Merge data-parallel types from the Parallelism TS 2

ABSTRACT

After the Parallelism TS 2 was published in 2018, data-parallel types (`simd<T>`) have been implemented and used, and we are receiving feedback, this paper proposes to merge Section 9 of the Parallelism TS 2 into the IS working draft.

CONTENTS

1 Introduction 1
2 Changes after TS feedback 1
3 Wording 3
A Bibliography 33
1 INTRODUCTION

[P0214R9] introduced simd<T> and related types and functions into the Parallelism TS 2 Section 9. The TS was published in 2018. An incomplete and non-conforming (because P0214 evolved) implementation existed for the whole time P0214 progressed through the committee. Shortly after the GCC 9.1.0 release, a complete implementation of Section 9 of the TS was made available.

Note: This paper is not yet proposing the merge, but is aiming to start the process and raise awareness. Later revisions will actually call for a merge.

1.1 related papers

P0350 Before publication of the TS, SG1 approved [P0350R0] which did not progress in time in LEWG to make it into the TS. P0350 is moving forward independently.

P0918 After publication of the TS, SG1 approved [P0918R2] which adds shuffle, interleave, sum_to, multiply_sum_to, and saturated_simd_cast. P0918 will move forward independently.

Both papers currently have no shipping vehicle and are basically blocked on this paper.

2 CHANGES AFTER TS FEEDBACK

This section is mostly a stub. [P1915R0] (Expected Feedback from simd in the Parallelism TS 2) was just published and asks for specific feedback. After gathering feedback, the relevant changes will be added to a new revision of this paper.

2.1 missing SIMD_MASK generator constructor

The simd generator constructor is very useful for initializing objects from scalars in a portable (different size()) fashion. The need for a similar constructor for simd_mask is less frequent, but if only for consistency, there should be one. Besides consistency, it is also useful, of course. Consider a predicate function that is given without simd interface (e.g. from a library). How do you construct a simd_mask from it? With a generator constructor it is easy:

```cpp
simd<T> f(simd<T> x, Predicate p) {
    const simd_mask<T> k({[](auto i) { return p(x[i]); }}); 
    where(k, x) = 0;
}
```
2 Changes after TS feedback

```cpp
return x;
}
```

Without the generator constructor one has to write e.g.:

```cpp
simd<T> f(simd<T> x, Predicate p) {
    simd_mask<T> k;
    for (size_t i = 0; i < simd<T>::size(); ++i) {
        k[i] = p(x[i]);
    }
    where(k, x) = 0;
    return x;
}
```

The latter solution makes it hard to initialize the `simd_mask` as `const`, is more verbose, is harder to optimize, and cannot use the sequencing properties the generator constructor allows.

Therefore add:

```cpp
template<class G> simd_mask(G&& gen) noexcept;
```

### 2.2 add missing casts for `SIMD_MASK`

The `simd_cast` and `static_simd_cast` overloads for `simd_mask` were forgotten for the TS. Without those casts (and no casts via constructors) mixing different arithmetic types is painful. There is no motivation for forbidding casts on `simd_mask`.

Therefore add the following overloads:

```cpp
template<class T, class U, class Abi> see below simd_cast(const simd_mask<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below static_simd_cast(const simd_mask<U, Abi>&) noexcept;
```

### 2.3 `ELEMENT_REFERENCE` is overspecified

`element_reference` is spelled out in a lot of detail. It may be better to define its requirements in a table instead.

This change is not reflected in the wording, pending encouragement from WG21 (mostly LWG).
2.4 clean up math function overloads

The wording that produces simd overloads misses a few cases and leaves room for ambiguity. There is also no explicit mention of integral overloads that are supported in <cmath> (e.g. std::cos(1) calling std::cos(double)).

This needs more work and is not reflected in the wording at this point.

3 WORDING

The following applies the wording to the WD. It adds all of Section 9 out of N4808 into the IS WD without any markup since it is a strict addition. Changes relative to N4808, which contains editorial changes after the publication of the TS, are marked using color for additions and removals.

3.1 add section 9 of n4808 with modifications

Add a new subclause after §26.8 [c.math]

(3.1.1) 26.9 Data-Parallel Types

(3.1.1.1) 26.9.1 General

1 The data-parallel library consists of data-parallel types and operations on these types. A data-parallel type consists of elements of an underlying arithmetic type, called the element type. The number of elements is a constant for each data-parallel type and called the width of that type.

2 Throughout this Clause, the term data-parallel type refers to all supported (1) specializations of the simd and simd_mask class templates. A data-parallel object is an object of data-parallel type.

3 An element-wise operation applies a specified operation to the elements of one or more data-parallel objects. Each such application is unsequenced with respect to the others. A unary element-wise operation is an element-wise operation that applies a unary operation to each element of a data-parallel object. A binary element-wise operation is an element-wise operation that applies a binary operation to corresponding elements of two data-parallel objects.

4 Throughout this Clause, the set of vectorizable types for a data-parallel type comprises all cv-unqualified arithmetic types other than bool.

5 Remark: The intent is to support acceleration through data-parallel execution resources, such as SIMD registers and instructions or execution units driven by a common instruction decoder. If such execution resources are unavailable, the interfaces support a transparent fallback to sequential execution.

(3.1.1.2) 26.9.2 Header <experimental/simd> synopsis

[simd.synopsis]
namespace std::experimental {
inline namespace parallelism_v2 {
namespace simd_abi {
    using scalar = see below;
    template<int N> using fixed_size = see below;
    template<class T> inline constexpr int max_fixed_size = implementation-defined;
    template<class T> using compatible = implementation-defined;
    template<class T> using native = implementation-defined;

    template<class T, size_t N, class... Abis> struct deduce { using type = see below; };
    template<class T, size_t N, class... Abis> using deduce_t = typename deduce<T, N, Abis...>::type;
}

struct element_aligned_tag {};
struct vector_aligned_tag {};
template<size_t> struct overaligned_tag {};
inline constexpr element_aligned_tag element_aligned {};
inline constexpr vector_aligned_tag vector_aligned {};
template<size_t N> inline constexpr overaligned_tag<N> overaligned {};

#endif

// 26.9.4, simd type traits
    template<class T> struct is_abi_tag;
    template<class T> inline constexpr bool is_abi_tag_v = is_abi_tag<T>::value;

    template<class T> struct is_simd;
    template<class T> inline constexpr bool is_simd_v = is_simd<T>::value;

    template<class T> struct is_simd_mask;
    template<class T> inline constexpr bool is_simd_mask_v = is_simd_mask<T>::value;

    template<class T> struct is_simd_flag_type;
    template<class T> inline constexpr bool is_simd_flag_type_v = is_simd_flag_type<T>::value;

    template<class T, class Abi = simd_abi::compatible<T>> struct simd_size;
    template<class T, class Abi = simd_abi::compatible<T>> inline constexpr size_t simd_size_v = simd_size<T,Abi>::value;

    template<class T, class U = typename T::value_type> struct memory_alignment;
    template<class T, class U = typename T::value_type> inline constexpr size_t memory_alignment_v = memory_alignment<T,U>::value;

    template<class T, class V> struct rebind_simd { using type = see below; };
    template<class T, class V> using rebind_simd_t = typename rebind_simd<T, V>::type;

    template<int N, class V> struct resize_simd { using type = see below; };
    template<int N, class V> using resize_simd_t = typename resize_simd<N, V>::type;

#endif

// 26.9.6, Class template simd
    template<class T, class Abi = simd_abi::compatible<T>> class simd{
        template<class T> using native_simd = simd<T, simd_abi::native<T>>;
        template<class T, int N> using fixed_size_simd = simd<T, simd_abi::fixed_size<N>>;
    }

