SIPE: Small Integer Plus Exponent

Vincent Lefèvre

SIPE: Small Integer Plus Exponent

Vincent Lefèvre

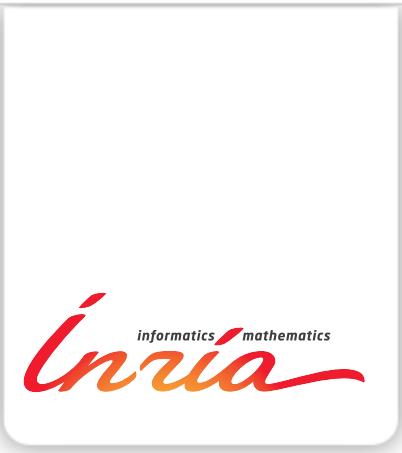
RESEARCH REPORT

N° 7832

December 2011

Project-Team Arénaire

ISSN 0249-6399 ISRN INRIA/RR--7832--FR+ENG



SIPE: Small Integer Plus Exponent

Vincent Lefèvre*

Project-Team Arénaire

Research Report n° 7832 — December 2011 — 21 pages

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. The goal of such a tool is to do proofs of algorithms/properties or computations of error bounds in these precisions, in order to generalize them to higher precisions. The supported operations are the addition, the subtraction, the multiplication, the FMA, and miscellaneous comparisons and conversions.

Key-words: low precision, arithmetic operations, correct rounding, C language

* INRIA / LIP / CNRS / ENS Lyon / UCBL / Université de Lyon, Lyon, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

SIPE: Small Integer Plus Exponent

Résumé : SIPE (Small Integer Plus Exponent) est une mini-bibliothèque sous forme de fichier d'en-tête C permettant d'effectuer des calculs à très basses précisions avec arrondi correct au plus près. Le but d'un tel outil est de faire des preuves d'algorithmes/propriétés ou des calculs de bornes d'erreur dans ces précisions, afin de les généraliser à des précisions supérieures. Les opérations supportées sont l'addition, la soustraction, la multiplication, le FMA, et diverses comparaisons et conversions.

Mots-clés : basse précision, opérations arithmétiques, arrondi correct, langage C

1 Introduction

Calculations on computers are most often done in floating-point arithmetic, as specified by the IEEE 754 standard, first published in 1985 [2] and revised in 2008 [3].

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*, $m = x_0.x_1x_2 \dots x_{p-1}$ (with $0 \leq x_i \leq \beta - 1$) is a p-digit fixed-point number called the *significand* (also improperly called *mantissa*), 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 (we say that IEEE 754 has a signed zero), 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 that will be chosen does not make the sign appear explicitly like here).

The IEEE 754 standard also specifies that when one does an arithmetic operation in one of the supported floating-point formats, the result be *correctly rounded* in one of the *rounding-direction attributes* (a.k.a. *rounding modes*), that is, the rounding of the exact result must be returned. 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.

This standard defines some fixed formats, in binary ($\beta = 2$) and in decimal ($\beta = 10$). The most common ones and most often implemented are in binary with $p = 24$ (binary32, a.k.a. single precision) and $p = 53$ (binary64, a.k.a. double precision).

For the sake of simplicity, we will assume $\beta = 2$ (future work may consider $\beta = 10$, and possibly other values). But for various reasons, one may want to do 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 inputs is proportional to 2^p, such tests will be much faster with small values of p and may still be significant to deduce 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. [5, 6][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 [1], 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.

[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 [5] 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.

Most of the code of SIPE was written in April/May 2008. Some functions were added in November 2009 in the context of [6] (though MPFR was initially used for the tests performed for this article), and several bugs were fixed in 2011 (in addition to minor changes).

Section 2 presents the basic implementation choices for SIPE. Section 3 gives the code and describes the algorithms used to implement the operations. We present results and timings in Section 4 and conclude in Section 5.

2 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 the IEEE 754-1985 round-to-nearest mode (roundTiesToEven in IEEE 754-2008). The directed 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 data.
- 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; since the only available operations are currently the addition, the subtraction and the 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 3 are valid only for much smaller values of p; the upper bound on p will depend on these algorithms. This gave the name of the library: *Small Integer Plus Exponent* (SIPE), on an idea similar to DPE[4] (meaning *Double Plus Exponent*).

