SIPE: Small Integer Plus Exponent

Vincent Lefèvre

SIPE: Small Integer Plus Exponent

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

***Abstract*—SIPE (Small Integer Plus Exponent) is a mini-library in the form of a C header file, to perform computations in very low precisions with correct rounding to nearest in radix 2. 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 the addition, the subtraction, the multiplication, the FMA, and miscellaneous comparisons and conversions. 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 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 if 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 (the representation chosen in SIPE does not make a sign appear explicitly like here).

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*). Here we will only deal with rounding to nearest, with the *even-rounding rule* if the exact result is the middle of two consecutive machine numbers: this is the rounding-to-nearest mode of IEEE 754-1985, and roundTiesToEven in IEEE 754-2008. We chose this rounding-direction attribute because this is the default one for binary formats [2, §4.3.3] and various floating-point algorithms were designed for this one in particular.

The most common binary formats and most often implemented are the two formats entirely specified by the first 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 the performance 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 in a low precision, that 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

[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 be avoided.

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 uses the usual 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 the division, but not the FMA;
- the *sreal* library does not seem to be very optimized.

Section II presents the basic implementation choices for SIPE. Section III describes some of the algorithms used to implement the operations. We present results and timings in Section IV and conclude in Section V.

II. Basic Choices

Let us recall the criteria we want to focus on:

- The results must be correctly rounded in the chosen floating-point system (binary, low precision). The only rounding mode that will be considered is roundTiesToEven. The other rounding modes could be considered in future work.
- The library needs to 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 deal with 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 the 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/functions 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). More will be said about it later, but first, let us describe how the precisions are handled and how floating-point numbers are encoded.

Contrary to MPFR, where each MPFR object (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 a future work (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.

We have chosen to encode each floating-point number by a structure consisting of two native signed integers (they will typically be registers of the processor): an integer M representing a signed significand and an integer E representing an exponent. Though the integer M can hold values allowing one to represent numbers for up to $p = 32$ or $p = 64$ 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. This gave the name of the library: *Small Integer Plus Exponent* (SIPE), on an idea similar to DPE[5] (meaning *Double Plus Exponent*).

There are several conventions to define the pair (significand,exponent). 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) is a bounded integer (respectively called *integral significand* and *quantum exponent* in [7]). One has the relation: $E = e-p+1$. 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 Position) of x.[6] 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 (see Section III); 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.

[4]From an algorithm point of view, such exceptions will correspond to integer overflows in SIPE. Thus there may be some detection support from the language implementation or the processor, e.g. in a LIA-1 [6] context.

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

[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*.

Let us discuss other possibilities for the encoding of floating-point numbers. We could have chosen:

- 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 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 implementation; this could be observed just by changing the compiler version, see Section IV.
- 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. The operations would have been done in this format (this part being very fast, as entirely done in hardware), and would have been followed by a rounding to the target precision p implemented with Veltkamp's splitting algorithm [8], [9] (and [7, Section 4.4.1]). Unfortunately this method involves two successive roundings: one in the native precision, then one in the target precision. It always gives the correct rounding in directed rounding, but not in rounding to nearest; this is the well-known *double-rounding problem*. Even though a wrong result may occur with a very low probability, one would need to detect it for each operation (and IEEE 754 provides no exceptions for ties, so that this is impossible to detect efficiently).

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 section.

III. The SIPE Implementation

The main idea behind SIPE 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

Let $\delta = E_x - E_y$ when $x \neq 0$ and $y \neq 0$. The algorithm of the addition (`sipe_add`) and subtraction (`sipe_sub`) distinguishes several cases:

1) If $x = 0$, we return y for the addition, $-y$ for the subtraction. This corresponds to class 1.
2) If $y = 0$, we return x. This corresponds to class 1.
3) If $\delta > p + 1$ (corresponding to class 3), then $|y|$ is so small compared to $|x|$ that $x \pm y$ rounds to x. Indeed, $|y| = |M_y| \cdot 2^{E_y} < 2^{E_y+p} \leq 2^{E_x-2}$, and if E_r denotes the exponent of the exact result $r = x \pm y$, then $|E_r - E_x| \leq 1$, so that $|r - x| = |y| < 2^{E_r-1} = \frac{1}{2}\,\mathrm{ulp}(r)$, because x is normalized. Thus we return x.
4) If $-\delta > p + 1$ (corresponding to class 3), then $|x|$ is so small compared to $|y|$ that $x \pm y$ rounds to $\pm y$ (for the same reason). Thus we return $\pm y$.

In the remaining cases, $|\delta| \leq p+1$, 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 compute $x \pm y$ exactly without an integer overflow by $(M_x \pm M_y \cdot 2^{-\delta}, E_x)$ or $(M_x \cdot 2^{\delta} \pm M_y, E_y)$, then round and normalize the result (see Section III-E). This corresponds to class 2.

