Aarith

Release 1.0

Marcel Brand, Oliver Keszcoze, Michael Witterauf

Feb 27, 2022

GENERAL

Aarith is a header-only, arbitrary precision number library for C++. It is intended to be used as a drop-in replacement of the native data types.

Aarith currently supports

- IEEE 754 like floating-point numbers of arbitrary bit-width for both, the exponent and the mantissa
- Two's complement integers of arbitrary bit-width (signed and unsigned)

CHAPTER

ONE

INSTALLATION

Copy the contents of this repository into a <destination> folder and include Aarith in your CMake build using:

```
add_subdirectory(<destination>)
```

You can then link your target <targetname> against Aarith with:

```
target_link_libraries(<targetname> PUBLIC aarith::Library)
```

1.1 Requirements/Dependencies

Aarith is intended to be used without introducing further dependencies. You can use it immediately without having to further software.

1.1.1 Tests

If you want to run the tests, you need catch2, the Google benchmark library.

If you want to run the tests against other number libraries, you need to install MPFR and MPIR.

1.1.2 Documentation

The documentation is [available online](add link!). If you want to build it locally, you need Python, Sphinx, the readthedocs Theme, breathe and Doxygen.

CHAPTER

TWO

QUICK START

Aarith is intended to be used as a drop-in replacement for the native data types. You only need to include the headers and can start using Aarith immediately:

```
#include <aarith/float.hpp>
#include <aarith/integer.hpp>

int main()
{
    using namespace aarith;
    uint64_t a = 10, b = 20;
    uinteger<64> a_ = 10, b_ = 20;
    std::cout << "a + b = " << (a + b) << "\n";
    std::cout << "a_+ b_ = " << (a_ + b_) << "\n";

    float x=3.0F, y=2.5F;
    floating_point<8,23> x_{3.0F}, y_{2.5F};
    std::cout << "x + y = " << (x * y) << "\n";
    std::cout << "x_+ y_ = " << (x_ * y_) << "\n";
}
```

This gives the expected output of

```
$ ./arithmetic_example
a + b = 30
a_+ b_ = 30
x * y = 7.5
x_* y_ = 7.5
```

Hint: To make usage of Aarith more convenient, the following type aliases are shipped with Aarith:

```
using half_precision = floating_point<5, 10, uint64_t>;
using single_precision = floating_point<8, 23, uint64_t>;
using double_precison = floating_point<11, 52, uint64_t>;
using quadruple_precision = floating_point<15, 112, uint64_t>;
using bfloat16 = floating_point<8, 7, uint64_t>;
using tensorfloat32 = floating_point<8, 10, uint64_t>;
```

Further examples for how to use aarith can be found at the *Uses Cases* and in the *examples* and *experiments* source code folders (the tests can also give a good idea of how to use aarith).

We also refer the interested user to [Keszocze2021].

CHAPTER

THREE

USE CASES

In the following, use cases for Aarith will be shown. (So far, we only present one)

3.1 The FAU Adder

The FAU Adder (see [Echavarria2016]) splits the operands into a most-significant part (MSP) and least-significant part (LSP) and cuts the carry chain in between these parts. To reduce the error, some bits of the LSP (`shared_bits` in total) are used the predict the carry.

The following code implements the FAU adder. This piece of code is intended to show how easily individual bits ca be accessed when designing hardware units.

```
template <size_t width, size_t lsp_width, size_t shared_bits = 0>
uinteger<width + 1> FAUadder(const uinteger<width>& a, const uinteger<width>& b)
{

    // make sure that the parameters
    static_assert(shared_bits <= lsp_width);
    static_assert(lsp_width < width);
    static_assert(lsp_width > 0);

    //************************
    // Extract MSP and LSP
    // can't use structured binding (i.e. `` auto [msp, lsp] = split<lsp_index>(a)``) as msp and lsp need
    // not bee of same width
    constexpr size_t lsp_index = lsp_width - 1;
    const auto a_split = split<lsp_index>(a);
    const auto b_split = split<lsp_index>(b);

    const uinteger<lsp_width> a_lsp = a_split.second;
    const uinteger<lsp_width> b_lsp = b_split.second;

    constexpr size_t msp_width = width - lsp_width;
    const uinteger<msp_width> a_msp = a_split.first;
    const uinteger<msp_width> b_msp = b_split.first;
    //************************

    // sum up LSP including the computation of the carry
    uinteger<lsp_width + 1> lsp_sum = expanding_add(a_lsp, b_lsp);
```

(continues on next page)

(continued from previous page)

```
    // remove the carry for later use
    uinteger<lsp_width> lsp = width_cast<lsp_width>(lsp_sum);

    bool predicted_carry = false;
    // conditionally perform carry prediction
    if constexpr (shared_bits > 0)
    {
        // extract the shared bit of both operands
        uinteger<shared_bits> a_shared = bit_range<lsp_index, lsp_index - (shared_bits - 1)>(a);
        uinteger<shared_bits> b_shared = bit_range<lsp_index, lsp_index - (shared_bits - 1)>(b);

        // compute the carry
        uinteger<shared_bits + 1> shared_sum = expanding_add(a_shared, b_shared);

        predicted_carry = shared_sum.msb();
    }

    // if there was a carry but we did not predict one (i.e. it wasn't used in the MSP)
    // we need to perform an all1 error correction
    if (lsp_sum.msb() && !predicted_carry)
    {
        lsp = lsp.all_ones();
    }

    // finally put MSP and LSP together
    const uinteger<msp_width + 1> msp = expanding_add(a_msp, b_msp, predicted_carry);

    uinteger<width + 1> result{lsp};

    const auto extended_msp = width_cast<width + 1>(msp);
    result = add(result, extended_msp << lsp_width);
    return result;
}
```

CHAPTER

FOUR

AARITH'S PHILOSOPHY

Aarith is build with the following points in mind:

- **Aarith is should easy to use:**
  - e.g. no functions such as `u_add_32(num_a, num_b)`, just plain `num_a + num_b`
  - easily define new number formats, e.g. `using bfloat16 = floating_point<8, 7, uint64_t>;` (see [Burgess2019])
- **Aarith should allow for easy access to the individual bits of the stored number:**
  - for the design of new hardware units (e.g., implementing arithmetic units such as the FAU Adder [Echavarria2016], *see example*
  - for debugging unexpected results
- Aarith should be publicly available

4.1 Easy to use

We want Aarith to blend in with C++. That is, we want it to provide the usual operations, such as + or <<.

The following program computes the sum of `1.0` and `2.0` using 200 digits using the MPFR library.

```
#include <stdio.h>

#include <gmp.h>
#include <mpfr.h>

int main (void)
{
  unsigned int i;
  mpfr_t s, t, u;

  mpfr_init2 (t, 200);
  mpfr_set_d (t, 1.0, GMP_RNDD);
  mpfr_init2 (s, 200);
  mpfr_set_d (s, 2.0, GMP_RNDD);
  mpfr_init2 (u, 200);
  mpfr_add (s, s, u, GMP_RNDD);
  printf ("Sum is ");
  mpfr_out_str (stdout, 10, 0, s, GMP_RNDD);
  putchar ('\n');
```

(continues on next page)

(continued from previous page)

```
  mpfr_clear (s);
  mpfr_clear (t);
  mpfr_clear (u);
  return 0;
}
```

The equivalent program using Aarith looks like this (the parameters for the exponent and mantissa width, i.e. `E` and `M`, need to be chosen large enough to fit a 200 digits number):

```
#include <aarith/float.hpp>
#include <aarith/integer.hpp>

int main()
{
    floating_point<E,M> x_{1.0F}, y_{2.0F};
    std::cout << "Sum is " << (x + y) << "\n";

}
```

4.2 No Surprises!

Aarith performs very little implicit type conversions. Most of the constructors are `explicit`. Especially, Aarith does not use `typedef`'s involving native data types. This is motivated by the following situation. Consider the following program:

```
uint8_t u8 = 42; uinteger<8> aarith_u8{u8};
std::cout << "uint8_t=" << u8
    << " (as int=" << int{u8} << ")"
    << " aarith::uinteger<8>="
    << aarith_u8 << "\n";
```

Running this program gives the following output:

```
$ ./output_example
uint8_t=* (as int=42) aarith::uinteger<8>=42
```

The asterisk * most likely was not was the user was expecting to see. Such a conversion is *never* carried out by Aarith.

4.3 Speed

Aarith is *not* extensively optimized for speed! There are other libraries for that. If raw speed is your goal, try MPFR and MPIR.