[4] `https://gforge.inria.fr/projects/dpe/`

There are several conventions to define the (significand,exponent) pair. One was given at the beginning of Section 1: the integer M would represent a p-bit fixed-point number. But since M is really 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 called the *ulp* (Unit in the Last Position).[5]

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 integer overflow in some cases (0 happens to be the most practical value in the C code, but otherwise this choice is quite arbitrary). Indeed, consider the sequence $x_{i+1} = x_i^2$, with $x_0 = 0$ represented by $(M_0, E_0) = (0, 1)$. If the exact product of two numbers were computed by multiplying the significands and adding the exponents and the subsequent rounding left 0 untouched, then one would get $M_i = 0$ and $E_i = 2^i$, yielding an integer overflow on E_i after several iterations. And even though the final result would not depend on E_i, the undefined behavior could have unwanted side effects in practice (e.g., because an integer overflow may generate an exception or because the code may be transformed in an uncontrolled manner, due to optimizations based on the fact that an undefined behavior is forbidden thus cannot occur).

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 may greatly depend on the C implementation.

- 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, we need to detect it for each operation (and IEEE 754 provides no exceptions for ties, so that this is very difficult to detect).

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. These functions are described in the following section.

[5] 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 term *ulp*.

3 The `sipe.h` Header File

Since the code is quite short and most of it is needed for the description and proof of the implementation, we have chosen to include it entirely (without most comments, as they would be redundant here). This allows us to give a full description of the implementation. We give comments and proofs *after* each block of code. The latest version can be found at:

`http://www.vinc17.net/software/sipe.h`

```
#ifndef __GNUC__
#error "GCC is required"
#endif
```

We require the GCC compiler: indeed, for the efficiency, we will use some GCC builtin functions: `__builtin_expect` (to provide information on branch probabilities) and builtins of the `__builtin_clz` family (to determine the position of the most significant bit 1, for rounding). We will also use the fact that a right shift on a negative integer (which is implementation-defined [4, Section 6.5.7]) is done with sign extension.[6] But the code could be adapted for other compilers.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <inttypes.h>
```

These are the standard ISO C99 headers we need.

```
#ifndef SIPE_MINSIZE
#define SIPE_MINSIZE 32
#endif

#if SIPE_MINSIZE != 32 && SIPE_MINSIZE != 64
#error "SIPE_MINSIZE must be 32 or 64"
#endif

#define SIPE_INT_TYPE_AUX(S) int_fast ## S ## _t
#define SIPE_INT_TYPE(S) SIPE_INT_TYPE_AUX(S)

typedef SIPE_INT_TYPE(SIPE_MINSIZE) sipe_int_t;
typedef int_fast8_t sipe_exp_t;
```

SIPE can work either with integers M (integral significand) having at least 32 bits (the default) or with integers having at least 64 bits (if the code is compiled with `-DSIPE_MINSIZE=64`). Of course, the latter allows the chosen precision to be larger, but it will be slower on 32-bit machines. We just tell the compiler to use what it believes to be the fastest type satisfying this constraint on the integer size; the exact size of the type will just matter to determine the upper bound on the allowed precisions (see below). It is a way to favor the performance over the allowed precisions.

Similarly, we just require the exponent to fit on 8 bits (including its sign): with the supported operations and the low precision, there should be no need for large exponents. The algorithms

[6] `http://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html`

and the code will still work with other signed integer types, so that it could be modified if need be. Moreover, the exact type size is chosen by the compiler, and will not matter either in the following. The only requirement is that no integer overflows occur in the functions below. More will be said below for the concerned functions, but in short, if $E_{\max}$ denotes the maximum value of the `sipe_exp_t` type, then if the exponent E of each input satisfies $|E| \leq E_{\max}/3$, no integer overflows on the exponents will occur.

```
#define SIPE_TYPESIZE(T) (sizeof(T) * CHAR_BIT)
#define SIPE_SIZE SIPE_TYPESIZE(sipe_int_t)
```

Here we determine the size (or bit-width) S of the `sipe_int_t` type (which will hold the integers M), so that if the real size is larger than the minimum requirement, one can choose any precision allowed for this size (not just those allowed for the minimum requirement), as said above.

```
#define SIPE_ONE ((sipe_int_t) 1)
#define SIPE_TWO_TO(N) (SIPE_ONE << (N))
```

`SIPE_ONE` is the value 1 with the type used for the integral significand, and `SIPE_TWO_TO(N)` evaluates to 2^N with the type of the integral significand (by shifting 1 by N positions to the left).