// 26.9.8, Class template simd_mask
    template<class T, class Abi = simd_abi::compatible<T>> class simd_mask{
        template<class T> using native_simd_mask = simd_mask<T, simd_abi::native<T>>;
        template<class T, int N> using fixed_size_simd_mask = simd_mask<T, simd_abi::fixed_size_mask<N>>;
    }
}
// 5. Casts

```cpp
template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below static_simd_cast(const simd<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below simd_cast(const simd_mask<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below static_simd_cast(const simd_mask<U, Abi>&) noexcept;
```

```cpp
template<class T, class Abi>
to_fixed_size(const simd<T, Abi>&) noexcept;
to_fixed_size(const simd_mask<T, Abi>&) noexcept;
```

```cpp
to_native(const fixed_size_simd<T, Abi>&) noexcept;
to_native(const fixed_size_simd_mask<T, Abi>&) noexcept;
```

```cpp
to_compatible(const fixed_size_simd<T, Abi>&) noexcept;
to_compatible(const fixed_size_simd_mask<T, Abi>&) noexcept;
```

```cpp
to_5<...> Sizes, class T, class Abi>
tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
split(const simd<T, Abi>&) noexcept;
tuple<simd_mask<T, simd_mask_abi::deduce_t<T, Sizes>>...>
split(const simd_mask<T, Abi>&) noexcept;
```

```cpp
to_5<...> Sizes, class T, class Abi>
tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
split(const simd<T, Abi>&) noexcept;
tuple<simd_mask<T, simd_mask_abi::deduce_t<T, Sizes>>...>
split(const simd_mask<T, Abi>&) noexcept;
```

```cpp
tuple< simd<T, simd_abi::deduce_t<T, N>>...>
split(const simd<T, Abi>&) noexcept;
tuple< simd_mask<T, simd_mask_abi::deduce_t<T, N>>...>
split(const simd_mask<T, Abi>&) noexcept;
```

```cpp
// 4. Reductions

```cpp
template<class T, class Abi> bool all_of(const simd_mask<T, Abi>&) noexcept;
```
template<class T, class Abi> bool any_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> bool none_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> bool some_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> int popcount(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> int find_first_set(const simd_mask<T, Abi>&);
template<class T, class Abi> int find_last_set(const simd_mask<T, Abi>&);

bool all_of(T) noexcept;
bool any_of(T) noexcept;
bool none_of(T) noexcept;
bool some_of(T) noexcept;
int popcount(T) noexcept;
int find_first_set(T);
int find_last_set(T);

// 26.9.5, Where expression class templates
template<class M, class T> class const_where_expression;
template<class M, class T> class where_expression;

// 5. Where functions
template<class T, class Abi>
where_expression<simd_mask<T, Abi>, simd<T, Abi>>
where(const typename simd<T, Abi>::mask_type &, simd<T, Abi>&) noexcept;

template<class T, class Abi>
const_where_expression<simd_mask<T, Abi>, simd<T, Abi>>
where(const typename simd<T, Abi>::mask_type &, const simd<T, Abi>&) noexcept;

template<class T, class Abi>
where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
where(const type_identity_t<simd_mask<T, Abi>>&, simd_mask<T, Abi>&) noexcept;

template<class T, class Abi>
const_where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
where(const type_identity_t<simd_mask<T, Abi>>&, const simd_mask<T, Abi>&) noexcept;

template<class T>
where_expression<bool, T>
where(see below k, T d) noexcept;

template<class T>
const_where_expression<bool, T>
where(see below k, const T d) noexcept;

// 4. Reductions
template<class T, class Abi, class BinaryOperation = plus>>
T reduce(const simd<T, Abi>&, BinaryOperation = {});

template<class M, class V, class BinaryOperation>
typename V::value_type reduce(const const_where_expression<M, V>& x,
   typename V::value_type identity_element,
   BinaryOperation binary_op);

template<class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x,
   plus<> binary_op = ||) noexcept;

template<class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x,
The header `<experimental/simd>` defines class templates, tag types, trait types, and function templates for element-wise operations on data-parallel objects.

### 26.9.3 simd ABI tags

namespace simd_abi {
    using scalar = see below;
    template<int N> using fixed_size = see below;
    template<class T> inline constexpr int max_fixed_size = implementation-defined;
    template<class T> using compatible = implementation-defined;
    template<class T> using native = implementation-defined;
}

An ABI tag is a type in the `std::experimental::parallelism_v2::simd_abi` namespace that indicates a choice of size and binary representation for objects of data-parallel type. **Remark:** The intent is for the size and binary representation to depend on the target architecture. The ABI tag, together with a given element type implies a number of elements. ABI tag types are used as the second template argument to `simd` and `simd_mask`. **Remark:** The ABI tag is orthogonal to selecting the machine instruction set. The selected machine instruction set limits the usable ABI tag types, though (see 1). The ABI tags enable users to safely pass objects of data-parallel type between translation unit boundaries (e.g. function calls or I/O).
3 scalar is an alias for an unspecified ABI tag that is different from fixed_size<1>. Use of the scalar tag type requires data-parallel types to store a single element (i.e., simd<T, simd_abi::scalar>::size() returns 1).
4 The value of max_fixed_size<T> is at least 32.
5 fixed_size<N> is an alias for an unspecified ABI tag. fixed_size does not introduce a non-deduced context. Use of the simd_abi::fixed_size<N> tag type requires data-parallel types to store N elements (i.e., simd<T, simd_abi::fixed_size<N>>::size() is N). simd<T, fixed_size<N>> and simd_mask<T, fixed_size<N>> with N > 0 and N <= max_fixed_size<T> shall be supported. Additionally, for every supported simd<T, Abi> (see 1), where Abi is an ABI tag that is not a specialization of simd_abi::fixed_size, N == simd<T, Abi>::size() shall be supported.
6 Remark: It is unspecified whether simd<T, fixed_size<N>> with N > max_fixed_size<T> is supported. The value of max_fixed_size<T> can depend on compiler flags and can change between different compiler versions.
7 Remark: An implementation can forego ABI compatibility between differently compiled translation units for simd and simd_mask specializations using the same simd_abi::fixed_size<N> tag. Otherwise, the efficiency of simd<T, Abi> is likely to be better than for simd<T, fixed_size<N>_v<T, Abi>> (with Abi not a specialization of simd_abi::fixed_size).
8 An implementation may define additional extended ABI tag types in the std::experimental::parallelism_v2::simd_abi namespace, to support other forms of data-parallel computation.
9 compatible<T> is an implementation-defined alias for an ABI tag. Remark: The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type T that ensures ABI compatibility between translation units on the target architecture. [Example: Consider a target architecture supporting the extended ABI tags __simd128 and __simd256, where the __simd256 type requires an optional ISA extension on said architecture. Also, the target architecture does not support long double with either ABI tag. The implementation therefore defines compatible<T> as an alias for:
   • scalar if T is the same type as long double, and
   • __simd128 otherwise.
   — end example]
10 native<T> is an implementation-defined alias for an ABI tag. Remark: The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type T that is supported on the currently targeted system. For target architectures without ISA extensions, the native<T> and compatible<T> aliases will likely be the same. For target architectures with ISA extensions, compiler flags may influence the native<T> alias while compatible<T> will be the same independent of such flags. [Example: Consider a target architecture supporting the extended ABI tags __simd128 and __simd256, where hardware support for __simd256 only exists for floating-point types. The implementation therefore defines native<T> as an alias for
   • __simd256 if T is a floating-point type, and
   • __simd128 otherwise.
   — end example]
11 template<T, size_t N, class... Abis> struct deduce { using type = see below; };
12 The member type shall be present if and only if
   • T is a vectorizable type, and
   • simd_abi::fixed_size<N> is supported (see 26.9.3), and
   • every type in the Abis pack is an ABI tag.
Where present, the member typedef `type` shall name an ABI tag type that satisfies

- `simd_size<T, type> == N`, and
- `simd<T, type>` is default constructible (see 1).

If `N` is 1, the member typedef `type` is `simd_abi::scalar`. Otherwise, if there are multiple ABI tag types that satisfy the constraints, the member typedef `type` is implementation-defined. **Remark:** It is expected that extended ABI tags can produce better optimizations and thus are preferred over `simd_abi::fixed_size<N>`. Implementations can base the choice on `Abis`, but can also ignore the `Abis` arguments.

The behavior of a program that adds specializations for `deduce` is undefined.

(3.1.1.4) 26.9.4 simd type traits

```cpp
template<class T> struct is_abi_tag { see below };  
1 The type `is_abi_tag<T>` is a UnaryTypeTrait with a base characteristic of `true_type` if `T` is a standard or extended ABI tag, and `false_type` otherwise.
2 The behavior of a program that adds specializations for `is_abi_tag` is undefined.

template<class T> struct is_simd { see below };  
3 The type `is_simd<T>` is a UnaryTypeTrait with a base characteristic of `true_type` if `T` is a specialization of the `simd` class template, and `false_type` otherwise.
4 The behavior of a program that adds specializations for `is_simd` is undefined.