CHAPTER

FIVE

LITERATURE

CHAPTER

SIX

CHANGELOG

6.1 v1.0.1 – 27.02.2022

Added:

- Add specializations for `std::numeric_limits` for the `aarith::floating_point` numbers
- Add a constructor for `aarith::word_array` that takes a bit string as a parameter
- Add a constructor for `aarith::floating_point` that takes a bit string as a parameter
- Switched to Apache 2 license

Changed:

- Update to Catch2 v2.13.18

Removed:

Fixed:

6.2 v1.0.0 – 15.03.2021

First public release of Aarith containing

- Signed and unsigned arbitrary precision Two's Complement integers
- Arbitrary precision (for the exponent and mantissa) IEEE 754 floating-point like floating-point numbers

CHAPTER

SEVEN

TWO'S COMPLEMENT INTEGERS

Header `aarith/integer/integers.hpp`

The template classes `integer` and `uinteger` represent signed and unsigned integers of arbitrary, but compile-time static, precision stored in two's complement format

The aarith integers exhibit the usual overflow/underflow behavior (i.e. modulo 2^n) which is `not` undefined behavior!

template<size_t **Width**, class **WordType** = uint64_t>

class `aarith`::**uinteger** : public *aarith*::*word_array*<*Width*, *WordType*>

Public Functions

inline bool constexpr **is_negative**() const
Returns whether the number is negative.

Returns Always returns false

inline explicit constexpr **operator uint8_t**() const
Converts to an uint8_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint8_t storing the value of this uinteger

inline explicit constexpr **operator uint16_t**() const
Converts to an uint16_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint16_t storing the value of this uinteger

inline explicit constexpr **operator uint32_t**() const
Converts to an uint32_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint32_t storing the value of this uinteger

inline explicit constexpr **operator uint64_t**() const
Converts to an uint64_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint64_t storing the value of this uinteger

template<size_t **Width**, class **WordType** = uint64_t>

class aarith::integer : public *aarith*::*word_array*<*Width*, *WordType*>

Public Functions

inline explicit constexpr **operator int8_t**() const
Converts to an int8_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An int8_t storing the value of this integer

inline explicit constexpr **operator int16_t**() const
Converts to an int16_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An int16_t storing the value of this integer

inline explicit constexpr **operator int32_t**() const
Converts to an int32_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An int32_t storing the value of this integer

inline explicit constexpr **operator int64_t**() const
Converts to an int64_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An int64_t storing the value of this integer

inline explicit constexpr **operator uint8_t**() const
Converts to an uint8_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint8_t storing the value of this integer

inline explicit constexpr **operator uint16_t**() const
Converts to an uint16_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint16_t storing the value of this integer

inline explicit constexpr **operator uint32_t**() const
Converts to an uint32_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint32_t storing the value of this integer

inline explicit constexpr **operator uint64_t**() const
Converts to an uint64_t.

Note that there will be a possible loss of precision as this method simply cuts of the "overflowing" bits.

Returns An uint64_t storing the value of this integer

inline constexpr bool **is_negative**() const
Returns whether the number is negative.

Returns Whether the number is negative

Public Static Functions

static inline constexpr *integer* **minus_one**()
Returns minus one.

Returns minus one

CHAPTER

EIGHT

ARITHMETIC & LOGIC OPERATIONS

Header `aarith/integer/integer_operations.hpp`

namespace **`aarith`**

Functions

template<typename **`I`**, typename **`T`**>
constexpr auto **`expanding_add`**(const *I* &a, const *T* &b, const bool initial_carry)

Adds two unsigned integers of, possibly, different bit widths.

Template Parameters

- **`I`** – Integer type of the first summand
- **`T`** – Integer type of the second summand

Parameters

- **`a`** – First summand
- **`b`** – Second summand
- **`initial_carry`** – True if there is an initial carry coming in

Returns Sum of a and b with bit width max(I::width,T::width)+1

template<typename **`I`**, typename **`T`**>
constexpr auto **`expanding_add`**(const *I* &a, const *T* &b)

template<typename **`I`**>
constexpr auto **`sub`**(const *I* &a, const *I* &b) -> *I*

Computes the difference of two integers.

Template Parameters **`I`** – The integer type used in the subtraction

Parameters

- **`a`** – Minuend
- **`b`** – Subtrahend

Returns Difference between a and b

template<typename **`I`**, typename **`T`**>

constexpr auto **expanding_sub**(const *I* &a, const *T* &b)

Subtracts two integers of, possibly, different bit widths.

Expanding does not, in contrast to

See also:

expanding_add, ensure that no underflow will happen. It simply makes sure that the resulting bit width is the larger of both input bit widths.

Template Parameters

- **W** – Width of the minuend
- **V** – Width of the subtrahend

Parameters

- **a** – Minuend
- **b** – Subtrahend

Returns Difference of correct bit width

template<typename **I**>
I constexpr **add**(const *I* &a, const *I* &b)

Adds two integers.

Template Parameters **I** – The integer type used for the addition

Parameters

- **a** – First summand
- **b** – Second summand

Returns Sum of a and b

template<std::size_t **W**, std::size_t **V**, typename **WordType**>
constexpr *uinteger*<*W* + *V*, *WordType*> **schoolbook_expanding_mul**(const *uinteger*<*W*, *WordType*> &a, const *uinteger*<*V*, *WordType*> &b)

Multiplies two unsigned integers expanding the bit width so that the result fits.

This implements the simplest multiplication algorithm (binary "long multiplication") that adds up the partial products everywhere where the first multiplicand has a 1 bit. The simplicity, of course, comes at the cost of performance.

Template Parameters

- **W** – The bit width of the first multiplicand
- **V** – The bit width of the second multiplicand

Parameters

- **a** – First multiplicand
- **b** – Second multiplicand

Returns Product of a and b

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*> && *is_unsigned_v*<*I*>>>
constexpr *I* **schoolbook_mul**(const *I* &a, const *I* &b)

Multiplies two integers.

The result is then cropped to fit the initial bit width

See also:

booth_expanding_mul for that.

Note: No Type conversion is performed. If the bit widths do not match, the code will not compile! Use

---

Template Parameters `I` – The integer type to operate on

Parameters

- `a` – First multiplicand
- `b` – Second multiplicand

Returns Product of a and b

template<std::size_t `W`, std::size_t `V`, typename `WordType`>
constexpr *uinteger*<*W* + *V*, *WordType*> **`expanding_karazuba`**(const *uinteger*<*W*, *WordType*> &a, const *uinteger*<*V*, *WordType*> &b)

Multiplies two unsigned integers using the Karazuba algorithm expanding the bit width so that the result fits.

This implements the karazuba multiplication algorithm (divide and conquer).

Template Parameters

- `W` – The bit width of the first multiplicant
- `V` – The bit width of the second multiplicant

Parameters

- `a` – First multiplicant
- `b` – Second multiplicant

Returns Product of a and b

template<std::size_t `W`, std::size_t `V`, typename `WordType`>
constexpr std::pair<*uinteger*<*W*, *WordType*>, *uinteger*<*W*, *WordType*>> **`restoring_division`**(const *uinteger*<*W*, *WordType*> &numerator, const *uinteger*<*V*, *WordType*> &denominator)

Implements the restoring division algorithm.

See also:

https://en.wikipedia.org/wiki/Division_algorithm#Restoring_division

Parameters

- `numerator` – The number that is to be divided

- **denominator** – The number that divides the other number

Template Parameters **W** – Width of the numbers used in division.

Returns Pair of (quotient, remainder)

template<typename **I**>
constexpr auto **remainder**(const *I* &numerator, const *I* &denominator) -> *I*

Computes the remainder of the division of one integer by another integer.

Note: For signed integers, weird under-/overflows for ::min() may occur

---

Template Parameters **I** – Integer type to work on

Parameters

- **numerator** – The number that is to be divided
- **denominator** – The number that divides the other number

Returns The remainder of the division operation

template<typename **I**>
constexpr auto **div**(const *I* &numerator, const *I* &denominator) -> *I*

Divides one integer by another integer.

Note: integer<W>::min/integer<W>(-1) will return <integer<W>::min,0>, i.e. some weird overflow happens for signed integers

---

Template Parameters **I** – Integer type to work on

Parameters

- **numerator** – The number that is to be divided
- **denominator** – The number that divides the other number

Returns The quotient of the division operation

template<size_t **Width**, typename **WordType**>
constexpr auto **abs**(const *integer*<*Width*, *WordType*> &n) -> *integer*<*Width*, *WordType*>

Computes the absolute value of a given signed integer.

There is a potential loss of precision as abs(integer::min) > integer::max

Template Parameters **Width** – The width of the signed integer

Parameters **n** – The signed inter to be "absolute valued"

Returns The absolute value of the signed integer

template<size_t **Width**, typename **WordType**>
constexpr auto **expanding_abs**(const *integer*<*Width*, *WordType*> &n) -> *uinteger*<*Width*, *WordType*>

Computes the absolute value of a given signed integer.