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). Let us give the code as an example:

```
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 in Section II, 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.

[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.

C. FMA and FMS

The functions `sipe_fma` and `sipe_fms` respectively compute the fused multiply-add $xy + z$ (FMA) and the 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 one of the cases is a bit more difficult.

In detail: Let $s = 1$ for the FMA, $s = -1$ for the FMS. 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`). Then 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 $t + s \cdot \mathrm{sign}(z) \cdot 2^{E_t-1}$ have the same rounding (here, since $z \neq 0$, we have $\mathrm{sign}(z) = \pm 1$). 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 $\delta < -(2p+1)$, then $|t| = |M_x| \cdot |M_y| \cdot 2^{E_t} < 2^{E_t+2p} \leq 2^{E_z-2}$. The following is the same as the proof done for `sipe_add`.

In the remaining cases, $-(2p+1) \leq \delta \leq p$. 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: $|M| < 2^{2p} \cdot 2^p + 2^p < 2^{3p+1}$. 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 $p_{\max} = \lfloor (S-2)/3 \rfloor$.

D. Simple Operations (Class 1)

SIPE supports the usual Boolean valued comparisons of numbers `sipe_eq` ($=$), `sipe_ne` ($\neq$), `sipe_le` ($\leq$), `sipe_lt` ($<$), `sipe_ge` ($\geq$), `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 as arguments: (1) a variable `X` holding a SIPE 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 $|M| < 2^{S-1}$.

Note: this macro is actually a simplified form of another macro, which can also return the rounding error if $|M| < 2^{2p}$, e.g. after an exact multiplication.

This more general 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 s between the precision p of the floating-point system and the size (in bits) of $|M|$. We distinguish the following three cases:

- Difference $s = 0$, 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 $s > 0$, 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. However it is believed that in most applications, at least those using SIPE, the computed (exact) significands do not fit on $p-1$ bits in general, thus tend to become normalized naturally; 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. The benefit is that normalization makes operations easier to implement, as seen above.
- Difference $s < 0$, i.e. $|M| \geq 2^p$. We need to round the value and compute the error term, which is done in the following way.
  To round the absolute value $|M|$ of the significand, we use the formula $j = \lfloor (j_0 + u)/2 \rfloor$, where j_0 is $|M|$ truncated on $p+1$ bits (with a right-shift), and $u = 1$ except if 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: We assume that $|M| < 2^{2p}$; the error 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 $|M'| \leq \frac{1}{2}\,\mathrm{ulp}(M) \leq 2^{p-1}$, so that the error term is exactly representable. $|M'|$ 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 $s = 0$ can actually be regarded as a particular case of $s \geq 0$, 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. Results and Timings

In [3], [4], the TwoSum algorithm in radix 2 (due to Knuth [10] and Møller [11]) 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), and SIPE in precision 12 (chosen at compile time), with `SIPE_ROUND_ZOPT` being 0 (denoted SIPE/0) and 1 (denoted SIPE/1). 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 SIPE (together with the FMA). Functions involving a multiplication are currently not tested at all for the timings, and this could be part of future work.

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 and is known to be a good compiler[8], and all the GCC versions available 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. We chose the options `-O3 -march=native -std=c99` 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). 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, mentioned below, 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; 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.

