and `long double` (64-bit precision) have been tested. For each of the 7 cases:

- `sipe_next{above,below,tozero}` on non-zero arguments in precisions $p = 2$ to 7: $\sum_{p=2}^{7}(29+p)\cdot 2^p = 8\,844$ tests for each function;
- `sipe_add`, `sipe_sub`, `sipe_mul`, `SIPE_2MUL`, in precisions $p = 2$ to 7: $\sum_{p=2}^{7}(1+(29+p)\cdot 2^p)^2 = 27\,812\,398$ tests for each function;
- `sipe_fma` and `sipe_fms`, in precisions $p = 2$ to 7: $\sum_{p=2}^{7}(1+(29+p)\cdot 2^p)^3 = 110\,621\,353\,626$ tests for each function;
- `sipe_add_si`, `sipe_sub_si`, `sipe_mul_si` (operations between a SIPE floating-point number and a native integer with a p-bit precision) in precisions $p = 2$ to 7, with all values of the integer argument i such that $|i| \le 2^p$: $\sum_{p=2}^{7}(1+(29+p)\cdot 2^p)(2^{p+1}+1) = 1\,567\,338$ tests for each function;
- `sipe_eq`, `sipe_ne`, `sipe_le`, `sipe_gt`, `sipe_ge`, `sipe_lt`, `sipe_min`, `sipe_max` in precisions $p = 2$ and $p = 3$: $\sum_{p=2}^{3}(1+(29+p)\cdot 2^p)^2 = 81\,674$ tests for each function.

The above tests took a total of about 99 hours (mainly due to the time spent in the MPFR FMA/FMS functions); less than 12 hours were required for up to precision 6. Thanks to these tests, two obvious sign-related bugs had been found in an early version of SIPE.

Future work will consist in using SIPE for other problems than the minimality of the TwoSum algorithm. For instance, work to find the largest relative error for the DblMult algorithm as described in [16] has started, but the fact that this function has four inputs makes the search quite difficult; however, once some guesses have been done in the smallest precisions, one does not need to test the whole domain for higher precision. This would allow us to conjecture a very tight precision-independent error bound, then attempt to prove it.

For some other works, improving SIPE may be needed. This could mean implementing other operations, such as division and square root, and other error-free transformations, such as `SIPE_2SUM` (a macro that would compute a rounded sum and the corresponding error term).

Support for round-toward-zero in the floating-point based implementation could also be added. It should algorithmically be simpler than round-to-nearest due to the absence of the double-rounding problem, but it involves a change of the rounding direction attribute needed for the native operations, and there are technical problems behind that: ISO C specifies a `fesetround` function, but should it be called locally by SIPE or by the user before using SIPE? And one would need to check the compiler support; for instance, GCC needs specific options. This would also need some tests on processors with static rounding modes.

Another future SIPE improvement could be the support of other rounding modes (toward plus infinity and toward minus infinity). Decimal support would also be interesting, but would require a new floating-point representation and a complete rewrite.

REFERENCES

[1] IEEE, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. New York: Institute of Electrical and Electronics Engineers, 1985. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.1985.82928

[2] ——, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[3] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller, "On the computation of correctly-rounded sums," INRIA, Lyon, France, Research report RR-7262, Apr. 2010. [Online]. Available: http://hal.inria.fr/inria-00475279

[4] ——, "On the computation of correctly-rounded sums," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 289–298, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2011.27

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[6] International Organization for Standardization, *ISO/IEC 10967-1: Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*. International Organization for Standardization, 1994.

[7] V. Lefèvre, "SIPE: Small integer plus exponent," in *Proceedings of the 21th IEEE Symposium on Computer Arithmetic*, Austin, Texas, USA, Apr. 2013. [Online]. Available: http://hal.inria.fr/hal-00763954

[8] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torrès, *Handbook of Floating-Point Arithmetic*, 1st ed. Birkhäuser Boston, 2010. [Online]. Available: http://www.springer.com/birkhauser/mathematics/book/978-0-8176-4704-9

[9] G. W. Veltkamp, "ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie," RC-Informatie, Technishe Hogeschool Eindhoven, Tech. Rep. 22, 1968.

[10] ——, "ALGOL procedures voor het rekenen in dubbele lengte," RC-Informatie, Technishe Hogeschool Eindhoven, Tech. Rep. 21, 1969.

[11] S. Boldo and G. Melquiond, "Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd," *IEEE Transactions on Computers*, vol. 57, no. 4, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1109/TC.2007.70819

[12] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison Wesley, 1998, vol. 2.

[13] O. Møller, "Quasi double-precision in floating-point addition," *BIT*, vol. 5, pp. 37–50, 1965.

[14] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, no. 3, pp. 224–242, 1971.

[15] V. Lefèvre, "SIPE: Small integer plus exponent," INRIA, Lyon, France, Research report RR-7832, Dec. 2011. [Online]. Available: http://hal.inria.fr/hal-00650659

[16] P. Kornerup, C. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller, "Computing correctly rounded integer powers in floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 37, no. 1, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1644001.1644005