This method returns an unsigned integer. This means that the absolute value will fit and no overflow will happen.

Template Parameters **Width** – The width of the signed integer

Parameters **n** – The signed inter to be "absolute valued"

Returns The absolute value of the signed integer

template<class **IntA**, class **IntB**>
constexpr auto **fun_add_expand**(const *IntA* &a, const *IntB* &b, const bool initial_carry = false)

Adds two signed integers of, possibly, different bit widths.

This is an implementation using a more functional style of programming. It is not particularly fast and only here for educational purposes. You can use method as a means to understand how to work on an integer.

Template Parameters

- **IntA** – The integer type of the first summand
- **IntB** – The integer type of the second summand

Parameters

- **a** – First summand
- **b** – Second summand
- **initial_carry** – True if there is an initial carry coming in

Returns Sum of correct maximal bit width

template<typename **I**>
constexpr auto **fun_add**(const *I* &a, const *I* &b, const bool initial_carry = false) -> *I*

Adds two integers of, possibly, different bit widths.

See also:

fun_add_expand

Template Parameters **I** – Integer type used in the addition

Parameters

- **a** – First summand
- **b** – Second summand
- **initial_carry** – True if there is an initial carry coming in

Returns Sum of a and b

template<size_t **Width**, typename **WordType**>
auto constexpr **operator>>=**(*integer*<*Width*, *WordType*> &lhs, const size_t rhs) -> *integer*<*Width*, *WordType*>

Arithmetic right-shift operator.

This shift preserves the signedness of the integer.

Template Parameters **Width** – The width of the signed integer

Parameters

- **lhs** – The integer to be shifted
- **rhs** – The number of bits to be shifted

Returns The shifted integer

template<size_t **Width**, typename **WordType**, typename **U**, typename = std::enable_if_t<*is_unsigned_v*<*U*>>>

auto constexpr **operator>>=**(*integer*<*Width*, *WordType*> &lhs, const *U* &rhs) -> *integer*<*Width*, *WordType*>&

template<size_t **Width**, typename **WordType**>
auto constexpr **operator>>**(const *integer*<*Width*, *WordType*> &lhs, const size_t rhs) -> *integer*<*Width*, *WordType*>

Arithmetic right-shift operator.

This shift preserves the signedness of the integer.

Template Parameters **Width** – The width of the signed integer

Parameters

- **lhs** – The integer to be shifted
- **rhs** – The number of bits to be shifted

Returns The shifted integer

template<size_t **Width**, typename **WordType**, typename **U**, typename = std::enable_if_t<*is_unsigned_v*<*U*>>>
auto constexpr **operator>>**(const *integer*<*Width*, *WordType*> &lhs, const *U* &rhs) -> *integer*<*Width*, *WordType*>

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
I &**operator--**(*I* &a)

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
I **operator--**(*I* &a, int)

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
I &**operator++**(*I* &a)

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
I **operator++**(*I* &a, int)

template<size_t **W**, size_t **V**, typename **WordType**>
constexpr auto **naive_expanding_mul**(const *integer*<*W*, *WordType*> &m, const *integer*<*V*, *WordType*> &r)

Naively multiplies two signed integers.

Template Parameters

- **W** – The bit width of the first multiplicand
- **V** – The bit width of the second multiplicand

Parameters

- **a** – First multiplicand
- **b** – Second multiplicand

Returns Product of a and b

template<size_t **W**, typename **WordType**>
constexpr *integer*<*W*, *WordType*> **naive_mul**(const *integer*<*W*, *WordType*> &a, const *integer*<*W*, *WordType*> &b)

Naively multiplies two integers.

The result is then cropped to fit the initial bit width

See also:

booth_expanding_mul for that.

Note: No Type conversion is performed. If the bit widths do not match, the code will not compile! Use

---

Template Parameters **I** – The integer type to operate on

Parameters

- **a** – First multiplicand
- **b** – Second multiplicand

Returns Product of a and b

template<size_t **x**, size_t **y**, typename **WordType**>
constexpr auto **booth_expanding_mul**(const *integer*<*x*, *WordType*> &m, const *integer*<*y*, *WordType*> &r) -> *integer*<*y* + *x*, *WordType*>

Multiplies two signed integers.

This implements the Booth multiplication algorithm with extension to correctly handle the most negative number. See https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm for details.

Template Parameters

- **x** – The bit width of the first multiplicant
- **y** – The bit width of the second multiplicant

Parameters

- **m** – Multiplicand
- **r** – Multiplier

Returns Product of m and r

template<size_t **W**, typename **WordType**>
constexpr *integer*<*W*, *WordType*> **booth_mul**(const *integer*<*W*, *WordType*> &a, const *integer*<*W*, *WordType*> &b)

Multiplies two integers.

The result is then cropped to fit the initial bit width

See also:

booth_expanding_mul for that.

Note: No Type conversion is performed. If the bit widths do not match, the code will not compile! Use

---

Template Parameters **I** – The integer type to operate on

Parameters

- **a** – First multiplicand

- **b** – Second multiplicand

Returns Product of a and b

template<size_t **x**, size_t **y**, typename **WordType**>
constexpr auto **booth_inplace_expanding_mul**(const *integer*<*x*, *WordType*> &m, const *integer*<*y*, *WordType*> &r) -> *integer*<*y* + *x*, *WordType*>

Multiplies two signed integers.

This implements the Booth multiplication algorithm with extension to correctly handle the most negative number. See https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm for details.

Template Parameters

- **x** – The bit width of the first multiplicant
- **y** – The bit width of the second multiplicant

Parameters

- **m** – Multiplicand
- **r** – Multiplier

Returns Product of m and r

template<size_t **W**, typename **WordType**>
constexpr *integer*<*W*, *WordType*> **booth_inplace_mul**(const *integer*<*W*, *WordType*> &a, const *integer*<*W*, *WordType*> &b)

Multiplies two integers.

The result is then cropped to fit the initial bit width

See also:

booth_expanding_mul for that.

Note: No Type conversion is performed. If the bit widths do not match, the code will not compile! Use

---

Template Parameters **I** – The integer type to operate on

Parameters

- **a** – First multiplicand
- **b** – Second multiplicand

Returns Product of a and b

template<size_t **W**, typename **WordType**>
constexpr auto **negate**(const *integer*<*W*, *WordType*> &n) -> *integer*<*W*, *WordType*>

Negates the value.

Template Parameters **W** – The width of the signed integer

Parameters **n** – The signed integer whose sign is to be changed

Returns The negative value of the signed integer

template<size_t **W**, typename **WordType**>

constexpr int8_t **signum**(*integer*<*W*, *WordType*> n)

Computes the sign of the integer.

For the number zero, the function returns a signum of 0, -1 for negative numbers and +1 for positive numbers.

Template Parameters

- **W** – The width of the integer
- **WordType** – The word type to store the data in

Parameters **n** – The integer

Returns The sign of the integer

template<size_t **W**, typename **WordType**>
constexpr int8_t **signum**(*uinteger*<*W*, *WordType*> n)

Computes the sign of the unsigned integer.

For the number zero, the function returns a signum of 0 and a 1 for every other number.

Template Parameters **W** – The width of the unsigned integer

Parameters **n** – The integer

Returns The sign of the integer

template<std::size_t **W**, std::size_t **V**, typename **WordType**>
constexpr std::pair<*integer*<*W*, *WordType*>, *integer*<*W*, *WordType*>> **restoring_division**(const *integer*<*W*, *WordType*> &numerator, const *integer*<*V*, *WordType*> &denominator)

Implements the restoring division algorithm.

See also:

https://en.wikipedia.org/wiki/Division_algorithm#Restoring_division

---

Note: integer<W>::min/integer<W>(-1) will return <integer<W>::min,0>, i.e. some weird overflow happens

---

Parameters

- **numerator** – The number that is to be divided
- **denominator** – The number that divides the other number

Template Parameters **W** – Width of the numbers used in division.

Returns Pair of (quotient, remainder)

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
constexpr *I* **mul**(const *I* &a, const *I* &b)

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
auto constexpr **expanding_mul**(const *I* &a, const *I* &b)

template<typename **IntegerType**>
IntegerType **pow**(const *IntegerType* &base, const size_t exponent)

Exponentiation function.

Note: This function does not make any attempts to be fast or to prevent overflows!