```
#if SIPE_USE_FMA
#define SIPE_PREC_MAX ((SIPE_SIZE - 2) / 3)
#else
#define SIPE_PREC_MAX ((SIPE_SIZE - 2) / 2)
#endif
```

The upper bound on the precision is determined from the size of the type used for the integral significand. We have $p_{\max} = \lfloor (S-2)/3 \rfloor$ if the FMA is allowed, $p_{\max} = \lfloor (S-2)/2 \rfloor$ otherwise. These formulas are deduced from the algorithms described below.

```
#define SIPE_LIKELY(C) (__builtin_expect (!!(C), 1))
#define SIPE_UNLIKELY(C) (__builtin_expect (!!(C), 0))
```

For some tests in the code below, we know whether the result is true with a very high probability or a very low probability. We will give this information to the compiler with the macros `SIPE_LIKELY` and `SIPE_UNLIKELY`.

```
#define SIPE_ABSINT(X) ((X) >= 0 ? (X) : - (X))
```

This macro computes the absolute value of a number X: X if $X \geq 0$, $-X$ otherwise. It will be applied only on integral significands, which cannot take the minimum value of the type (due to the constraints on the precision). Thus it is guaranteed that the subtraction `- (X)` will not overflow[7], and this macro is correct.

[7] In ISO C, the range of signed integers is of the form either $[-T, T]$ (never seen in practice) or $[-(T+1), T]$. If $X = -(T+1)$, then $-X = T+1$ is not representable, and only in this case.

```
static inline int sipe_clz (sipe_int_t i)
{
  if (SIPE_TYPESIZE (int) >= SIPE_SIZE)
    return __builtin_clz ((unsigned int) i)
      - (SIPE_TYPESIZE (int) - SIPE_SIZE);

  if (SIPE_TYPESIZE (long) >= SIPE_SIZE)
    return __builtin_clzl ((unsigned long) i)
      - (SIPE_TYPESIZE (long) - SIPE_SIZE);

  if (SIPE_TYPESIZE (long long) >= SIPE_SIZE)
    return __builtin_clzll ((unsigned long long) i)
      - (SIPE_TYPESIZE (long long) - SIPE_SIZE);

  fprintf (stderr, "sipe: unsupported sipe_int_t size for sipe_clz");
  exit (119);
}
```

This function `sipe_clz` computes the number of leading zeros of an integer of type `sipe_int_t` (integral significand).[8]

```
typedef struct {
  sipe_int_t i;
  sipe_exp_t e;
} sipe_t;
```

This is the structure consisting of two native signed integers: if `X` is an object of type `sipe_t` representing a number, `X.i` contains the integral significand M and `X.e` contains the exponent E: the value of `X` is $M \cdot 2^E$. We will assume that except for temporary results, the numbers are normalized, i.e. either $M = 0$ or $2^{p-1} \leq |M| \leq 2^p - 1$, and if $M = 0$, then $E = 0$.

[8] We assume that the integers have no *padding bits* [4, Section 6.2.6.2], which is true with GCC.

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

The purpose of this macro is to round a temporary variable `X` (of type `sipe_t`) to precision `PREC` and normalize it. Of course, this means that the value M of its significand field `X.i` is not required to satisfy the condition $M = 0$ or $2^{p-1} \leq |M| \leq 2^p - 1$. We just assume that $|M| < 2^{S-1}$, where S is the bit-width of the `sipe_int_t` type, as defined above; this condition will be checked each time this macro `SIPE_ROUND` is used in the code below. And if $M = 0$, the exponent field `X.e` may have any value. Note that this is a macro and not an inline function since the choice was to modify the variable `X` *in place*.

This macro works in the following way. First, if $M = 0$, we just need to set the exponent field `X.e` to 0. Now assume that $M \neq 0$. We will work mainly on its absolute value $|M|$, stored in `_i`. Since $|M| < 2^{S-1}$, it is representable in the `sipe_int_t` type. Then we compute the difference between the precision p of the floating-point system and the size (in bits) of $|M|$. If this difference is zero, then we do not need to do anything. If this difference is strictly positive, i.e. if $|M| < 2^{p-1}$,

then we normalize M. If this difference is strictly negative, then we need to round the value, which is done in the following way. To round the absolute value $|M|$ of the significand, we use the formula $j = \lfloor (j_0 + c)/2 \rfloor$, where j_0 is $|M|$ truncated on $p+1$ bits (with a right-shift) and $c = 1$, 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 $c = 0$. Note: without this case $c = 0$, one would get 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, 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.

Now that we have this rounding macro, the operations will be implemented in the following way. Special cases, for which we can return a result almost immediately, and handled separately, and the generic cases are computed exactly, then rounded and normalized with `SIPE_ROUND`.