template<class T> struct is_simd_mask { see below };  
5 The type `is_simd_mask<T>` is a UnaryTypeTrait with a base characteristic of `true_type` if `T` is a specialization of the `simd_mask` class template, and `false_type` otherwise.
6 The behavior of a program that adds specializations for `is_simd_mask` is undefined.

template<class T> struct is_simd_flag_type { see below };  
7 The type `is_simd_flag_type<class T>` is a UnaryTypeTrait with a base characteristic of `true_type` if `T` is one of
   - `element_aligned_tag`, or
   - `vector_aligned_tag`, or
   - `overaligned_tag<N>` with `N > 0` and `N` an integral power of two, and
     `false_type` otherwise.
8 The behavior of a program that adds specializations for `is_simd_flag_type` is undefined.

template<class T, class Abi = simd_abi::compatible<T>> struct simd_size { see below };  
9 `simd_size<T, Abi>` shall have a member `value` if and only if
   - `T` is a vectorizable type, and
   - `is_abi_tag_v<Abi>` is true.
```
Remark: The rules are different from those in (1).

If value is present, the type simd_size<T, Abi> is a BinaryTypeTrait with a base characteristic of integral_constant<size_t, N> with N equal to the number of elements in a simd<T, Abi> object.

Remark: If simd<T, Abi> is not supported for the currently targeted system, simd_size<T, Abi>::value produces the value simd<T, Abi>::size() would return if it were supported.

The behavior of a program that adds specializations for simd_size is undefined.

template<class T, class U = typename T::value_type> struct memory_alignment { see below; }

memory_alignment<T, U> shall have a member value if and only if
- is_simd_mask_v<T> is true and U is bool, or
- is_simd_v<T> is true and U is a vectorizable type.

If value is present, the type memory_alignment<T, U> is a BinaryTypeTrait with a base characteristic of integral_constant<size_t, N> for some implementation-defined N (see 5 and 4). Remark: value identifies the alignment restrictions on pointers used for (converting) loads and stores for the give type T on arrays of type U.

The behavior of a program that adds specializations for memory_alignment is undefined.

template<class T, class V> struct rebind_simd { using type = see below; }

The member type is present if and only if
- V is either simd<U, Abi0> or simd_mask<U, Abi0>, where U and Abi0 are deduced from V, and
- T is a vectorizable type, and

Let Abi1 denote the type deduce_t<T, simd_size_v<U, Abi0>, Abi0>. Where present, the member typedef type names simd<T, Abi1> if V is simd<U, Abi0> or simd_mask<T, Abi1> if V is simd_mask<T, Abi0>.

template<int N, class V> struct resize_simd { using type = see below; }

The member type is present if and only if
- V is either simd<T, Abi0> or simd_mask<T, Abi0>, where T and Abi0 are deduced from V, and
- simd_abi::deduce<T, N, Abi0> has a member type type.

Let Abi1 denote the type deduce_t<T, N, Abi0>. Where present, the member typedef type names simd<T, Abi1> if V is simd<T, Abi0> or simd_mask<T, Abi1> if V is simd_mask<T, Abi0>.

(3.1.5) 26.9.5 Where expression class templates

template<class M, class T> class const_where_expression {
    const M mask;  // exposition only
    T& data;       // exposition only

public:
    const_where_expression(const const_where_expression&) = delete;
    const_where_expression& operator=(const const_where_expression&) = delete;
The class templates `const_where_expression` and `where_expression` abstract the notion of selecting elements of a given object of arithmetic or data-parallel type.

The first templates argument `M` shall be cv-unqualified `bool` or a cv-unqualified `simd_mask` specialization.

If `M` is `bool`, `T` shall be a cv-unqualified arithmetic type. Otherwise, `T` shall either be `M` or `typename M::simd_type`.

In this subclause, if `M` is `bool`, `data[0]` is used interchangeably for `data`, `mask[0]` is used interchangeably for `mask`, and `M::size()` is used interchangeably for `1`.

The selected indices signify the integers \(i \in \{j \in \mathbb{N} | j < M::size() \land mask[j]\}\). The selected elements signify the elements `data[i]` for all selected indices \(i\).

In this subclause, the type `value_type` is an alias for `T` if `M` is `bool`, or an alias for `typename T::value_type` if `is_simd_mask_v<M>` is true.

Remark: The `where` functions initialize `mask` with the first argument to `where` and `data` with the second argument to `where`.

Returns: A copy of `data` with the indicated unary operator applied to all selected elements.

```
t operator-() const && noexcept;
T operator+() const && noexcept;
T operator~() const && noexcept;
```

```
t template<class U, class Flags> void copy_to(U* mem, Flags f) const &&;
```

```
t template<class M, class T>
class where_expression : public const_where_expression <M, T> {
public:
    template<class U> void operator=(U&& x) && noexcept;
    template<class U> void operator+=(U&& x) && noexcept;
    template<class U> void operator-=(U&& x) && noexcept;
    template<class U> void operator*=(U&& x) && noexcept;
    template<class U> void operator/=(U&& x) && noexcept;
    template<class U> void operator%=(U&& x) && noexcept;
    template<class U> void operator&=(U&& x) && noexcept;
    template<class U> void operator|=(U&& x) && noexcept;
    template<class U> void operator^=(U&& x) && noexcept;
    template<class U> void operator<<=(U&& x) && noexcept;
    template<class U> void operator>>=(U&& x) && noexcept;
    void operator++() && noexcept;
    void operator++(int) && noexcept;
    void operator--() && noexcept;
    void operator--(int) && noexcept;

    template<class U, class Flags> void copy_from(const U* mem, Flags ) &&;
};
```

```
1 The class templates `const_where_expression` and `where_expression` abstract the notion of selecting elements of a given object of arithmetic or data-parallel type.
2 The first templates argument `M` shall be cv-unqualified `bool` or a cv-unqualified `simd_mask` specialization.
3 If `M` is `bool`, `T` shall be a cv-unqualified arithmetic type. Otherwise, `T` shall either be `M` or `typename M::simd_type`.
4 In this subclause, if `M` is `bool`, `data[0]` is used interchangeably for `data`, `mask[0]` is used interchangeably for `mask`, and `M::size()` is used interchangeably for `1`.
5 The selected indices signify the integers \(i \in \{j \in \mathbb{N} | j < M::size() \land mask[j]\}\). The selected elements signify the elements `data[i]` for all selected indices \(i\).
6 In this subclause, the type `value_type` is an alias for `T` if `M` is `bool`, or an alias for `typename T::value_type` if `is_simd_mask_v<M>` is true.
7 Remark: The `where` functions initialize `mask` with the first argument to `where` and `data` with the second argument to `where`.
8 Returns: A copy of `data` with the indicated unary operator applied to all selected elements.
9 Requires:
   - If `M` is not `bool`, the largest selected index is less than the number of values pointed to by `mem`.```
• If the template parameter Flags is `vector_aligned_tag`, mem shall point to storage aligned by `memory_alignment_v<T, U>`.
• If the template parameter Flags is `overaligned_tag<N>`, mem shall point to storage aligned by N.
• If the template parameter Flags is `element_aligned_tag`, mem shall point to storage aligned by `alignof(U)`.

**Effects:** Copies the selected elements as if `mem[i] = static_cast<U>({data[i]})` for all selected indices i.

**Throws:** Nothing.

**Remarks:** This function shall not participate in overload resolution unless
- `is_simd_flag_type_v<Flags>` is true, and
- either
  - U is bool and value_type is bool, or
  - U is a vectorizable type and value_type is not bool.

```
template<class U> void operator=( U&& x) && noexcept ;

**Effects:** Replaces data[i] with `static_cast<T>({std::forward<U>{x}})[i]` for all selected indices i.

**Remarks:** This operator shall not participate in overload resolution unless U is convertible to T.
```
```template<class U> void operator+=( U&& x) && noexcept ;
template<class U> void operator-=( U&& x) && noexcept ;
template<class U> void operator*=( U&& x) && noexcept ;
template<class U> void operator/=( U&& x) && noexcept ;
template<class U> void operator%=( U&& x) && noexcept ;
template<class U> void operator&=( U&& x) && noexcept ;
template<class U> void operator|=( U&& x) && noexcept ;
template<class U> void operator^=( U&& x) && noexcept ;
template<class U> void operator<<=( U&& x) && noexcept ;
template<class U> void operator>>=( U&& x) && noexcept ;
```  

```
**Effects:** Replaces data[i] with `static_cast<T>({data @ std::forward<U>{x}})[i]` (where @ denotes the indicated operator) for all selected indices i.

**Remarks:** Each of these operators shall not participate in overload resolution unless the return type of `data @ std::forward<U>{x}` is convertible to T. It is unspecified whether the binary operator, implied by the compound assignment operator, is executed on all elements or only on the selected elements.
```
```void operator++() && noexcept ;
void operator++(int) && noexcept ;
void operator--() && noexcept ;
void operator--(int) && noexcept ;
```  

```
**Effects:** Applies the indicated operator to the selected elements.

**Remarks:** Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type T.
```
```template<class U, class Flags> void copy_from(const U* mem, Flags) &&;```
```
Requires:

- If \( \text{is\_simd\_flag\_type\_v<U> is true} \), for all selected indices \( i \), \( i \) shall be less than the number of values pointed to by \( \text{mem} \).
- If the template parameter \( \text{Flags is vector\_aligned\_tag} \), \( \text{mem} \) shall point to storage aligned by \( \text{memory\_alignment\_v<T, U>} \).
- If the template parameter \( \text{Flags is overaligned\_tag<N>} \), \( \text{mem} \) shall point to storage aligned by \( N \).
- If the template parameter \( \text{Flags is element\_aligned\_tag} \), \( \text{mem} \) shall point to storage aligned by \( \text{alignof(U)} \).

Effects: Replaces the selected elements as if \( \text{data[i] = static\_cast<value\_type>(mem[i]) for all selected indices i.} \)

Throws: Nothing.

Remarks: This function shall not participate in overload resolution unless

- \( \text{is\_simd\_flag\_type\_v<Flags> is true, and} \)
- either
  - \( U \) is \text{bool} and \( \text{value\_type is bool, or} \)
  - \( U \) is a vectorizable type and \( \text{value\_type is not bool.} \)

(3.1.1.6) 26.9.6 Class template \text{simd}

(3.1.1.6.1) 26.9.6.1 Class template \text{simd overview}

template<class T, class Abi> class simd {
public:
    using value_type = T;
    using reference = ...
    using mask_type = simd\_mask<T, Abi>;
    using abi_type = Abi;

    static constexpr size_t size() noexcept;
    simd() noexcept = default;

    // 4, simd\_constructors
    template<class U> simd(U& value) noexcept;
    template<class U> simd(const simd<U, simd\_abi::fixed\_size=size\_t&>&) noexcept;
    template<class G> explicit simd(G&& gen) noexcept;
    template<class U, class Flags> simd(const U* mem, Flags f);

    // 5, simd\_copy\_functions
    template<class U, class Flags> copy_from(const U* mem, Flags f);
    template<class U, class Flags> copy_to(U* mem, Flags f);

    // 6, simd\_subscript\_operators
    reference operator[](size_t);
    value_type operator[](size_t) const;

    // 7, simd\_unary\_operators
    simd\_operator\++() noexcept;
    simd\_operator\++(int) noexcept;
simd\& operator--() noexcept;
simd operator--(int) noexcept;
mask_type operator!(() const noexcept;
simd operator-() const noexcept;
simd operator+() const noexcept;
simd operator-() const noexcept;

// 1. simd binary operators
friend simd operator+(const simd\&, const simd\&) noexcept;
friend simd operator-(const simd\&, const simd\&) noexcept;
friend simd operator*(const simd\&, const simd\&) noexcept;
friend simd operator/(const simd\&, const simd\&) noexcept;
friend simd operator%(const simd\&, const simd\&) noexcept;
friend simd operator&(const simd\&, const simd\&) noexcept;
friend simd operator|(const simd\&, const simd\&) noexcept;
friend simd operator^(const simd\&, const simd\&) noexcept;
friend simd operator<<(const simd\&, const simd\&) noexcept;
friend simd operator>>(const simd\&, const simd\&) noexcept;
friend simd operator<<(const simd\&, int) noexcept;
friend simd operator>>(const simd\&, int) noexcept;

// 2. simd compound assignment
friend simd\& operator+=(simd\&, const simd\&) noexcept;
friend simd\& operator-=(simd\&, const simd\&) noexcept;
friend simd\& operator*=(simd\&, const simd\&) noexcept;
friend simd\& operator/=(simd\&, const simd\&) noexcept;
friend simd\& operator%=(simd\&, const simd\&) noexcept;
friend simd\& operator&=(simd\&, const simd\&) noexcept;
friend simd\& operator|=(simd\&, const simd\&) noexcept;
friend simd\& operator^=(simd\&, const simd\&) noexcept;
friend simd\& operator<<=(simd\&, const simd\&) noexcept;
friend simd\& operator>>=(simd\&, const simd\&) noexcept;
friend simd\& operator<<=(simd\&, int) noexcept;
friend simd\& operator>>=(simd\&, int) noexcept;

// 3. simd compare operators
friend mask_type operator==(const simd\&, const simd\&) noexcept;
friend mask_type operator!=(const simd\&, const simd\&) noexcept;
friend mask_type operator>=(const simd\&, const simd\&) noexcept;
friend mask_type operator<=(const simd\&, const simd\&) noexcept;
friend mask_type operator>(const simd\&, const simd\&) noexcept;
friend mask_type operator<(const simd\&, const simd\&) noexcept;

1 The class template simd is a data-parallel type. The width of a given simd specialization is a constant expression, determined by the template parameters.

2 Every specialization of simd shall be a complete type. The specialization simd<T, Abi> is supported if T is a vectorizable type and

- Abi is simd_abi::scalar, or
- Abi is simd_abi::fixed_size<N>, with N constrained as defined in 26.9.3.

If Abi is an extended ABI tag, it is implementation-defined whether simd<T, Abi> is supported. Remark: The intent is for implementations to decide on the basis of the currently targeted system.

If simd<T, Abi> is not supported, the specialization shall have a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. Otherwise, the following are true:
• is_nothrow_move_constructible_v<simd<T, Abi>>. and
• is_nothrow_move_assignable_v<simd<T, Abi>>. and
• is_nothrow_default_constructible_v<simd<T, Abi>>.

[Example: Consider an implementation that defines the extended ABI tags __simd_x and __gpu_y. When the compiler is invoked to translate to a machine that has support for the __simd_x ABI tag for all arithmetic types other than long double and no support for the __gpu_y ABI tag, then:
• simd<T, simd_abi::__gpu_y> is not supported for any T and has a deleted constructor.
• simd<long double, simd_abi::__simd_x> is not supported and has a deleted constructor.
• simd<double, simd_abi::__simd_x> is supported.
• simd<long double, simd_abi::scalar> is supported.

—end example]

3 Default initialization performs no initialization of the elements; value-initialization initializes each element with T(). Remark: Thus, default initialization leaves the elements in an indeterminate state.

4 Implementations should enable explicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class simd:
   explicit operator implementation-defined() const;
   explicit simd(const implementation-defined& init);

   [Example: Consider an implementation that supports the type __vec4f and the function __vec4f __vec4f-_addsub(__vec4f, __vec4f) for the currently targeted system. A user may require the use of __vec4f_addsub for maximum performance and thus writes:
   using V = simd<float, simd_abi::__simd128>;
   V addsub(V a, V b) {
     return static_cast<V>(__vec4f_addsub(static_cast<__vec4f>(a), static_cast<__vec4f>(b)));
   }

—end example]

(3.1.1.6.2) 26.9.6.2 simd width

static constexpr size_t size() noexcept;
1

Returns: The width of simd<T, Abi>.

(3.1.1.6.3) 26.9.6.3 Element references

A reference is an object that refers to an element in a simd or simd_mask object. reference::value_type is the same type as simd::value_type or simd_mask::value_type, respectively.