Note: If exponent equals std::numeric_limits<size_t>::max(), this method throws an exception, unless base equals zero

Template Parameters **W** – Bit width of the integer

Parameters

- **base** –
- **exponent** –

Returns The base to the power of the exponent

template<typename **IntegerType**>
IntegerType **pow**(const *IntegerType* &base, const *IntegerType* &exponent)

Exponentiation function.

Note: This function does not make any attempts to be fast or to prevent overflows!

Note: If exponent equals std::numeric_limits<IntegerType>::max(), this method throws an exception, unless base equals zero

Template Parameters **IntegerType** – The type of integer used in the computation

Parameters

- **base** –
- **exponent** –

Returns The base to the power of the exponent

template<std::size_t **W**, typename **WordType**>
constexpr *uinteger*<*W*, *WordType*> **karazuba**(const *uinteger*<*W*, *WordType*> &a, const *uinteger*<*W*, *WordType*> &b)

Multiplies two unsigned integers using the Karazuba algorithm.

This implements the karazuba multiplication algorithm (divide and conquer).

Template Parameters **W** – The bit width of the multiplicants

Parameters

- **a** – First multiplicant

- **b** – Second multiplicant

Returns Product of a and b

template<typename **Integer**>
constexpr *Integer* **distance**(const *Integer* &a, const *Integer* &b)

Computes the distance (i.e. the absolute difference) between two integers.

Template Parameters **Integer** – The integer type to operate on

Parameters

- **a** – First integer
- **b** – Second integer

Returns The distance between the two integers

template<typename **W**, typename **I**, typename = std::enable_if_t<*is_word_array_v*<*W*> && *is_integral_v*<*I*> && *is_unsigned_v*<*I*>>>
constexpr auto **operator>>=**(*W* &lhs, const *I* rhs) -> *W*

Left-shift assignment operator.

Template Parameters **W** – The word_container type to work on

Parameters

- **lhs** – The word_container to be shifted
- **rhs** – The number of bits to shift

Returns The shifted word_container

template<typename **W**, typename **I**, typename = std::enable_if_t<*is_word_array_v*<*W*> && *is_integral_v*<*I*> && *is_unsigned_v*<*I*>>>
constexpr auto **operator>>**(const *W* &lhs, const *I* rhs) -> *W*

Left-shift assignment operator.

Template Parameters **W** – The word_container type to work on

Parameters

- **lhs** – The word_container to be shifted
- **rhs** – The number of bits to shift

Returns The shifted word_container

namespace **integer_operators**

Convenience namespace to include when code should be written the "normal" way. There is one caveat though: No automatic type conversion will take place!

Functions

template<typename **I**, typename = std::enable_if_t<*is_integral*<*I*>::value>>
auto constexpr **operator-**(const *I* &num) -> *I*

template<typename **I**, typename = std::enable_if_t<*is_integral*<*I*>::value>>
auto constexpr **operator+**(const *I* &lhs, const *I* &rhs) -> *I*

template<typename **I**, typename = std::enable_if_t<*is_integral*<*I*>::value>>

auto constexpr **operator-**(const *I* &lhs, const *I* &rhs) -> *I*

template<typename **I**, typename = std::enable_if_t<*is_integral_v*<*I*>>>
auto constexpr **operator***(const *I* &lhs, const *I* &rhs) -> *I*

template<typename **I**, typename = std::enable_if_t<*is_integral*<*I*>::value>>
auto constexpr **operator/**(const *I* &lhs, const *I* &rhs) -> *I*

template<typename **I**, typename = std::enable_if_t<*is_integral*<*I*>::value>>
auto constexpr **operator%**(const *I* &lhs, const *I* &rhs) -> *I*

CHAPTER

NINE

COMPARISON OPERATIONS

Header `aarith/integer/integer_comparisons.hpp`

namespace **aarith**

Functions

template<size_t **W**, size_t **V**, typename **T**, template<size_t, typename> typename **Int**>
constexpr bool **operator==**(const *Int*<*W*, *T*> &a, const *Int*<*V*, *T*> &b)

> **Note:** Two numbers can be equal even though they have different bit widths!

template<size_t **W**, size_t **V**, typename **WordType**>
constexpr bool **operator<**(const *uinteger*<*W*, *WordType*> &a, const *uinteger*<*V*, *WordType*> &b)

template<typename **W**, typename **V**>
constexpr bool **operator<=**(const *W* &a, const *V* &b)

template<typename **W**, typename **V**>
constexpr bool **operator>=**(const *W* &a, const *V* &b)

template<typename **W**, typename **V**>
constexpr bool **operator>**(const *W* &a, const *V* &b)

template<size_t **W**, size_t **V**, typename **WordType**>
constexpr bool **operator<**(const *integer*<*W*, *WordType*> &a, const *integer*<*V*, *WordType*> &b)

template<typename **Integer**>
const *Integer* &**min**(const *Integer* &a, const *Integer* &b)

template<typename **Integer**>
const *Integer* &**max**(const *Integer* &a, const *Integer* &b)

CHAPTER

TEN

FLOATING-POINT NUMBERS

Header `aarith/float/floating_point.hpp`

The template class `floating_point` represents a floating-point number of arbitrary, but compile-time static precision.

template<size_t **E**, size_t **M**, typename **WordType**>

class *aarith*::**floating_point**

Public Functions

inline constexpr bool **is_positive**() const
Tests whether the floating-point number is positive.

This returns true for zeros and NaNs as well.

Returns True iff the sign bit is not set

inline constexpr bool **is_negative**() const
Tests whether the floating-point number is negative.

This returns true for zeros and NaNs as well.

Returns True iff the sign bit is set

inline constexpr bool **is_finite**() const
Returns whether the number is finite.

Note: NaNs are *not* considered finite

Returns True iff the number is finite

inline constexpr bool **is_nan**() const
Checks whether the floating point number is NaN (not a number)

Note: There is no distinction between signalling and non-signalling NaN

Returns True iff the number is NaN

inline constexpr bool **is_qNaN**() const
Checks if the number is a quiet NaN.

Returns True iff the number is a quiet NaN

inline constexpr bool **is_sNaN**() const
Checks if the number is a signalling NaN.

Returns True iff the number is a signalling NaN

inline constexpr bool **is_zero**() const
Checks whether the floating point number is zero.

Returns true for both the positive and negative zero

Returns True iff the floating point is zero

inline constexpr bool **is_pos_zero**() const
Checks whether the floating point number is positive zero.

Returns True iff the floating point is positive zero

inline constexpr bool **is_neg_zero**() const
Checks whether the floating point number is negative zero.

Returns True iff the floating point is negative zero

inline constexpr bool **is_normalized**() const
Checks whether the number is normal.

This is true if and only if the floating-point number is normal (not zero, subnormal, infinite, or NaN).

Returns True iff the number is normalized

inline constexpr bool **is_denormalized**() const
Returns whether the number is denormalized.

Note: Denormalized numbers do *not* include: NaN, +/- inf and, surprisingly, zero.

Returns True iff the number is denormalized

inline constexpr bool **is_subnormal**() const
Tests if the number is subnormal.

Note: Zero is *not* considered subnormal!

Returns True iff the number is subnormal

inline constexpr bool **is_special**() const
Returns whether the number is denormalized or NaN/Inf.

Returns True iff the number is denornmalized, infinite or a NaN

inline explicit constexpr **operator float**() const
Casts the normalized float to the native float type.

Note: The cast is only possible when there will be no loss of precision

---

Returns The value converted to float format

inline explicit constexpr **operator double**() const
Casts the normalized float to the native double type.

Note: The cast is only possible when there will be no loss of precision

Returns The value converted to double format

Public Static Functions

static inline constexpr *floating_point* **zero**()

Returns The value zero

static inline constexpr *floating_point* **neg_zero**()

Returns The value negative zero

static inline constexpr *floating_point* **one**()

Returns The value one

static inline constexpr *floating_point* **neg_one**()

Returns The value one

static inline constexpr *floating_point* **pos_infinity**()

Returns positive infinity

static inline constexpr *floating_point* **neg_infinity**()

Returns negative infinity

static inline constexpr *floating_point* **min**()

Returns The smallest finite value

static inline constexpr *floating_point* **max**()

Returns The largest finite value

static inline constexpr *floating_point* **smallest_normalized**()

Returns Smallest positive normalized value

static inline constexpr *floating_point* **smallest_denormalized**()

Returns Smallest positive denormalized value

static inline constexpr *floating_point* **round_error**()

Returns The maximal rounding error (assuming round-to-nearest)

static inline constexpr *floating_point* **qNaN**(const IntegerFrac &payload = IntegerFrac::msb_one())
Creates a quiet NaN value.

Parameters **payload** – The payload to store in the NaN

Returns The bit representation of the quiet NaN containing the payload

static inline constexpr *floating_point* **sNaN**(const IntegerFrac &payload = IntegerFrac::one())
Creates a signalling NaN value.