```
static inline sipe_int_t sipe_to_int (sipe_t x)
{
  return x.e < 0 ? x.i >> - x.e : x.e > 0 ? x.i << x.e : x.i;
}
```

This function `sipe_to_int` converts a `sipe_t` value into a `sipe_int_t` integer. The value is required to be an integer representable in a `sipe_int_t`, so that there is not any rounding.

Let us prove that every operation of the `return` statement is well-defined in C and does what we want. The constraints and semantics of the bitwise shift operators `>>` and `<<` are specified by the ISO C99 standard [4, Section 6.5.7]. For $E < 0$, $-E$ is well-defined if and only if $|E| \leq E_{\max}$, i.e. $E \neq -E_{\max} - 1$. If $M = 0$, then $E = 0$, and `x.i` is returned directly. If $M \neq 0$, then we have the following two cases:

- If $E < 0$, then $M/2^{-E}$ must be a non-zero integer, thus larger than or equal to 1, so that $-E$ is necessarily less than the width of `sipe_int_t`, implying that the right shift is not undefined. If $M > 0$, then the right shift computes $\lfloor M/2^{-E} \rfloor$. If $M < 0$, then the right shift is implementation-defined, but we depend on GCC, and GCC documents that negative numbers are represented in two's complement and that the right shift acts on negative numbers by sign extension, so that it still computes $\lfloor M/2^{-E} \rfloor$. Since $M/2^{-E}$ is an integer (as a requirement), the returned value is $M/2^{-E}$.
- If $E > 0$, then we have $|x| = |M| \cdot 2^E \geq 2^E$, which is required to be representable in a `sipe_int_t`. Thus E is necessarily less than the width of `sipe_int_t`, implying that the left shift is well-defined and computes a multiplication by 2^E. Therefore the returned value is $M \cdot 2^E$.

Thus the expression in the `return` statement computes $M/2^{-E}$ if $E < 0$, $M \cdot 2^E$ if $E > 0$, and M if $E = 0$, i.e. it computes $M \cdot 2^E$ for any value of E, i.e. it returns the value of x (as an integer), as required.

```
#ifdef __GNU_MP_VERSION
static inline void sipe_to_mpz (mpz_t z, sipe_t x)
{
  mpz_set_si (z, x.i);
  if (x.e < 0)
    mpz_tdiv_q_2exp (z, z, - x.e);
  else
    mpz_mul_2exp (z, z, x.e);
}
#endif
```

This function `sipe_to_mpz` converts a `sipe_t` value into an `mpz_t` integer from the GMP arbitrary-precision library[9], rounding the input value (thus which need not be an integer) to the nearest integer toward zero (truncation). This function is defined only if the `gmp.h` header file has been included before. The only requirements, implied by the code, are that $|E|$ must fit in a `sipe_exp_t` (i.e. $E \neq -E_{\max} - 1$) and in an `mp_bitcnt_t` (which is an `unsigned long`, at least up to GMP 5).

```
#define SIPE_DEFADDSUB(OP,ADD,OPS)                                      \
  static inline sipe_t sipe_##OP (sipe_t x, sipe_t y, int prec)         \
  {                                                                     \
    sipe_exp_t delta = 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) || delta > prec + 1)                   \
      return x;                                                         \
    if (delta < - (prec + 1))                                           \
      return (ADD) ? y : (sipe_t) { - y.i, y.e };                       \
    r = delta < 0 ?                                                     \
      ((sipe_t) { (x.i) OPS (y.i << - delta), x.e }) :                  \
      ((sipe_t) { (x.i << delta) OPS (y.i),   y.e });                   \
    SIPE_ROUND (r, prec);                                               \
    return r;                                                           \
  }

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

This defines the functions `sipe_add` and `sipe_sub`, to perform respectively an addition and a subtraction of two `sipe_t` numbers. The difference of the exponents is required to be representable in a `sipe_exp_t`. The particular cases are handled first:

- If $x = 0$, we return y for the addition, $-y$ for the subtraction.
- If $y = 0$, we return x.