Class reference is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name.
class reference // exposition only
{
public:
  reference() = delete;
  reference(const reference&) = delete;

  operator value_type() const noexcept;

}
template<class U> reference operator=(U&& x) && noexcept;
template<class U> reference operator+=(U&& x) && noexcept;
template<class U> reference operator-=(U&& x) && noexcept;
template<class U> reference operator*=(U&& x) && noexcept;
template<class U> reference operator/=(U&& x) && noexcept;
template<class U> reference operator%=(U&& x) && noexcept;
template<class U> reference operator|=(U&& x) && noexcept;
template<class U> reference operator&=(U&& x) && noexcept;
template<class U> reference operator^=(U&& x) && noexcept;
template<class U> reference operator <<= (U&& x) && noexcept;
template<class U> reference operator >>= (U&& x) && noexcept;
template<class U> reference operator++() && noexcept;
value_type operator++(int) && noexcept;
template<class U> reference operator--() && noexcept;
value_type operator--(int) && noexcept;
friend void swap(reference&& a, reference&& b) noexcept;
friend void swap(value_type & a, reference&& b) noexcept;
friend void swap(reference&& a, value_type & b) noexcept;
};
operator value_type() const noexcept;

3 Returns: The value of the element referred to by *this.

template<class U> reference operator=(U&& x) && noexcept;

4 Effects: Replaces the referred to element in simd or simd_mask with static_cast<value_type>(std ::forward<U>(x)).

5 Returns: A copy of *this.

6 Remarks: This function shall not participate in overload resolution unless declval<value_type&>() =
std::forward<U>(x) is well-formed.

template<class U> reference operator+=(U&& x) && noexcept;
template<class U> reference operator-=(U&& x) && noexcept;
template<class U> reference operator*=(U&& x) && noexcept;
template<class U> reference operator/=(U&& x) && noexcept;
template<class U> reference operator%=(U&& x) && noexcept;
template<class U> reference operator|=(U&& x) && noexcept;
template<class U> reference operator&=(U&& x) && noexcept;
template<class U> reference operator^=(U&& x) && noexcept;
template<class U> reference operator <<= (U&& x) && noexcept;
template<class U> reference operator >>= (U&& x) && noexcept;

7 Effects: Applies the indicated compound operator to the referred to element in simd or simd_mask and
std::forward<U>(x).

8 Returns: A copy of *this.

9 Remarks: This function shall not participate in overload resolution unless declval<value_type&>() @=
std::forward<U>(x) (where @= denotes the indicated compound assignment operator) is well-formed.
reference operator++() && noexcept;
reference operator--() && noexcept;

**Effects:** Applies the indicated operator to the referred to element in `simd` or `simd_mask`.

**Returns:** A copy of `*this`.

**Remarks:** This function shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

value_type operator++(int) && noexcept;
value_type operator--(int) && noexcept;

**Effects:** Applies the indicated operator to the referred to element in `simd` or `simd_mask`.

**Returns:** A copy of the referred to element before applying the indicated operator.

**Remarks:** This function shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

friend void swap(reference&& a, reference&& b) noexcept;
friend void swap(value_type & a, reference&& b) noexcept;
friend void swap(reference&& a, value_type & b) noexcept;

**Effects:** Exchanges the values `a` and `b` refer to.

### 3.1.1.6.4 simd constructors

**Effects:** Constructs an object with each element initialized to the value of the argument after conversion to `value_type`.

**Remarks:** Let `From` denote the type `remove_cv_t<remove_reference_t<U>>`. This constructor shall not participate in overload resolution unless:

- `From` is a vectorizable type and every possibly value of `From` can be represented with type `value_type`, or
- `From` is not an arithmetic type and is implicitly convertible to `value_type`, or
- `From` is `int`, or
- `From` is `unsigned int` and `value_type` is an unsigned integral type.

**Effects:** Constructs an object where the `ith` element equals `static_cast<T>(x[i])` for all `i` in the range of `[0, size())`.

**Remarks:** This constructor shall not participate in overload resolution unless

- `abi_type` is `simd_abi::fixed_size<size()>`, and
- every possible value of `U` can be represented with type `value_type`, and
- if both `U` and `value_type` are integral, the integer conversion rank (??) of `value_type` is greater than the integer conversion rank of `U`.

17
template<class G> simd(G& gen) noexcept;

Effects: Constructs an object where the \(i\)th element is initialized to \(\text{gen} \left( \text{integral\_constant<\text{size\_t}, i\rangle} \right)\).

Remarks: This constructor shall not participate in overload resolution unless \(\text{simd} \left( \text{gen} \left( \text{integral\_constant<\text{size\_t}, i\rangle} \right) \right)\) is well-formed for all \(i\) in the range of \([0, \text{size()}]\). The calls to \text{gen} are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by \text{gen}(?).

template<class U, class Flags> simd(const U* mem, Flags);

Requires:
- \([\text{mem}, \text{mem} + \text{size()}]\) is a valid range.
- If the template parameter Flags is vector\_aligned\_tag, \text{mem} shall point to storage aligned by memory\_alignment\_v<simd, U>.
- If the template parameter Flags is overaligned\_tag<N>, \text{mem} shall point to storage aligned by \(N\).
- If the template parameter Flags is element\_aligned\_tag, \text{mem} shall point to storage aligned by alignof(U).

Effects: Constructs an object where the \(i\)th element is initialized to \(\text{static\_cast<T>(mem[i])}\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: This constructor shall not participate in overload resolution unless
- \(\text{is\_simd\_flag\_type\_v<\text{Flags}}\) is true, and
- \(U\) is a vectorizable type.

(3.1.6.5) 26.9.6.5 simd copy functions [simd.copy]

template<class U, class Flags> void copy_from(const U* mem, Flags);

Requires:
- \([\text{mem}, \text{mem} + \text{size()}]\) is a valid range.
- If the template parameter Flags is vector\_aligned\_tag, \text{mem} shall point to storage aligned by memory\_alignment\_v<simd, U>.
- If the template parameter Flags is overaligned\_tag<N>, \text{mem} shall point to storage aligned by \(N\).
- If the template parameter Flags is element\_aligned\_tag, \text{mem} shall point to storage aligned by alignof(U).

Effects: Replaces the elements of the simd object such that the \(i\)th element is assigned with \(\text{static\_cast<T>(mem[i])}\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: This function shall not participate in overload resolution unless
- \(\text{is\_simd\_flag\_type\_v<\text{Flags}}\) is true, and
- \(U\) is a vectorizable type.

template<class U, class Flags> void copy_to(U* mem, Flags) const;
Requires:
  • \([\text{mem}, \text{mem} + \text{size()}]\) is a valid range.
  • If the template parameter \(\text{Flags}\) is \text{vector_aligned_tag}, \text{mem} shall point to storage aligned by \text{memory_alignment_v<\text{simd}, \text{U}>}.
  • If the template parameter \(\text{Flags}\) is \text{overaligned_tag<N>}, \text{mem} shall point to storage aligned by \(N\).
  • If the template parameter \(\text{Flags}\) is \text{element_aligned_tag}, \text{mem} shall point to storage aligned by \(\text{alignof(U)}\).

Effects: Copies all \text{simd} elements as if \(\text{mem}[i] = \text{static_cast<}\text{U}\{\text{operator[]}(i)\}\})\) for all \(i\) in the range of \([0, \text{size()})\).

Remarks: This function shall not participate in overload resolution unless
  • \(\text{is_simd_flag_type_v<\text{Flags}>}\) is true, and
  • \(\text{U}\) is a vectorizable type.

(3.1.1.6.6) 26.9.6.6 \text{simd} subscript operators

\[\text{reference operator[]}(\text{size_t i});\]

\text{Requires:} \(i < \text{size()}.\)
\text{Returns:} A reference (see 3) referring to the \(i\)th element.
\text{Throws:} Nothing.

\[\text{value_type operator[]}(\text{size_t i}) \text{ const;}\]

\text{Requires:} \(i < \text{size()}.\)
\text{Returns:} The value of the \(i\)th element.
\text{Throws:} Nothing.

(3.1.1.6.7) 26.9.6.7 \text{simd} unary operators

Effects in this subclause are applied as unary element-wise operations.

\[\text{simd\& operator++()} \text{ noexcept;}\]

\text{Effects:} Increments every element by one.
\text{Returns:} \(*\text{this}.\)

\[\text{simd operator++(int) noexcept;}\]

\text{Effects:} Increments every element by one.
\text{Returns:} A copy of \(*\text{this}\) before incrementing.

\[\text{simd\& operator--()} \text{ noexcept;}\]
Effects: Decrements every element by one.
Returns: *this.

simd operator--(int) noexcept;

Effects: Decrements every element by one.
Returns: A copy of *this before decrementing.

mask_type operator!() const noexcept;

Returns: A simd_mask object with the \( i \)th element set to \( \text{operator[]} (i) \) for all \( i \) in the range of \([0, \text{size()}]\).

simd operator()() const noexcept;

Returns: A simd object where each bit is the inverse of the corresponding bit in *this.
Remarks: This operator shall not participate in overload resolution unless \( T \) is an integral type.

simd operator()() const noexcept;

Returns: *this.

simd operator()() const noexcept;

Returns: A simd object where the \( i \)th element is initialized to \( \text{operator[]} (i) \) for all \( i \) in the range of \([0, \text{size()}]\).