Parameters **payload** – The payload to store in the NaN (must not be zero)

Returns The bit representation of the signalling NaN containing the payload

static inline constexpr *floating_point* **NaN**()
Returns a floating point number indicating not a number (NaN).

Returns A non-signalling not a number value

CHAPTER

ELEVEN

ARITHMETIC & LOGIC OPERATIONS

Header `aarith/float/float_operations.hpp`

namespace **aarith**

Functions

template<size_t **E**, size_t **M**, class **Function_add**, class **Function_sub**>
auto **add_**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, *Function_add* fun_add,
Function_sub fun_sub) -> *floating_point*<*E*, *M*>

Generic addition of two *floating_point* values.

This method computes the sum of two floating-point values using the provided functions `fun_add` and `fun_sub` to compute the new mantissa. This generic function allows to easily implement own adders, e.g. to develop new hardware implementations.

Note: As an end-user of aarith, you will, most likely, never need to call this function.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width
- **Function_add** – Function object type for performing an addition
- **Function_sub** – Function object fype for performing a subtraction

Parameters

- **lhs** – Left-hand side argument of the usm
- **rhs** – Right-hand side argument of the sum
- **fun_add** – Function performing the addition of the mantissae
- **fun_sub** – Function performing the subtraction of the mantissae

Returns The sum of lhs + rhs using the provided functions

template<size_t **E**, size_t **M**, class **Function_add**, class **Function_sub**>

auto **sub_**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, *Function_add* fun_add, *Function_sub* fun_sub) -> *floating_point*<*E*, *M*>

Generic subtraction of two `floating_point` values.

This method computes the difference of two floating-point values using the provided functions `fun_add` and `fun_sub` to compute the new mantissa. This generic function allows to easily implement own adders, e.g. to develop new hardware implementations.*

Note: As an end-user of aarith, you will, most likely, never need to call this function.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width
- **Function_add** – Function object type for performing an addition
- **Function_sub** – Function object fype for performing a subtraction

Parameters

- **lhs** – Left-hand side argument of the usm
- **rhs** – Right-hand side argument of the sum
- **fun_add** – Function performing the addition of the mantissae
- **fun_sub** – Function performing the subtraction of the mantissae

Returns The sum of lhs + rhs using the provided functions

template<size_t **E**, size_t **M**>
auto **add**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> *floating_point*<*E*, *M*>

Adds two `floating_point` values.

Parameters

- **lhs** – The first number that is to be summed up
- **rhs** – The second number that is to be summed up

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The sum

template<size_t **E**, size_t **M**>
auto **sub**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> *floating_point*<*E*, *M*>

Subtract two `floating_point` values.

Parameters

- **lhs** – The minuend
- **rhs** – The subtrahend

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The difference lhs-rhs

template<size_t **E**, size_t **M**, typename **WordType**>
auto **mul**(const *floating_point*<*E*, *M*, *WordType*> lhs, const *floating_point*<*E*, *M*, *WordType*> rhs) -> *floating_point*<*E*, *M*, *WordType*>

Multiplies two `floating_point` numbers.

Parameters

- **lhs** – The multiplicand
- **rhs** – The multiplicator

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The product lhs*rhs

template<size_t **E**, size_t **M**, typename **WordType**>
auto **div**(const *floating_point*<*E*, *M*, *WordType*> lhs, const *floating_point*<*E*, *M*, *WordType*> rhs) -> *floating_point*<*E*, *M*, *WordType*>

Division with floating_points: lhs/rhs.

Parameters

- **lhs** – The dividend
- **rhs** – The divisor

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1
- **WordType** – The word type used to internally store the data

Returns The quotient lhs/rhs

template<size_t **E**, size_t **M**, typename **WordType** = uint64_t>
constexpr *floating_point*<*E*, *M*, *WordType*> **negate**(const *floating_point*<*E*, *M*, *WordType*> &x)

Computes the negative value of the floating-point number.

Quoting the standard: copies a floating-point operand x to a destination in the same format, reversing the sign bit. negate(x) is not the same as subtraction(0, x)

Note: This method ignores NaN values in the sense that they are also copied and the sign bit flipped.

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa
- **WordType** – The word type used to internally store the data

Returns The negated value of the provided number

template<size_t **E**, size_t **M**, typename **WordType** = uint64_t>

constexpr *floating_point*<*E*, *M*, *WordType*> **copy**(const *floating_point*<*E*, *M*, *WordType*> &x)
Copies the floating-point number.

Quoting the standard: copies a floating-point operand x to a destination in the same format, with no change to the sign bit.

Note: This method ignores NaN values in the sense that they are also copied not signalling any error.

Note: This is a rather useless method that only exists to be more compliant with the IEEE 754 (2019) standard.

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa
- **WordType** – The word type used to internally store the data

Returns The copied value

template<size_t **E**, size_t **M**, typename **WordType** = uint64_t>
constexpr *floating_point*<*E*, *M*, *WordType*> **copySign**(const *floating_point*<*E*, *M*, *WordType*> &x, const *floating_point*<*E*, *M*, *WordType*> &y)
Copies a floating-point number using the sign of another number.

Quoting the standard: copies a floating-point operand x to a destination in the same format as x, but with the sign bit of y.

Note: This method ignores NaN values in the sense that they are also copied not signalling any error.

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa
- **WordType** – The word type used to internally store the data

Returns The copied value

template<size_t **Start**, size_t **End**, size_t **E**, size_t **M**, typename **WordType**>
constexpr *word_array*<(*Start* - *End*) + 1, *WordType*> **bit_range**(const *floating_point*<*E*, *M*, *WordType*> &f)
Extracts a bitstring range from the bit representation of the float.

Note that the indexing is done

- zero based starting from the LSB
- is inclusive (i.e. the start and end point are part of the range)

Template Parameters

- **Start** – Starting index (inclusive, from left to right)
- **Eend** – Ending index (inclusive, from left to right)

- **E** – Width of the exponent
- **M** – Width of the mantissa

Parameters **f** – Float from which the range is taken from

Returns Range float[End,Start], inclusive

namespace **float_operators**

This additional nesting of a namespace allows to include aarith without having the usual operator names imported as well.

The use case for this is to allow explicitly replace the conventional arithmetic operations with sepcialized ones. This can, e.g., be used when evaluating approximate operations in the context of neural networks. The name lookup of C++ makes it necessary not to see the operators earlier.

Functions

template<size_t **E**, size_t **M**, typename **WordType**>
auto **operator+**(const *floating_point*<*E*, *M*, *WordType*> &lhs, const *floating_point*<*E*, *M*, *WordType*> &rhs) -> *floating_point*<*E*, *M*, *WordType*>

template<size_t **E**, size_t **M**, typename **WordType**>
auto **operator-**(const *floating_point*<*E*, *M*, *WordType*> &lhs, const *floating_point*<*E*, *M*, *WordType*> &rhs) -> *floating_point*<*E*, *M*, *WordType*>

template<size_t **E**, size_t **M**, typename **WordType**>
auto **operator***(const *floating_point*<*E*, *M*, *WordType*> &lhs, const *floating_point*<*E*, *M*, *WordType*> &rhs) -> *floating_point*<*E*, *M*, *WordType*>

template<size_t **E**, size_t **M**, typename **WordType**>
auto **operator/**(const *floating_point*<*E*, *M*, *WordType*> &lhs, const *floating_point*<*E*, *M*, *WordType*> &rhs) -> *floating_point*<*E*, *M*, *WordType*>

template<size_t **E**, size_t **M**, typename **WordType**>
auto **operator-**(const *floating_point*<*E*, *M*, *WordType*> &x) -> *floating_point*<*E*, *M*, *WordType*>

CHAPTER

TWELVE

COMPARISON OPERATIONS

Header `aarith/float/float_comparisons.hpp`

namespace **aarith**

Functions

template<size_t **E**, size_t **M**>
auto constexpr **operator<**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> bool

template<size_t **E**, size_t **M**>
bool constexpr **bitwise_equality**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs)

Compares to floating point numbers bit by bit.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width

Parameters

- **lhs** –
- **rhs** –

Returns True iff the floats match in every single bit

template<size_t **E**, size_t **M**>
auto constexpr **operator==**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> bool

template<size_t **E**, size_t **M**, size_t **E_**, size_t **M_**, typename = std::enable_if_t<(*E* != *E_*) || (*M* != *M_*)>>
bool constexpr **logical_equality**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E_*, *M_*> rhs)

template<size_t **E**, size_t **M**, size_t **E_**, size_t **M_**, typename = std::enable_if_t<(*E* != *E_*) || (*M* != *M_*)>>
auto constexpr **operator==**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E_*, *M_*> rhs) -> bool

template<size_t **E**, size_t **M**, size_t **E_**, size_t **M_**, typename = std::enable_if_t<(*E* != *E_*) || (*M* != *M_*)>>
bool constexpr **operator<**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E_*, *M_*> rhs)

template<size_t **E**, size_t **M**, size_t **E_**, size_t **M_**>

auto constexpr **operator!=**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E_*, *M_*> rhs) -> bool

template<size_t **E**, size_t **M**>
auto constexpr **operator>**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> bool

template<size_t **e**, size_t **m**>
auto constexpr **operator>=**(const *floating_point*<*e*, *m*> lhs, const *floating_point*<*e*, *m*> rhs) -> bool

template<size_t **e**, size_t **m**>
auto constexpr **operator<=**(const *floating_point*<*e*, *m*> lhs, const *floating_point*<*e*, *m*> rhs) -> bool

CHAPTER

THIRTEEN

UTILITIES AND HELPERS

namespace **aarith**

Enums

enum **IEEEClass**

Enumeration of the different types of floating-points.