[9] `http://gmplib.org/`

- If $E_x - E_y > p + 1$, 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 $|r - x| = |y| < 2^{E_r-1} = \frac{1}{2}\,\mathrm{ulp}(r)$. Thus we return x.
- If $E_y - E_x > p + 1$, 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, $|E_x - E_y| \leq p + 1$, so that with the requirement on the precision, we can compute $x \pm y$ exactly without an integer overflow, then round the result with `SIPE_ROUND`.

```
static inline sipe_t sipe_neg (sipe_t x, int prec)
{
  return (sipe_t) { - x.i, x.e };
}
```

This function `sipe_neg` computes the opposite of a `sipe_t` number by taking the opposite of its significand. There are no specific requirements.

```
static inline sipe_t sipe_set_si (sipe_int_t x, int prec)
{
  sipe_t r = { x, 0 };
  SIPE_ROUND (r, prec);
  return r;
}
```

This function `sipe_set_si` converts an integer x of type `sipe_int_t` into a `sipe_t` number. The value is $x \cdot \beta^0$, so that we just need to set the exponent to 0 and normalize the result with `SIPE_ROUND`.

Note: if the input integer is too large for the precision, `SIPE_ROUND` will also round it correctly. However, the main intent of this function is to initialize a `sipe_t` number from a simple (exactly representable) integer, not to round a non-exactly representable integer (see below).

```
static inline sipe_t sipe_add_si (sipe_t x, sipe_int_t y, int prec)
{
  sipe_t r = { y, 0 };
  SIPE_ROUND (r, prec);
  return sipe_add (x, r, prec);
}
```

This function `sipe_add_si` adds a simple integer y to a `sipe_t` number. It is actually a combination of `sipe_set_si` and `sipe_add`; thus this function is not really necessary, but just allows one to write more readable code.

Note: If the integer y is not exactly representable in the target precision, two roundings will occur, which may yield a result different from the one obtained with a single rounding. This is *not* a bug, as the purpose of this function is not to provide a mixed float-integer operation. Supporting such an operation would have made the code more complex, therefore slower, and it was not needed in practice.

The `SIPE_ROUND` macro is used here to normalize the intermediate result, as required by `sipe_add` (from the above note, it will not actually do any rounding under normal conditions).

```
static inline sipe_t sipe_sub_si (sipe_t x, sipe_int_t y, int prec)
{
  sipe_t r = { y, 0 };
  SIPE_ROUND (r, prec);
  return sipe_sub (x, r, prec);
}
```

This function `sipe_sub_si` subtracts a simple integer y from a `sipe_t` number. It is actually a combination of `sipe_set_si` and `sipe_sub`. This function is similar to `sipe_add_si`; see this function above for the description.

```
#define SIPE_DEFNEXT(DIR,OPS,OPN)                                       \
  static inline sipe_t sipe_next##DIR (sipe_t x, int prec)              \
  {                                                                     \
    sipe_t r;                                                           \
    if (SIPE_UNLIKELY (x.i == OPN (SIPE_TWO_TO (prec - 1))))            \
      {                                                                 \
        r.i = OPN (SIPE_TWO_TO (prec) - 1);                             \
        r.e = x.e - 1;                                                  \
      }                                                                 \
    else                                                                \
      {                                                                 \
        r.i = x.i OPS 1;                                                \
        r.e = x.e;                                                      \
        SIPE_ROUND (r, prec);                                           \
      }                                                                 \
    return r;                                                           \
  }

SIPE_DEFNEXT(above,+,-)
SIPE_DEFNEXT(below,-,+)
```

This defines the functions `sipe_nextabove` and `sipe_nextbelow`, which respectively return the next representable floating-point number above and below the input number. Because the exponent range is regarded as unbounded,[10] the behavior is not defined for the input number zero. Apart from this particular input, these functions correspond to the nextUp and nextDown operations from the IEEE 754-2008 standard.

These functions are implemented in the following way. The operation consists in adding or subtracting one ulp to/from the input value, except when the exponent would decrease. So, one first tests whether this particular case occurs: M is equal to -2^{p-1} for `sipe_nextabove`, and to $+2^{p-1}$ for `sipe_nextbelow`. In such a case, the integral significand of the returned value is $\pm(2^p - 1)$ (i.e. the maximum significand value, with the correct sign), and its quantum exponent is $E - 1$. Otherwise we just need to add/subtract 1 to/from M, then normalize the result in case this operation gave a significand of the form $\pm 2^p$ (the significand will be set to $\pm 2^{p-1}$ and the exponent will be incremented).