(3.1.7) 26.9.7 simd non-member operations

(3.1.7.1) 26.9.7.1 simd binary operators

friend simd operator+(const simd & lhs, const simd & rhs) noexcept;
friend simd operator-(const simd & lhs, const simd & rhs) noexcept;
friend simd operator*(const simd & lhs, const simd & rhs) noexcept;
friend simd operator/(const simd & lhs, const simd & rhs) noexcept;
friend simd operator%(const simd & lhs, const simd & rhs) noexcept;
friend simd operator&(const simd & lhs, const simd & rhs) noexcept;
friend simd operator|(const simd & lhs, const simd & rhs) noexcept;
friend simd operator^(const simd & lhs, const simd & rhs) noexcept;
friend simd operator<<(const simd & lhs, const simd & rhs) noexcept;
friend simd operator>>(const simd & lhs, const simd & rhs) noexcept;

Returns: A simd object initialized with the results of applying the indicated operator to \( \text{lhs} \) and \( \text{rhs} \) as a binary element-wise operation.
Remarks: Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type value_type.
friend simd operator<<(const simd & v, int n) noexcept;
friend simd operator>>(const simd & v, int n) noexcept;

3 Returns: A simd object where the \(i\)th element is initialized to the result of applying the indicated operator to \(v[i]\) and \(n\) for all \(i\) in the range of \([0, \text{size()}]\).

4 Remarks: These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type value_type.

### 26.9.7.2 simd compound assignment

friend simd & operator+=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator-=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator*=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator/=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator%=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator&=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator|=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator^=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator<<=( simd & lhs, const simd & rhs) noexcept;
friend simd & operator>>=( simd & lhs, const simd & rhs) noexcept;

1 Effects: These operators apply the indicated operator to \(\text{lhs}\) and \(\text{rhs}\) as an element-wise operation.

2 Returns: \(\text{lhs}\).

3 Remarks: These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type value_type.

friend simd & operator<<=( simd & lhs, int n) noexcept;
friend simd & operator>>=( simd & lhs, int n) noexcept;

4 Effects: Equivalent to: return operator@=(lhs, simd(n));

5 Remarks: These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type value_type.

### 26.9.7.3 simd compare operators

friend mask_type operator==(const simd & lhs, const simd & rhs) noexcept;
friend mask_type operator!=(const simd & lhs, const simd & rhs) noexcept;
friend mask_type operator>=(const simd & lhs, const simd & rhs) noexcept;
friend mask_type operator<=(const simd & lhs, const simd & rhs) noexcept;
friend mask_type operator>(const simd & lhs, const simd & rhs) noexcept;
friend mask_type operator<(const simd & lhs, const simd & rhs) noexcept;

1 Returns: A simd_mask object initialized with the results of applying the indicated operator to \(\text{lhs}\) and \(\text{rhs}\) as a binary element-wise operation.

### 26.9.7.4 Reductions

1 In this subclause, BinaryOperation shall be a binary element-wise operation.
template<class T, class Abi, class BinaryOperation = plus<>>
T reduce(const simd<T, Abi>& x, BinaryOperation binary_op = {});

Requires: binary_op shall be callable with two arguments of type T returning T, or callable with two arguments of type simd<T, A1> returning simd<T, A1> for every A1 that is an ABI tag type.

Returns: GENERALIZED_SUM(binary_op, x.data[i], ...) for all i in the range of [0, size())

Throws: Any exception thrown from binary_op.

template<class M, class V, class BinaryOperation >
typename V::value_type reduce(const const_where_expression<M, V>& x, typename V::value_type identity_element, BinaryOperation binary_op = {});

Requires: binary_op shall be callable with two arguments of type T returning T, or callable with two arguments of type simd<T, A1> returning simd<T, A1> for every A1 that is an ABI tag type. The results of binary_op(identity_element, x) and binary_op(x, identity_element) shall be equal to x for all finite values x representable by V::value_type.

Returns: If none_of(x.mask), returns identity_element. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all selected indices i.

Throws: Any exception thrown from binary_op.

template<class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, plus<> binary_op ) noexcept;

Returns: If none_of(x.mask), returns 0. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all selected indices i.

template<class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, multiplies<> binary_op ) noexcept;

Returns: If none_of(x.mask), returns 1. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all selected indices i.

template<class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, bit_and<> binary_op ) noexcept;

Requires: is_integral_v<V::value_type> is true.

Returns: If none_of(x.mask), returns ~V::value_type(). Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all selected indices i.

12

13

12

Requires: is_integral_v<V::value_type> is true.

Returns: If none_of(x.mask), returns 0. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all selected indices i.
template<class T, class Abi> T hmin(const simd<T, Abi>& x) noexcept;

14  Returns: The value of an element \( x[j] \) for which \( x[j] \leq x[i] \) for all \( i \) in the range of \([0, \text{size}())\).

template<class M, class V> typename V::value_type hmin(const const_where_expression<M, V>& x) noexcept;

15  Returns: If \( \text{none_of}(x\text{.mask}) \), the return value is \( \text{numeric_limits}<V\text{::value_type}>::\text{max}() \). Otherwise, returns the value of an element \( x\text{.data}[j] \) for which \( x\text{.mask}[j] == \text{true} \) and \( x\text{.data}[j] \leq x\text{.data}[i] \) for all selected indices \( i \).

template<class T, class Abi> T hmax(const simd<T, Abi>& x) noexcept;

16  Returns: The value of an element \( x[j] \) for which \( x[j] \geq x[i] \) for all \( i \) in the range of \([0, \text{size}())\).

template<class M, class V> typename V::value_type hmax(const const_where_expression<M, V>& x) noexcept;

17  Returns: If \( \text{none_of}(x\text{.mask}) \), the return value is \( \text{numeric_limits}<V\text{::value_type}>::\text{lowest}() \). Otherwise, returns the value of an element \( x\text{.data}[j] \) for which \( x\text{.mask}[j] == \text{true} \) and \( x\text{.data}[j] \geq x\text{.data}[i] \) for all selected indices \( i \).

(3.1.1.7.5) 26.9.7.5 Casts

[simd.casts]

template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>& x) noexcept;

1  Let \( T \) denote \( T\text{::value_type} \) if \( \text{is_simd_v}\langle T \rangle \) is \text{true}, or \( T \) otherwise.

2  Returns: A simd object with the \( i \)th element initialized to \( \text{static_cast}<T\text{::value_type}>(x[i]) \) for all \( i \) in the range of \([0, \text{size}())\).

3  Remarks: The function shall not participate in overload resolution unless

- every possible value of type \( U \) can be represented with type \( T \), and
- either
  - \( \text{is_simd_v}\langle T \rangle \) is \text{false}, or
  - \( T\text{::size()} == \text{simd<U, Abi>::size()} \) is \text{true}.

4  The return type is

- \( T \) if \( \text{is_simd_v}\langle T \rangle \) is \text{true};
- otherwise, simd<T, Abi> if \( U \) is the same type as \( T \);
- otherwise, simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>

template<class T, class U, class Abi> see below static_simd_cast(const simd<U, Abi>& x) noexcept;

5  Let \( T \) denote \( T\text{::value_type} \) if \( \text{is_simd_v}\langle T \rangle \) is \text{true} or \( T \) otherwise.

6  Returns: A simd object with the \( i \)th element initialized to \( \text{static_cast}<T\text{::value_type}>(x[i]) \) for all \( i \) in the range of \([0, \text{size}())\).

7  Remarks: The function shall not participate in overload resolution unless either

- \( \text{is_simd_v}\langle T \rangle \) is \text{false}, or
T::size() == simd<U, Abi>::size() is true.

The return type is
- T if is_simd_v<T> is true;
- otherwise, simd<T, Abi> if either U is the same type as T or make_signed_t<U> is the same type as make_signed_t<T>;
- otherwise, simd<T, simd_abi::fixed_size< simd<U, Abi>::size()>>.

template<class T, class U, class Abi> see below sind_cast(const simd_mask<U, Abi>& x) noexcept;
template<class T, class U, class Abi> see below static_sind_cast(const simd_mask<U, Abi>& x) noexcept;

Returns: A simd_mask object with the \(i\)th element initialized to \(x[i]\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: The functions shall not participate in overload resolution unless either
- is_simd_mask_v<T> is false, or
- T::size() == simd_size_v<U, Abi> is true.