Tables I, II, III, IV, V, VI, and VII present `minasm` timings on two different machines based on x86_64 processors and a machine based on a PowerPC processor, as well as SIPE/`double` and MPFR/SIPE timing ratios. Compilation is 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.

[8] And some GCC features are currently required by SIPE for better performance. However it would still be possible to write a more portable version if need be, at least for testing purpose.

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

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.

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. One can see that the use 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 SIPE is between 2 and 6 times as fast as the use of MPFR for precision 12.

		timings (in seconds)				ratios	
args	g	`double`	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.43	8.92	2.22	2.22	5.2	4.0
1 2 6	2	0.38	8.94	2.00	1.95	5.2	4.5
1 2 6	4	0.38	8.82	2.12	2.09	5.5	4.2
1 2 6	6	0.42	8.80	2.12	2.02	4.9	4.3
1 4 6	–	5.10	66.12	16.21	16.57	3.2	4.0
1 4 6	2	6.18	67.03	15.75	16.34	2.6	4.2
1 4 6	4	5.13	63.95	16.06	15.94	3.1	4.0
1 4 6	6	5.53	64.47	16.26	15.62	2.9	4.0
1 6 5	–	0.19	1.74	0.43	0.44	2.3	4.0
1 6 5	2	0.23	1.82	0.48	0.50	2.1	3.7
1 6 5	4	0.22	1.79	0.49	0.51	2.3	3.6
1 6 5	6	0.25	1.74	0.49	0.47	1.9	3.6

TABLE I
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX MACHINE), WITH GCC 4.4.7 (DEBIAN 4.4.7-2).

There is also work in progress to use SIPE to find or prove properties on other floating-point algorithms.

V. CONCLUSION

We presented a library whose purpose is to do simple operations in binary floating-point systems in very low precisions with correct rounding to nearest, 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); [12] (a preliminary and longer version of the paper) gives a more detailed proof of the implementation. 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) would also need to be

		timings (in seconds)				ratios	
args	g	`double`	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.41	8.93	2.40	2.51	6.0	3.6
1 2 6	2	0.41	8.89	2.08	2.14	5.1	4.2
1 2 6	4	0.40	8.86	2.14	2.10	5.3	4.2
1 2 6	6	0.40	8.86	2.10	2.12	5.3	4.2
1 4 6	–	5.15	64.74	29.47	29.87	5.8	2.2
1 4 6	2	8.80	67.49	28.03	28.68	3.2	2.4
1 4 6	4	7.03	63.85	27.64	27.09	3.9	2.3
1 4 6	6	5.68	64.77	27.18	27.16	4.8	2.4
1 6 5	–	0.19	1.76	0.86	0.87	4.6	2.0
1 6 5	2	0.33	1.86	0.84	0.85	2.6	2.2
1 6 5	4	0.30	1.78	0.85	0.84	2.8	2.1
1 6 5	6	0.24	1.74	0.81	0.81	3.4	2.1

TABLE II
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX MACHINE), WITH GCC 4.6.3 (DEBIAN 4.6.3-8). TIMINGS OBTAINED WITH GCC 4.5.4 (DEBIAN 4.5.4-1) ARE VERY SIMILAR, THUS ARE NOT INCLUDED HERE.

		timings (in seconds)				ratios	
args	g	`double`	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.54	8.92	2.02	2.04	3.8	4.4
1 2 6	2	0.40	8.96	1.69	1.73	4.3	5.2
1 2 6	4	0.38	8.95	1.84	1.86	4.9	4.8
1 2 6	6	0.44	8.99	1.86	1.84	4.2	4.9
1 4 6	–	5.19	65.07	14.85	14.68	2.8	4.4
1 4 6	2	7.92	68.36	14.57	14.51	1.8	4.7
1 4 6	4	6.53	65.11	15.64	16.05	2.4	4.1
1 4 6	6	7.00	67.54	15.20	15.40	2.2	4.4
1 6 5	–	0.19	1.74	0.41	0.40	2.1	4.3
1 6 5	2	0.31	1.87	0.43	0.41	1.4	4.5
1 6 5	4	0.28	1.80	0.48	0.50	1.8	3.7
1 6 5	6	0.26	1.83	0.45	0.45	1.7	4.1

TABLE III
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX MACHINE), WITH GCC 4.7.1 (DEBIAN 4.7.1-9).

formalized. However we have also done almost-exhaustive tests of some functions in some precisions, namely the following functions have been tested against GNU MPFR on *all* `sipe_t` input values having a quantum exponent between $1-3p$ and $2p-1$, with both `SIPE_ROUND_ZOPT = 0` and `SIPE_ROUND_ZOPT = 1`:

- `sipe_add`, `sipe_sub`, `sipe_mul`, and `SIPE_2MUL` (a macro that computes a rounded product and the corresponding error term), in precisions $p = 2$ to 7 (23 080 720 tests for each function and each value of `SIPE_ROUND_ZOPT`);
- `sipe_fma` and `sipe_fms`, in precisions $p = 2$ to 7 (89 302 423 104 tests for each function and each value of

		timings (in seconds)				ratios	