These are defined in Section 5.7.2. of the 2019 standard

Values:

enumerator **signalingNaN**

enumerator **quietNaN**

enumerator **negativeInfinity**

enumerator **negativeNormal**

enumerator **negativeSubnormal**

enumerator **negativeZero**

enumerator **positiveZero**

enumerator **positiveSubnormal**

enumerator **positiveNormal**

enumerator **positiveInfinity**

enumerator **UNCLASSIFIED**

enum **Radix**
The possible radices used to store floating-point numbers.

Aarith uses base two only.

Values:

enumerator **Two**

enumerator **Ten**

Functions

template<typename **F**>
bool constexpr **isSignMinus**(const *F* &f)
Tests whether a floating-point number is negative.

isSignMinus(x) is true if and only if x has negative sign. isSignMinus applies to zeros and NaNs as well.

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is negative

template<typename **F**>
bool constexpr **isNormal**(const *F* &f)
Tests whether a floating-point number is normal.

isNormal(f) is true if and only if f is normal (not zero, subnormal, infinite, or NaN)

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is normal

template<typename **F**>
bool constexpr **isFinite**(const *F* &f)
Tests whether a floating-point number is finite.

isFinite(f) is true if and only if f is zero, subnormal or normal (not infinite or NaN).

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is finite

template<typename **F**>
bool constexpr **isZero**(const *F* &f)
Tests whether a floating-point number is zero (ignoring the sign).

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is +/- zero

template<typename **F**>
bool constexpr **isSubnormal**(const *F* &f)
Tests whether a floating-point number is zero (ignoring the sign).

Note: Zero is *not* considered subnormal!

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is +/- zero

template<typename **F**>
bool constexpr **isInfinite**(const *F* &f)
Tests whether a floating-point number is infinite (ignoring the sign)

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is infinite

template<typename **F**>
bool constexpr **isNaN**(const *F* &f)
Tests whether a floating-point number is NaN.

Note: This method does not distinguish between signalling and quiet NaNs

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is NaN

template<typename **F**>
bool constexpr **isSignaling**(const *F* &f)
Tests whether a floating-point number is a signaling NaN.

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is a signaling NaN

template<typename **F**>
bool constexpr **isQuiet**(const *F* &f)
Tests whether a floating-point number is a quiet NaN.

Note: This method is not required by the IEEE 754 standard

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True iff the number is a quiet NaN

template<typename **F**>
constexpr *IEEEClass* **fp_class**(const *F* &f)

Determines the class (e.g. NaN, positive subnormal) of a floating-point number.

This method corresponds to the "class" method described in Section 5.7.2 of the 2019 standard. The function name had to be changed as C++ does not allow to name functions "class".

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns The class of the floating-point number

template<typename **F**>
Radix constexpr **radix**([[maybe_unused]] const *F* &f)

Returns the radix of the floating-point number.

Aarith only supports base two, hence Radix::Two is the only return value.

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns

template<typename **F**>
bool constexpr **isCanonical**([[maybe_unused]] const *F* &f)

Tests whether a floating-point number is canonical.

Aarith does not support non-canonical numbers hence this method always returns true.

Template Parameters **F** – The floating-point type

Parameters **f** – The floating-point number to test

Returns True

template<typename **F**, typename = std::enable_if_t<std::is_floating_point<*F*>::value>>
constexpr size_t **get_mantissa_width**()

template<>
constexpr size_t **get_mantissa_width**<float>()

template<>
constexpr size_t **get_mantissa_width**<double>()

template<typename **F**, typename = std::enable_if_t<std::is_floating_point<*F*>::value>>
constexpr size_t **get_exponent_width**()

template<>
constexpr size_t **get_exponent_width**<float>()

template<>
constexpr size_t **get_exponent_width**<double>()

template<typename **F**, typename = std::enable_if_t<std::is_floating_point<*F*>::value>>
inline auto **disassemble_float**(*F* num) -> *float_disassembly*

template<typename **F**, typename **WordType**, typename = std::enable_if_t<std::is_floating_point<*F*>::value>>
inline constexpr auto **extract_exponent**(*F* num)

template<typename **F**, typename **Wordtype**, typename = std::enable_if_t<std::is_floating_point<*F*>::value>>
inline constexpr auto **extract_mantissa**(*F* num)

struct **float_disassembly**

Public Members

uint64_t **exponent**

uint64_t **mantissa**

bool **is_neg**

namespace **float_extraction_helper**
Namespace to prevent people from accidentally using this trait.

template<typename **F**>

struct **bit_cast_to_type_trait**

template<>

struct **bit_cast_to_type_trait**<double>

Public Types

using **type** = uint64_t

Public Static Attributes

static constexpr size_t **width** = 64

template<>

struct **bit_cast_to_type_trait**<float>

Public Types

using **type** = uint32_t

Public Static Attributes

static constexpr size_t **width** = 32

13.1 NaN Payloads

Header `aarith/float/nan_payload.hpp`

Giving access to the payloads encoded in NaN values is mandatory as per [IEEE754].

namespace **aarith**

Functions

template<size_t **E**, size_t **M**, typename **WordType**>
floating_point<*E*, *M*, *WordType*> constexpr **getPayload**(const *floating_point*<*E*, *M*, *WordType*> &x)

Extracts the payload from an NaN.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width
- **WordType** – The data type the underlying data is stored in

Parameters **x** – Floating-point to extract the payload from

Returns The payload or -1, when x is not NaN

template<size_t **E**, size_t **M**, typename **WordType**>
floating_point<*E*, *M*, *WordType*> constexpr **setPayload**(const *floating_point*<*E*, *M*, *WordType*> &x)

Creates quiet NaN with a specified payload.

For some reason, negative parameters result in +0 to be returned. Don't ask me, ask the standard.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width
- **WordType** – The data type the underlying data is stored in

Parameters **x** – Floating-point number to take the payload to be stored from

Returns A quiet NaN with the specified payload, +0 in case of error

template<size_t **E**, size_t **M**, typename **WordType**>
floating_point<*E*, *M*, *WordType*> constexpr **setPayloadSignaling**(const *floating_point*<*E*, *M*, *WordType*> &x)

Creates signaling NaN with a specified payload.

For some reason, negative parameters result in +0 to be returned. Don't ask me, ask the standard.

Template Parameters

- **E** – Exponent width
- **M** – Mantissa width
- **WordType** – The data type the underlying data is stored in

Parameters **x** – Floating-point number to take the payload to be stored from

Returns A signaling NaN with the specified payload, +0 in case of error

CHAPTER

FOURTEEN

ANYTIME INSTRUCTIONS

See [Brand2019] and [Brand2020] for details.

Header `aarith/float/float_approx_operations.hpp`

template<size_t **E**, size_t **M**>
auto *aarith*::**anytime_add**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, const unsigned int bits = *M* + 1) -> *floating_point*<*E*, *M*>

Addition of two normfloats using anytime addition: lhs+rhs.

Parameters

- **lhs** – The first number that is to be summed up
- **rhs** – The second number that is to be summed up
- **bits** – The number of most-significant bits that are calculated of the mantissa addition

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The sum

template<size_t **E**, size_t **M**>
auto *aarith*::**anytime_sub**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, const unsigned int bits = *M* + 1) -> *floating_point*<*E*, *M*>

Subtraction with floating_points: lhs-rhs.