[10] Actually, there are implementation bounds, but this is a limitation: there are no associated underflow and overflow exceptions.

```
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;
}
```

This function `sipe_mul` computes the product of two `sipe_t` numbers. It is implemented in the following way: the significands M_x and M_y are multiplied without overflow as $p \leq (S-1)/2$: $M = M_x \cdot M_y$ with $|M| < 2^{S-1}$; the exponents E_x and E_y are added; then the result is rounded and normalized. The sum of the exponents is required to be representable in a `sipe_exp_t`; this should not be a problem in practice, in particular because `SIPE_ROUND` ensures that the exponent field of zero is 0 (without that, the implementation would be incorrect because successive multiplications involving zeros could end up with an integer overflow on the exponents).

Note: Since we have chosen the convention of integral significands (instead of significands of the form $x_0.x_1x_2\ldots x_{p-1}$) and these significands are represented by integers (`sipe_int_t` type), we do not need any correction on the exponent before rounding.

```
static inline sipe_t sipe_mul_si (sipe_t x, sipe_int_t y, int prec)
{
  sipe_t r = { y, 0 };
  SIPE_ROUND (r, prec);
  return sipe_mul (x, r, prec);
}
```

This function `sipe_mul_si` computes the product of a `sipe_t` number by a simple integer, which is required to be exactly representable in the floating-point system (to avoid two roundings). It is actually a combination of `sipe_set_si` and `sipe_mul`. See `sipe_add_si` for a more detailed description.

```
#if SIPE_USE_FMA

#define SIPE_DEFFMAFMS(OP,FMA,OPS)                                          \
  static inline sipe_t sipe_##OP (sipe_t x, sipe_t y, sipe_t z,             \
                                  int prec)                                 \
  {                                                                         \
    sipe_t r;                                                               \
    sipe_exp_t delta;                                                       \
                                                                            \
    r.i = x.i * y.i;                                                        \
    if (SIPE_UNLIKELY (r.i == 0))                                           \
      return (FMA) ? z : (sipe_t) { - z.i, z.e };                           \
    r.e = x.e + y.e;                                                        \
    if (SIPE_UNLIKELY (z.i == 0))                                           \
      {                                                                     \
        SIPE_ROUND (r, prec);                                               \
        return r;                                                           \
      }                                                                     \
    delta = r.e - z.e;                                                      \
    if (delta > prec)                                                       \
      {                                                                     \
        /* Warning! The sign of z.i is important if r is the                \
           middle of two consecutive machine numbers. */                   \
        r.i = 2 * r.i OPS (z.i < 0 ? -1 : 1);                               \
        r.e--;                                                              \
        SIPE_ROUND (r, prec);                                               \
        return r;                                                           \
      }                                                                     \
    if (delta < - (2 * prec + 1))                                           \
      return (FMA) ? z : (sipe_t) { - z.i, z.e };                           \
    r = delta < 0 ?                                                         \
      ((sipe_t) { (r.i) OPS (z.i << - delta), r.e }) :                      \
      ((sipe_t) { (r.i << delta) OPS (z.i),   z.e });                       \
    SIPE_ROUND (r, prec);                                                   \
    return r;                                                               \
  }

SIPE_DEFFMAFMS(fma,1,+)
SIPE_DEFFMAFMS(fms,0,-)

#endif
```

This defines the functions `sipe_fma` and `sipe_fms`, respectively computing the fused multiply-add $xy + z$ and the fused multiply-subtract $xy - z$, i.e. with a single rounding. They require a smaller precision bound than the other functions: $p_{\max} = \lfloor (S-2)/3 \rfloor$ instead of $\lfloor (S-2)/2 \rfloor$.

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 $p_{\max} = \lfloor (S - 2)/3 \rfloor$.

```
static inline int sipe_eq (sipe_t x, sipe_t y)
{
  return x.i == y.i && x.e == y.e;
}

static inline int sipe_ne (sipe_t x, sipe_t y)
{
  return x.i != y.i || x.e != y.e;
}
```

These Boolean functions `sipe_eq` and `sipe_ne` tells whether two `sipe_t` numbers are equal or different. Two numbers are equal if and only if they are identical.[11] Moreover, since `sipe_t` numbers are normalized, the representation is unique; thus two numbers are equal if and only if their significands are equal and their exponents are equal, which is what these functions test.