args	g	double	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.50	6.45	1.99	1.99	4.0	3.2
1 2 6	2	0.41	6.79	1.64	1.69	4.1	4.1
1 2 6	4	0.43	6.80	1.66	1.68	3.9	4.1
1 2 6	6	0.48	6.85	1.71	1.73	3.6	4.0
1 4 6	–	5.20	49.66	14.94	14.87	2.9	3.3
1 4 6	2	6.99	53.30	14.27	14.48	2.1	3.7
1 4 6	4	4.78	52.75	13.35	13.55	2.8	3.9
1 4 6	6	6.74	51.90	13.48	13.40	2.0	3.9
1 6 5	–	0.20	1.37	0.42	0.41	2.1	3.3
1 6 5	2	0.25	1.48	0.41	0.42	1.7	3.6
1 6 5	4	0.20	1.49	0.41	0.42	2.1	3.6
1 6 5	6	0.23	1.40	0.38	0.38	1.7	3.7

TABLE IV
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.27 GHZ INTEL XEON E5520 (DEBIAN/UNSTABLE GNU/LINUX MACHINE), WITH GCC 4.7.1 (DEBIAN 4.7.1-9) AND LTO ENABLED (GMP AND MPFR ALSO HAD TO BE RECOMPILED WITH LTO SUPPORT).

		timings (in seconds)				ratios	
args	g	double	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.52	8.92	2.54	2.45	4.8	3.6
1 2 6	2	0.57	9.10	2.63	2.39	4.4	3.6
1 2 6	4	0.58	9.03	2.59	2.41	4.3	3.6
1 2 6	6	0.56	9.02	2.60	2.50	4.6	3.5
1 4 6	–	9.12	67.91	19.11	18.56	2.1	3.6
1 4 6	2	12.98	68.98	19.52	18.56	1.5	3.6
1 4 6	4	9.59	70.06	18.82	17.98	1.9	3.8
1 4 6	6	10.01	66.59	19.46	18.55	1.9	3.5
1 6 5	–	0.32	1.91	0.55	0.54	1.7	3.5
1 6 5	2	0.46	1.99	0.57	0.57	1.2	3.5
1 6 5	4	0.35	2.01	0.55	0.55	1.6	3.7
1 6 5	6	0.32	1.98	0.54	0.53	1.7	3.7

TABLE V
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.4GHZ AMD OPTERON 8378 (DEBIAN/6.0.5, A.K.A. SQUEEZE, GNU/LINUX MACHINE), WITH GCC 4.3.5 (DEBIAN 4.3.5-4).

		timings (in seconds)				ratios	
args	g	double	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.55	9.02	2.92	2.90	5.3	3.1
1 2 6	2	0.47	8.90	2.72	2.57	5.6	3.4
1 2 6	4	0.49	8.22	2.72	2.68	5.5	3.0
1 2 6	6	0.53	8.57	2.83	2.65	5.2	3.1
1 4 6	–	9.02	69.93	21.09	20.81	2.3	3.3
1 4 6	2	10.74	70.56	20.32	19.31	1.8	3.6
1 4 6	4	8.91	63.96	20.09	19.78	2.2	3.2
1 4 6	6	9.83	61.27	20.96	19.57	2.1	3.0
1 6 5	–	0.33	1.98	0.59	0.59	1.8	3.4
1 6 5	2	0.35	1.97	0.60	0.58	1.7	3.3
1 6 5	4	0.35	1.82	0.60	0.59	1.7	3.1
1 6 5	6	0.34	1.91	0.59	0.56	1.7	3.3

TABLE VI
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 2.4GHZ AMD OPTERON 8378 (DEBIAN/6.0.5, A.K.A. SQUEEZE, GNU/LINUX MACHINE), WITH GCC 4.4.5 (DEBIAN 4.4.5-8).

		timings (in seconds)				ratios	
args	g	double	MPFR	SIPE/0	SIPE/1	S/D	M/S
1 2 6	–	0.65	14.49	2.76	2.69	4.2	5.3
1 2 6	2	0.67	14.30	2.61	2.62	3.9	5.5
1 2 6	4	0.66	14.19	2.67	2.63	4.0	5.4
1 2 6	6	0.66	14.24	2.68	2.64	4.0	5.4
1 4 6	–	10.21	117.93	21.38	20.78	2.1	5.6
1 4 6	2	12.07	115.78	22.38	22.73	1.9	5.1
1 4 6	4	9.81	116.47	20.50	20.20	2.1	5.7
1 4 6	6	10.17	116.63	20.57	20.58	2.0	5.7
1 6 5	–	0.34	3.32	0.61	0.60	1.8	5.5
1 6 5	2	0.42	3.32	0.68	0.69	1.6	4.8
1 6 5	4	0.34	3.34	0.63	0.63	1.9	5.3
1 6 5	6	0.32	3.28	0.58	0.59	1.8	5.6

TABLE VII
TIMINGS AND RATIOS OBTAINED ON A 64-BIT 3.55GHZ POWER7 MACHINE OF THE GCC COMPILE FARM, WITH GCC 4.6.3 (RED HAT 4.6.3-2).

`SIPE_ROUND_ZOPT`);

- `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 $|i| \leq 2^p$ (1 420 068 tests for each function and each value of `SIPE_ROUND_ZOPT`);
- `sipe_eq`, `sipe_ne`, `sipe_le`, `sipe_gt`, `sipe_ge`, `sipe_lt`, `sipe_min`, `sipe_max` in precisions $p = 2$ and $p = 3$ (13 840 tests for each function and each value of `SIPE_ROUND_ZOPT`).

The above tests took a total of about 22 hours (mainly due to the time spent in the MPFR FMA/FMS functions); less than two hours were required for up to precision 6. Thanks to these tests, two obvious sign-related bugs had been found.

Future work will consist in using SIPE for other problems than the minimality of the TwoSum algorithm, and possibly improving it, depending on the needs. Another future SIPE improvement could be the support of the directed rounding attributes (which would typically be chosen at compile time). 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] 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

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

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

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

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

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