Parameters

- **lhs** – The minuend
- **rhs** – The subtrahend
- **bits** – The number of most-significant bits that are calculated of the mantissa subtraction

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The difference lhs-rhs

template<size_t **E**, size_t **M**>
auto *aarith*::**anytime_mul**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, const unsigned int bits = 2 * *M*) -> *floating_point*<*E*, *M*>

Multiplication with floating_points: lhs*rhs.

Parameters

- **lhs** – The multiplicand
- **rhs** – The multiplicator
- **bits** – The number of most-significant bits that are calculated of the mantissa multiplication

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The product lhs*rhs

template<size_t **E**, size_t **M**>
auto *aarith*::**anytime_div**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs, const unsigned int bits = *M* + 1) -> *floating_point*<*E*, *M*>

Anytime division with floating_points: lhs/rhs.

Parameters

- **lhs** – The dividend
- **rhs** – The divisor
- **bits** – The number of most-significant bits that are calculated of the mantissa division

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The quotient lhs/rhs

CHAPTER

FIFTEEN

THE FAU ADDER

15.1 When computing with unsigned integers

The FAU adder is an adder that speed up the computation of the sum by splitting the carry chain. In order to keep the error small, some bits of the least-significant part are used in a carry prediction unit.

See *the uses cases* and [Echavarria2016] for details.

Header `aaarith/integer/integer_approx_operations.hpp`

template<size_t **width**, size_t **lsp_width**, size_t **shared_bits** = 0>
uinteger<*width* + 1> `aarith::FAUadder`(const *uinteger*<*width*> &a, const *uinteger*<*width*> &b)

template<size_t **width**, size_t **lsp_width**, size_t **shared_bits** = 0>
uinteger<*width* + 1> `aarith::FAUsubtractor`(const *uinteger*<*width*> &a, const *uinteger*<*width*> &b)

15.2 When computing with floating_points

The FAU adder can also be used when computing with *floating_point* numbers. Here, the FAU adder ist used to approximately compute the mantissa.

Header `aaarith/float/float_approx_operations.hpp`

template<size_t **E**, size_t **M**, size_t **LSP**, size_t **SHARED**>
auto `aarith::FAU_add`(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> *floating_point*<*E*, *M*>

Addition of two floating_points using the FAU adder: lhs+rhs.

Parameters

- **lhs** – The first number that is to be summed up
- **rhs** – The second number that is to be summed up
- **bits** – The number of most-significant bits that are calculated of the mantissa addition

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The sum

template<size_t **E**, size_t **M**, size_t **LSP**, size_t **SHARED**>

auto aarith::**FAU_sub**(const *floating_point*<*E*, *M*> lhs, const *floating_point*<*E*, *M*> rhs) -> *floating_point*<*E*, *M*>
Subtraction with floating_points using the FAU adder: lhs-rhs.

Parameters

- **lhs** – The minuend
- **rhs** – The subtrahend
- **bits** – The number of most-significant bits that are calculated of the mantissa subtraction

Template Parameters

- **E** – Width of exponent
- **M** – Width of mantissa including the leading 1

Returns The difference lhs-rhs

CHAPTER

SIXTEEN

PUBLICATION

If you use Aarith (e.g., in your publication), please cite

Oliver Keszocze, Marcel Brand, Christian Heidorn, und Jürgen Teich. „Aarith: An Arbitrary Precision Number Library“, In: ACM/SIGAPP Symposium On Applied Computing. March 2021.

Bibtex:

```
@inproceedings{Keszocze2021,
  title = {Aarith: {{An Arbitrary Precision Number Library}}},
  booktitle = {ACM/SIGAPP Symposium On Applied Computing},
  author = {Keszocze, Oliver and Brand, Marcel and Heidorn, Christian and Teich, Jürgen},
  date = {2021-03},
  location = {{Virtual Event, South Korea}},
  series = {{{SAC}}'21}
}
```

16.1 word_array

The template class `word_array` serves as the base for all other types that interpret underlying words as numbers.

Hint: The class `word_array` is only used internally. The aarith end-user, most likely, will have no need to directly interact with this class. No knowledge of it is necessary to use aarith.

template<size_t **Width**, class **WordType** = uint64_t>

class *aarith*::**word_array**

Subclassed by *aarith::integer< Width, WordType >*, *aarith::uinteger< Width, WordType >*, *aarith::uinteger< BitWidth+1, WordType >*, *aarith::uinteger< E, WordType >*, *aarith::uinteger< MW, WordType >*

Public Functions

constexpr **word_array**() = default
Default constructor for the word array.

Initializes the word array to store only zeros.

inline explicit **word_array**(std::string_view bs)
Creates a *word_array* from a given bit string.

Example: word_array<5> w = word_array<5>::from_bit_string("11010");

If the supplied bit string is longer than the *word_array* to be created, the rest of the bits will be ignored. If the *word_array* has more bits than the string, these bits are initialized with zero.

Parameters **bs** – The bitstring to create the word array from

Returns A *word_array* with the same bits set as in the paramater bs

template<size_t **V**, typename **T**>
inline void **set_bits**(size_t end, const *word_array*<*V*, *T*> &other)

Template Parameters

- **V** – Bit width of the *word_array*
- **T** – Word type to store the data in

Parameters

- **end** –
- **other** – The *word_array* to take the values from

inline constexpr auto **msb**() const -> bit_type
Returns ths most significant bit.

The most significant bit is the Width's one (i.e. the one you can get via bit(Width-1)). This method is simply there for convenience.

inline void constexpr **set_msb**(const bool b)
Sets the value of the most significant bit (MSB)

Parameters **b** – The value the MSB is set to

inline auto constexpr **bit**(size_t index) const -> bit_type
Returns bit at given index.

Note: No bounds checking is performed! If your index is too large bad things will happen!

Parameters **index** – The index for which the bit is to be returned

Returns The bit at the indexed position

inline auto **at**(size_t pos) const
Returns a const reference to the element at specified location pos, with bounds checking.

If pos is not within the range of the container, an exception of type std::out_of_range is thrown.

Parameters **pos** – position of the element to return

Returns Const reference to the requested element.

inline auto **operator[]**(size_t pos) const
Returns a reference to the element at specified location pos. No bounds checking is performed.

Parameters **pos** – position of the element to return

Returns Const reference to the requested element.

inline constexpr bool **empty**() const noexcept
Checks if the container has no elements.

Returns false

inline auto **front**() const
Returns a const reference to the first element in the container.

Returns Const reference to the first element

inline auto **back**() const

Returns Const reference to the last element.

inline constexpr size_t **size**() const noexcept
Returns the number of words in the array.

Note: The most significant word might be be used entirely. The most significant bits might be masked away.

Returns The number of words used store the number

inline constexpr void **fill**(const word_type &value)
Assigns the specified value to all elements in the container.

Parameters **value** – the value to assign to the elements

inline bool constexpr **is_zero**() const noexcept
Tests if all bits are zero.

Returns True iff all bits are zero

Public Static Functions

static inline constexpr *word_array*<*Width*, *WordType*> **all_ones**()
Creates a word array consisting of ones only.

Returns <111…..11>

static inline constexpr *word_array*<*Width*, *WordType*> **msb_one**()
Creates a word array with only the most significant bit being one.

Returns <10000….00>

static inline constexpr *word_array*<*Width*, *WordType*> **all_zeroes**()
Creates a word array consisting of zeroes only.

Returns <000…00>

16.2 String and Number Utilities

This is a collection of the most basic string and number utilities that are used throughout aarith.

16.2.1 String Utilities

Header `aarith/core/core_string_utils`

namespace **aarith**

Functions

inline constexpr auto **number_of_decimal_digits**(size_t n_bits) -> size_t
Computes an approximation of the decimal deigits an n_bits number will use.

Parameters **n_bits** – The number of bits in the number

Returns The approximation of the number of decimal digits the n_bits number will have

template<class T, class U> constexpr auto rounded_integer_division (T dividend, U divisor) -> decltype(T
Division with subsequent ceil oberation.

Computes ceil(dividend/divisor)

Template Parameters

- **T** – Type of the dividend
- **U** – Type of the divisor

Parameters

- **dividend** – The actual dividend
- **divisor** – The actual divisor

Returns Returns ceil(dividend/divisor)

WordType auto **to_binary**(const *word_array*<Width, WordType> &value) -> std::string

template<size_t **W**, typename **WordType**, template<size_t, typename> class **WA**, typename = std::enable_if_t<*is_word_array_v*<*WA*<*W*, *WordType*>> && !*is_integral_v*<*WA*<*W*, *WordType*>>>>
auto **operator<<**(std::ostream &out, const *WA*<*W*, *WordType*> &value) -> std::ostream&
Outputs a *word_array* to an output stream using the convenient << operator form.