[11] This is not true for IEEE 754 floating-point systems because of NaN and signed zeros, but we do not have such particular data here.

```
#define SIPE_DEFCMP(OP,TYPE,E,L,G)                                      \
  static inline TYPE sipe_##OP##pos (sipe_t x, sipe_t y)                \
  {                                                                     \
    if (x.e < y.e)  return (L);                                         \
    if (x.e > y.e)  return (G);                                         \
    return ((E) ? x.i < y.i : x.i <= y.i) ? (L) : (G);                  \
  }                                                                     \
  static inline TYPE sipe_##OP##neg (sipe_t x, sipe_t y)                \
  {                                                                     \
    if (x.e < y.e)  return (G);                                         \
    if (x.e > y.e)  return (L);                                         \
    return ((E) ? x.i < y.i : x.i <= y.i) ? (L) : (G);                  \
  }                                                                     \
  static inline TYPE sipe_##OP (sipe_t x, sipe_t y)                     \
  {                                                                     \
    if ((E) && x.i == 0 && y.i == 0)  return (G);                       \
    if (x.i <= 0 && y.i >= 0)         return (L);                       \
    if (x.i >= 0 && y.i <= 0)         return (G);                       \
    return x.i < 0 ?                                                    \
      sipe_##OP##neg (x, y) :                                           \
      sipe_##OP##pos (x, y);                                            \
  }

SIPE_DEFCMP(le,int,0,1,0)
SIPE_DEFCMP(lt,int,1,1,0)
SIPE_DEFCMP(ge,int,1,0,1)
SIPE_DEFCMP(gt,int,0,0,1)
SIPE_DEFCMP(min,sipe_t,0,x,y)
SIPE_DEFCMP(max,sipe_t,0,y,x)
```

This defines the functions `sipe_le`, `sipe_lt`, `sipe_ge`, `sipe_gt`, `sipe_min`, and `sipe_max`. The first four ones return a Boolean integer (0 for false, 1 for true) corresponding to the relations $\leq$, $<$, $\geq$, $>$ between two `sipe_t` numbers. The functions `sipe_min` and `sipe_max` respectively return the minimum and the maximum of two `sipe_t` numbers.

The implementation is straightforward. The first three tests capture the cases where $x = 0$ and/or $y = 0$, and the cases where x and y have opposite signs. The `sipe_*_neg` function handles the case $x < 0$ and $y < 0$. The `sipe_*_pos` function handles the case $x > 0$ and $y > 0$.

```
#define SIPE_DEFCMPMAG(OP,X,Y)                                          \
  static inline sipe_t sipe_##OP##mag (sipe_t x, sipe_t y)              \
  {                                                                     \
    sipe_int_t absxi, absyi;                                            \
    if (x.i == 0)       return X;                                       \
    if (y.i == 0)       return Y;                                       \
    if (x.e < y.e)      return X;                                       \
    if (x.e > y.e)      return Y;                                       \
    absxi = SIPE_ABSINT (x.i);                                          \
    absyi = SIPE_ABSINT (y.i);                                          \
    if (absxi < absyi)  return X;                                       \
    if (absxi > absyi)  return Y;                                       \
    return x.i < 0 ? X : Y;                                             \
  }

SIPE_DEFCMPMAG(min,x,y)
SIPE_DEFCMPMAG(max,y,x)
```

This defines the functions `sipe_minmag` and `sipe_maxmag`, which correspond to the IEEE 754-2008 minNumMag and maxNumMag operations and are defined as follows: `sipe_minmag` (resp. `sipe_maxmag`) returns the input value that has the smaller (resp. larger) absolute value, and if these absolute values are equal, it returns the minimum (resp. maximum) input value.

We consider the cases $x = 0$ and $y = 0$ first (minimum absolute value). Then the absolute values are ordered by the exponents. And if the exponents are equal, the absolute values are ordered by the absolute values of the significands. The last line corresponds to $|x| = |y|$.

```
static inline int sipe_cmpmag (sipe_t x, sipe_t y)
{
  sipe_int_t absxi, absyi;
  if (x.i == 0 && y.i == 0)
    return 0;
  if (x.i == 0)
    return -1;
  if (y.i == 0)
    return 1;
  if (x.e < y.e)
    return -1;
  if (x.e > y.e)
    return 1;
  absxi = SIPE_ABSINT (x.i);
  absyi = SIPE_ABSINT (y.i);
  if (absxi < absyi)
    return -1;
  if (absxi > absyi)
    return 1;
  return 0;
}
```