The return type is
- T if is_simd_mask_v<T> is true;
- otherwise, simd_mask<T, Abi> if either U is the same type as T or make_signed_t<U> is the same type as make_signed_t<T>;
- otherwise, simd_mask<T, simd_abi::fixed_size<simd_size_v<U, Abi>>>.

fixed_size_sind<T, simd_size_v<T, Abi>> to_fixed_size(const simd<T, Abi>& x) noexcept;
template<class T, class Abi>
fixed_size_sind_mask<T, simd_size_v<T, Abi>> to_fixed_size(const simd_mask<T, Abi>& x) noexcept;

Returns: A data-parallel object with the \(i\)th element initialized to \(x[i]\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: These functions shall not participate in overload resolution unless simd_size_v<T, simd_abi::native<T>> == N is true.

template<class T, int N> native_simd<T> to_compatible(const fixed_size_simd<T, N>& x) noexcept;
template<class T, int N> native_simd_mask<T> to_compatible(const fixed_size_simd_mask<T, N>& x) noexcept;

Returns: A data-parallel object with the \(i\)th element initialized to \(x[i]\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: These functions shall not participate in overload resolution unless simd_size_v<T, simd_abi::compatible<T>> == N is true.

tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...,>
split(const simd<T, Abi>& x) noexcept;
template< size_t ... Sizes , class T, class Abi>
tuple<simd_mask<T, simd_abi::deduce_t<T, Sizes>>...,>
split(const simd_mask<T, Abi>& x) noexcept;
Returns: A tuple of data-parallel objects with the \(i\)th simd/simd_mask element of the \(j\)th tuple element initialized to the value of the element \(x\) with index \(i + \text{sum of the first } j\) values in the Sizes pack.

Remarks: These functions shall not participate in overload resolution unless the sum of all values in the Sizes pack is equal to \(\text{simd_size_v<T, Abi>}\).

\[
\text{template<class V, class Abi>}
\]
\[
\text{array}<V, \text{simd_size_v<typename V::value_type, Abi}> / V::size()>
\text{split(const simd<typename V::value_type, Abi>& x)} \text{noexcept;}
\]

Returns: An array of data-parallel objects with the \(i\)th simd/simd_mask element of the \(j\)th array element initialized to the value of the element in \(x\) with index \(i \times V::size()\).

Remarks: These functions shall not participate in overload resolution unless either:

- \(\text{is_simd_v<V>}\) is true and \(\text{simd_size_v<typename V::value_type, Abi}>\) is an integral multiple of \(V::size()\), or
- \(\text{is_simd_mask_v<V>}\) is true and \(\text{simd_size_v<typename V::simd_type::value_type, Abi}>\) is an integral multiple of \(V::size()\).

\[
\text{template< size_t N, class T, class A>}
\]
\[
\text{array}<\text{resize_simd}<\text{simd_size_v<T, A> / N}, \text{simd}<T, A>> N>
\text{split_by(const simd<T, A>& x)} \text{noexcept;}
\]

Returns: An array \(\text{arr}\) where \(\text{arr}[i][j]\) is initialized by \(x[i \times (\text{simd_size_v<T, A> / N}) + j]\).

Remarks: The functions shall not participate in overload resolution unless \(\text{simd_size_v<T, A}>\) is an integral multiple of \(N\).

\[
\text{template<class T, class... Abis>}
\]
\[
\text{simd<T, simd abi::deduce_t<T, (\text{simd_size_v<T, Abis>} + ...)>> concat(}
\text{const simd<T, Abis>\&... xs)} \text{noexcept;}
\]

Returns: A data-parallel object initialized with the concatenated values in the xs pack of data-parallel objects: The \(i\)th simd/simd_mask element of the \(j\)th parameter in the xs pack is copied to the return value’s element with index \(i + \text{the sum of the width of the first } j\) parameters in the xs pack.

\[
\text{template< class T, class Abi, size_t N>}
\]
\[
\text{resize_simd<\text{simd_size_v<T, Abi> * N}, \text{simd<T, Abi}>}}
\text{concat(const array<simd<T, Abi>, N\& arr)} \text{noexcept;}
\]

Returns: A data-parallel object, the \(i\)th element of which is initialized by \(\text{arr}[i / \text{simd_size_v<T, Abi>}][(i \% \text{simd_size_v<T, Abi>}].\)
(3.1.1.7.6) 26.9.7.6 Algorithms

```
template<class T, class Abi> simd<T, Abi> min(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```

**Returns:** The result of the element-wise application of \( \text{std::min}(a[i], b[i]) \) for all \( i \) in the range of \([0, \text{size()}]\).

```
template<class T, class Abi> simd<T, Abi> max(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```

**Returns:** The result of the element-wise application of \( \text{std::max}(a[i], b[i]) \) for all \( i \) in the range of \([0, \text{size()}]\).

```
template<class T, class Abi> pair<simd<T, Abi>, simd<T, Abi>> minmax(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```

**Returns:** A pair initialized with

- the result of element-wise application of \( \text{std::min}(a[i], b[i]) \) for all \( i \) in the range of \([0, \text{size()}]\) in the first member, and
- the result of element-wise application of \( \text{std::max}(a[i], b[i]) \) for all \( i \) in the range of \([0, \text{size()}]\) in the second member.

```
template<class T, class Abi> simd<T, Abi> clamp(const simd<T, Abi>& v, const simd<T, Abi>& lo, const simd<T, Abi>& hi);
```

**Requires:** No element in \( lo \) shall be greater than the corresponding element in \( hi \).

**Returns:** The result of element-wise application of \( \text{std::clamp}(v[i], lo[i], hi[i]) \) for all \( i \) in the range of \([0, \text{size()}]\).

(3.1.1.7.7) 26.9.7.7 Math library

For each set of overloaded functions within `<cmath>`, there shall be additional overloads sufficient to ensure that if any argument corresponding to a `double` parameter has type `simd<T, Abi>`, where `is_floating_point_v<T>` is true, then:

- All arguments corresponding to `double` parameters shall be convertible to `simd<T, Abi>`.
- All arguments corresponding to `double*` parameters shall be of type `simd<T, Abi>*`.
- All arguments corresponding to parameters of integral type `U` shall be convertible to `fixed_size_simd<U, simd_size_v<T, Abi>>`.
- All arguments corresponding to `U*`, where `U` is integral, shall be of type `fixed_size_simd<U, simd_size_v<T, Abi>>*`.
- If the corresponding return type is `double`, the return type of the additional overloads is `simd<T, Abi>`. Otherwise, if the corresponding return type is `bool`, the return type of the additional overload is `simd_mask<T, Abi>`. Otherwise, the return type is `fixed_size_simd<R, simd_size_v<T, Abi>>`, with `R` denoting the corresponding return type.
It is unspecified whether a call to these overloads with arguments that are all convertible to \( \text{simd}<T, \text{Abi}> \) but are not of type \( \text{simd}<T, \text{Abi}> \) is well-formed.

Each function overload produced by the above rules applies the indicated \(<\text{cmath}>\) function element-wise. For the mathematical functions, the results per element only need to be approximately equal to the application of the function which is overloaded for the element type.

The behavior is undefined if a domain, pole, or range error occurs when the input argument(s) are applied to the indicated \(<\text{cmath}>\) function.

If \( \text{abs} \) is called with an argument of type \( \text{simd}<X, \text{Abi}> \) for which \( \text{is_unsigned_v}<X> \) is true, the program is ill-formed.
The class template `simd_mask` is a data-parallel type with the element type `bool`. The width of a given `simd_mask` specialization is a constant expression, determined by the template parameters. Specifically, `simd_mask<T, Abi>::size() == simd<T, Abi>::size()`.

Every specialization of `simd_mask` shall be a complete type. The specialization `simd_mask<T, Abi>` is supported if T is a vectorizable type and

- Abi is `simd_abi::scalar`, or
- Abi is `simd_abi::fixed_size<N>`, with N constrained as defined in (26.9.3).

If Abi is an extended ABI tag, it is implementation-defined whether `simd_mask<T, Abi>` is supported. Remark: The intent is for implementations to decide on the basis of the currently targeted system.

If `simd_mask<T, Abi>` is not supported, the specialization shall have a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. Otherwise, the following are true:

- `is_nothrow_move_constructible_v<simd_mask<T, Abi>>`, and
- `is_nothrow_move_assignable_v<simd_mask<T, Abi>>`, and
- `is_nothrow_default_constructible_v<simd_mask<T, Abi>>`.

Default initialization performs no initialization of the elements; value-initialization initializes each element with `false`. Remark: Thus, default initialization leaves the elements in an indeterminate state.

Implementations should enable explicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class `simd_mask`:

```cpp
explicit operator implementation-defined() const;
explicit simd_mask(const implementation-defined& init) const;
```

The member type `reference` has the same interface as `simd<T, Abi>::reference`, except its value_type is `bool`. (3)

(3.1.1.8.2) 26.9.8.2 `simd_mask` width

```cpp
static constexpr size_t size() noexcept;
```

Returns: The width of `simd<T, Abi>`.

(3.1.1.8.3) 26.9.8.3 Constructors

```cpp
explicit simd_mask(value_type x) noexcept;
```

Effects: Constructs an object with each element initialized to x.