Template Parameters

- **W** – Width of the word array
- **WordType** – The type used to store the actual data
- **WA** – Generic word array type

Variables

U

16.2.2 Number Utilities

Header aarith/core/core_number_utils.hpp

namespace **aarith**

Functions

template<unsigned **Base**, unsigned **Value**>
constexpr double **log**()
Constexpr version of log.

This method is necessary as not all compilers already come with a constexpr ready math library.

Note: Only two values needed by aarith are epxlictly stored!

Template Parameters

- **Base** – The base of the loarithm
- **Value** – The value whose logarithm is to be computed

Returns The logarithm of Value base Base

template<>
constexpr double **log**<2, 10>()
The constexpr value of log_2(10)

Returns log_2(10)

template<>
constexpr double **log**<10, 2>()
The constexpr value of log_10(2)

Returns log_10(2)

template<class **Result**>
constexpr *Result* **ceil**(double num)
Constexpr version of the ceil operation.

This method is only necessary as not all compilers already have an constexpr ready math library.

Template Parameters **Result** – Type for the result

Parameters **num** – The number to round down

Returns num rounded down

template<class **Result**>
constexpr *Result* **floor**(double num)

constexpr size_t **pow**(const size_t base, const size_t exponent)
Exponentiation function.

Note: This function does not make any attempts to be fast or to prevent overflows!

Note: If exponent equals std::numeric_limits<size_t>::max(), this method throws an exception, unless base equals zero

Parameters

- **base** –
- **exponent** –

Returns The base to the power of the exponent

constexpr size_t **first_set_bit**(const size_t n)

constexpr size_t **floor_to_pow**(const size_t n)
Rounds down to the next power of two.

TODO (keszocze) remove this method when clang supports constexpr for std::log2 and std::floor

Parameters **n** – The number to round

Returns The largest number m=2^k for some k such that m <= n

16.3 Type Traits

Header `aarith/core/traits.hpp`

Unfortunately, adding specializations for type traits such as `is_integral` is undefined.

Hence we copy the relevant traits into the `aarith` namespace.

namespace **aarith**

Variables

template<class **Type**>

constexpr bool **is_word_array_v** = *is_word_array*<*Type*>::value
Test for a type being an word array.

Helper for the *is_word_array* type trait

Template Parameters **Type** – Type to check for being an aarith word array

template<class **Type**>

constexpr bool **is_integral_v** = *is_integral*<*Type*>::value
Test for a type being an aarith integer.

Helper for the *is_integral* type trait

Template Parameters **Type** – The type to check

template<class **Type**>

constexpr bool **is_unsigned_v** = *is_unsigned*<*Type*>::value
Test for an aarith type to be unsigned.

Template Parameters **Type** – The type to check

template<class **Type**>

constexpr bool **is_signed_v** = *is_signed*<*Type*>::value
Test for an aarith type to be signed.

Template Parameters **Type** – The type to check

template<class **T**>

constexpr bool **is_arithmetic_v** = *is_arithmetic*<*T*>::value
Test for an aarith type being an arithmetic type.

Template Parameters **T** – The type to check

template<class **Type**>

constexpr bool **is_float_v** = *is_float*<*Type*>::value
Tests if a type is an aarith *floating_point*.

Helper for the *is_float* type trait

Template Parameters **Type** – The type to check for being an aarith *floating_point*

template<typename **A**, typename **B**>

constexpr bool **same_word_type** = std::is_same_v<typename *A*::word_type, typename *B*::word_type>
Type trait to check if two types use the same word type to store the data.

Template Parameters

- **A** – First type
- **B** – Second type

template<typename **A**, typename **B**>

constexpr bool **same_signedness** = (*is_unsigned_v*<*A*> == *is_unsigned_v*<*B*>)
Type trait to check if two types have the same 'signedness'.

It returns true if and only if both types are signed or unsigned

Template Parameters

- **A** – First type
- **B** – Second type

template<typename **T**>

constexpr bool **is_unsigned_int** = std::is_same_v<*T*, std::size_t> || std::is_same_v<*T*, uint64_t> || std::is_same_v<*T*, uint32_t> || std::is_same_v<*T*, uint16_t> || std::is_same_v<*T*, uint8_t>
Type trait to check a type for being an unsigned integer.

It seems that the type traits of C++ have no reasonable concept of "unsigned integer" so we have to add this ourselves.

Template Parameters `T` – The type to check for "unsigned int'nes"

template<class `T`>

struct `is_arithmetic`

#include <traits.hpp> Type trait for an aarith type being an arithmetic type.

Template Parameters `T` – The type to check

Public Static Attributes

static constexpr bool `value` = false

By default, no type is an arithmetic type

template<class `Type`>

class `is_float`

#include <traits.hpp> Type trait to check if a type is an aarith *floating_point* number.

Note: This does *not* return true for the native data types such as float or double!

Template Parameters `Type` – The type to check

Public Static Attributes

static constexpr bool `value` = false

By default, a type is not of type aarith *floating_point*

template<class `Type`>

class `is_integral`

#include <traits.hpp> Type trait to check if a type is an aarith integer.

Template Parameters `Type` – The Type to check

Public Static Attributes

static constexpr bool `value` = false

By default, mo type is an aarith integer

template<class `Type`>

class `is_signed`

#include <traits.hpp> Type trait to check if a type is a signed aarith type.

Template Parameters `Type` – The type to check

Public Static Attributes

static constexpr bool **value** = false
By default, no type is an signed aarith type

template<class **Type**>

class **is_unsigned**
#include <traits.hpp> Type trait to check if a type is an unsigned aarith type.

Template Parameters **Type** – The type to check

Public Static Attributes

static constexpr bool **value** = false
By default, no type is an unsigned type

template<class **Type**>

class **is_word_array**
#include <traits.hpp> Type trait to check if a type is a word array.

See also:

word_array

Template Parameters **Type** – The type to check

Public Static Attributes

static constexpr bool **value** = false
By default, no type is an aarith word array

16.4 Bit Cast

Header `aarith/core/bit_cast.hpp`

Unfortunately, there is no easy way to completely re-interpret bits as a different type. There is the usual trick of using an union. This is undefined behavior. This is why we opted to use the solution from this talk.

In the near future, when C++20 is more widely availabe, we will switch to using std::bit_cast

namespace **aarith**

Functions

template<typename **To**, typename **From**, typename = std::enable_if_t<(sizeof(*To*) == sizeof(*From*)) && std::is_trivially_copyable_v<*From*> && std::is_trivially_copyable_v<*To*>>>
To **bit_cast**(const *From* &src) noexcept

To avoid undefined behaviour when type punning, we are using memcpy. This is rather annoying but seems to be the way to go. The implementation is stolen from this talk: https://www.youtube.com/watch?v=_qzMpk-22cc

Todo:
replace with std::bit_cast when switching to C++20

Template Parameters

- **To** – The type to convert to
- **From** – The type to convert from

Parameters **src** – The source of the bits

Returns The new type filled with the bits of src

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[Keszocze2021] If you use Aarith (e.g., in your publication), please cite

Oliver Keszocze, Marcel Brand, Christian Heidorn, and Jürgen Teich. „Aarith: An Arbitrary Precision Number Library“, In: ACM/SIGAPP Symposium On Applied Computing (SAC'21). March 2021.

Bibtex:

```
@inproceedings{Keszocze2021,
  title = {Aarith: {{An Arbitrary Precision Number Library}}},
  booktitle = {ACM/SIGAPP Symposium On Applied Computing},
  author = {Keszocze, Oliver and Brand, Marcel and Heidorn, Christian and Teich,
↪ Jürgen},
  date = {2021-03},
  location = {{Virtual Event, South Korea}},
  series = {{{SAC}}'21}
}
```

[Brand2020] Brand, M., Witterauf, M., Bosio, A., & Teich, J. (2020, July). Anytime Floating-Point Addition and Multiplication-Concepts and Implementations. In 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP) (pp. 157-164). IEEE.

[Brand2019] Brand, M., Witterauf, M., Hannig, F., & Teich, J. (2019, April). Anytime instructions for programmable accuracy floating-point arithmetic. In Proceedings of the 16th ACM International Conference on Computing Frontiers (pp. 215-219).

[Burgess2019] Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K., & Mansell, D. (2019, June). Bfloat16 processing for neural networks. In 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH) (pp. 88-91). IEEE.

[IEEE754] 754-2019 - IEEE Standard for Floating-Point Arithmetic

[Echavarria2016] Echavarria, J., Wildermann, S., Becher, A., Teich, J., & Ziener, D. (2016, December). Fau: Fast and error-optimized approximate adder units on lut-based fpgas. In 2016 International Conference on Field-Programmable Technology (FPT) (pp. 213-216). IEEE.

INDEX

A