This function `sipe_cmpmag` returns $\mathrm{sign}(|x| - |y|)$, i.e. -1 if $|x| < |y|$, $+1$ if $|x| > |y|$, and 0 if $|x| = |y|$. The implementation is similar to the one of `sipe_minmag` and `sipe_maxmag`.

```
static inline void sipe_outbin (FILE *stream, sipe_t x, int prec)
{
  sipe_int_t bit;

  if (x.i == 0)
    {
      putc ('0', stream);
      return;
    }
  else if (x.i < 0)
    {
      putc ('-', stream);
      x.i = - x.i;
    }

  fputs ("1.", stream);
  for (bit = SIPE_TWO_TO (prec - 2); bit != 0; bit >>= 1)
    putc (x.i & bit ? '1' : '0', stream);
  fprintf (stream, "e%" PRIdFAST8, x.e + prec - 1);
}
```

This function `sipe_outbin` outputs a `sipe_t` number written in binary. For non-zero numbers, after the possible minus sign, one has the significand of the form $1.x_1x_2\ldots x_{p-1}$ (since $x_0 \neq 0$, $x_0 = 1$), followed by the character `e`, followed by the binary exponent written in decimal.

4 Results and Timings

In [5, 6], the TwoSum algorithm in radix 2 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 here is an old version from 2009 and contains bugs (`minasm.c` is not affected by these bugs, though).

Table 1 presents `minasm.c` timings and ratios on two different machines, respectively using the `double` native floating-point type (IEEE 754 double-precision, i.e. 53 bits, in hardware), MPFR in precision 12, and SIPE in precision 12 (chosen at compile time). Note that these timings include the overhead for the tests of results; the real ratios are probably significantly higher. But these are real-world timings. One can see that the use of SIPE is between 2 and 6 times as slow as the use of `double` (but the test on `double` did not allow us to deduce minimality results for precisions up to 11, so that an arbitrarily-low precision library was really needed), mostly around 5 times as slow. And the use of SIPE is between 2 and 4 times as fast as the use of MPFR for precision 12, mostly around 3 times as fast.

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

		timings (in seconds)			ratios	
machine	args	`double`	MPFR/12	SIPE/12	S/D	M/S
AMD Opteron	1 2 6	0.55	7.90	2.91	5.3	2.7
	1 4 6	9.11	62.78	20.96	2.3	3.0
	1 6 5	0.31	1.85	0.59	1.9	3.1
Intel Xeon	1 2 6	0.41	8.97	2.40	5.9	3.7
	1 4 6	5.15	64.91	29.45	5.7	2.2
	1 6 5	0.19	1.80	0.86	4.5	2.1

Table 1: Timings and ratios obtained on two different machines, respectively using the `double` native floating-point type (53-bit precision, in hardware), MPFR in precision 12, and SIPE in precision 12 (chosen at compile time). The SIPE/`double` and MPFR/SIPE timing ratios are given in the last two columns.

5 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), but in order to let the paper readable, we did not prove every part of the implementation in detail. It is almost impossible to say whether a proof would be complete or not anyway, except by writing a formal proof, where the C language (including the preprocessor, since we quite heavily rely on it) would also need to be 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* input values having a quantum exponent between $1 - 3p$ and $2p - 1$:

- `sipe_add`, `sipe_sub`, `sipe_fma`, `sipe_fms` in precisions $p = 2$ to 7;
- `sipe_eq`, `sipe_ne`, `sipe_le`, `sipe_gt`, `sipe_ge`, `sipe_lt`, `sipe_min`, `mpfr_max` in precisions $p = 2$ and $p = 3$.

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 representation and a complete rewrite.

References

[1] 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*, 33(2), June 2007.
`http://doi.acm.org/10.1145/1236463.1236468`

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

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

[4] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[5] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly-rounded sums. Research report RR-7262, INRIA, Lyon, France, April 2010. http://www.vinc17.net/research/papers/rr_ccrsums2.pdf

[6] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly-rounded sums. *IEEE Transactions on Computers*, 2011. To appear. http://dx.doi.org/10.1109/TC.2011.27

[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*. Birkhäuser Boston, first edition, 2010. 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. Technical Report 22, RC-Informatie, Technishe Hogeschool Eindhoven, 1968.

[9] G. W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. Technical Report 21, RC-Informatie, Technishe Hogeschool Eindhoven, 1969.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399