```cpp
template<class U> simd_mask(const simd_mask<U, simd_abi::fixed_size<size()>>& u, x) noexcept;
```

Effects: Constructs an object of type `simd_mask` where the i-th element equals `x[i]` for all i in the range of `[0, size())`.

Remarks: This constructor shall not participate in overload resolution unless `abi_type` is `simd_abi::fixed_size<size()>`.

28
template<class G> simd_mask(G&& gen) noexcept;

Effects: Constructs an object where the $i$\textsuperscript{th} element is initialized to \texttt{gen\{}\texttt{integral\_constant</size\_t, i>{}\}}.

Remarks: This constructor shall not participate in overload resolution unless \texttt{static\_cast<\texttt{bool}\>(\texttt{gen\{}\texttt{integral\_constant</size\_t, i>{}\})\}} is well-formed for all $i$ in the range of \(0, \text{size()}\).

The calls to \texttt{gen} are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by \texttt{gen(??)}.

template<class Flags> simd_mask(const value_type* mem, Flags);

Requires:
- \([\text{mem, mem + size()}\] is a valid range.
- If the template parameter \texttt{Flags} is \texttt{vector\_aligned\_tag}, \texttt{mem} shall point to storage aligned by \texttt{memory\_alignment\_v<simd\_mask>}.
- If the template parameter \texttt{Flags} is \texttt{overaligned\_tag<N>}, \texttt{mem} shall point to storage aligned by \texttt{N}.
- If the template parameter \texttt{Flags} is \texttt{element\_aligned\_tag}, \texttt{mem} shall point to storage aligned by \texttt{alignof(value\_type)}.

Effects: Constructs an object where the $i$\textsuperscript{th} element is initialized to \texttt{mem[i]} for all $i$ in the range of \(0, \text{size()}\).

Throws: Nothing.

Remarks: This constructor shall not participate in overload resolution unless \texttt{is\_simd\_flag\_type\_v<\texttt{Flags}>\}} is true.

(3.1.8.4) 26.9.8.4 Copy functions [simd.mask.copy]

template<class Flags> void copy_from(const value_type* mem, Flags);

Requires:
- \([\text{mem, mem + size()}\] is a valid range.
- If the template parameter \texttt{Flags} is \texttt{vector\_aligned\_tag}, \texttt{mem} shall point to storage aligned by \texttt{memory\_alignment\_v<simd\_mask>}.
- If the template parameter \texttt{Flags} is \texttt{overaligned\_tag<N>}, \texttt{mem} shall point to storage aligned by \texttt{N}.
- If the template parameter \texttt{Flags} is \texttt{element\_aligned\_tag}, \texttt{mem} shall point to storage aligned by \texttt{alignof(value\_type)}.

Effects: Replaces the elements of the \texttt{ simd\_mask} object such that the $i$\textsuperscript{th} element is replaced with \texttt{mem[i]} for all $i$ in the range of \(0, \text{size()}\).

Throws: Nothing.

Remarks: This function shall not participate in overload resolution unless \texttt{is\_simd\_flag\_type\_v<\texttt{Flags}>\} is true.

template<class Flags> void copy_to(value_type* mem, Flags);
Requires:

- \([\text{mem}, \text{mem} + \text{size()})\) is a valid range.
- If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<simd_mask>.
- If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.
- If the template parameter Flags is element_aligned_tag, mem shall point to storage aligned by alignof(value_type).

Effects: Copies all simd_mask elements as if \(\text{mem}[i] = \text{operator}[][i]\) for all \(i\) in the range of \([0, \text{size()})\).

Throws: Nothing.

Remarks: This function shall not participate in overload resolution unless is_simd_flag_type_v<Flags> is true.

(3.1.1.8.5) 26.9.8.5 Subscript operators

reference operator[](size_t i);

Requires: \(i < \text{size()}.\)

Returns: A reference (see 3) referring to the \(i^{th}\) element.

Throws: Nothing.

value_type operator[](size_t i) const;

Requires: \(i < \text{size()}.\)

Returns: The value of the \(i^{th}\) element.

Throws: Nothing.

(3.1.1.8.6) 26.9.8.6 Unary operators

simd_mask operator!() const noexcept;

Returns: The result of the element-wise application of operator!.

(3.1.1.9) 26.9.9 Non-member operations

(3.1.1.9.1) 26.9.9.1 Binary operators

friend simd_mask operator&& (const simd_mask & lhs, const simd_mask & rhs) noexcept;
friend simd_mask operator|| (const simd_mask & lhs, const simd_mask & rhs) noexcept;
friend simd_mask operator& (const simd_mask & lhs, const simd_mask & rhs) noexcept;
friend simd_mask operator| (const simd_mask & lhs, const simd_mask & rhs) noexcept;
friend simd_mask operator^ (const simd_mask & lhs, const simd_mask & rhs) noexcept;
Returns: A simd_mask object initialized with the results of applying the indicated operator to lhs and rhs as a binary element-wise operation.

(3.1.1.9.2) 26.9.9.2 Compound assignment

friend simd_mask& operator&=(simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask& operator|=(simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask& operator^=(simd_mask& lhs, const simd_mask& rhs) noexcept;

Effects: These operators apply the indicated operator to lhs and rhs as a binary element-wise operation.

Returns: lhs.

(3.1.1.9.3) 26.9.9.3 Comparisons

friend simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
friend simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;

Returns: A simd_mask object initialized with the results of applying the indicated operator to lhs and rhs as a binary element-wise operation.

(3.1.1.9.4) 26.9.9.4 Reductions

template<class T, class Abi> bool all_of(const simd_mask<T, Abi>& k) noexcept;

Returns: true if all boolean elements in k are true, false otherwise.

template<class T, class Abi> bool any_of(const simd_mask<T, Abi>& k) noexcept;

Returns: true if at least one boolean element in k is true, false otherwise.

template<class T, class Abi> bool none_of(const simd_mask<T, Abi>& k) noexcept;

Returns: true if none of the one boolean elements in k is true, false otherwise.

template<class T, class Abi> bool some_of(const simd_mask<T, Abi>& k) noexcept;

Returns: true if at least one of the one boolean elements in k is true and at least one of the boolean elements in k is false, false otherwise.

template<class T, class Abi> int popcount(const simd_mask<T, Abi>& k) noexcept;

Returns: The number of boolean elements in k that are true.

template<class T, class Abi> int find_first_set(const simd_mask<T, Abi>& k);
Requires: any_of(k) returns true.

Returns: The lowest element index \( i \) where \( k[i] \) is true.

Throws: Nothing.

template<class T, class Abi> int find_last_set(const simd_mask<T, Abi>& k);

Requires: any_of(k) returns true.

Returns: The greatest element index \( i \) where \( k[i] \) is true.

Throws: Nothing.

bool all_of(T) noexcept;
bool any_of(T) noexcept;
bool none_of(T) noexcept;
bool some_of(T) noexcept;
int popcount(T) noexcept;

Returns: all_of and any_of return their arguments; none_of returns the negation of its argument; some_of returns false; popcount returns the integral representation of its argument.

Remarks: The parameter type \( T \) is an unspecified type that is only constructible via implicit conversion from bool.

int find_first_set(T);
int find_last_set(T);

Requires: The value of the argument is true.

Returns: 0.

Throws: Nothing.

Remarks: The parameter type \( T \) is an unspecified type that is only constructible via implicit conversion from bool.

\[(3.1.9.5)\] 26.9.9.5 where functions

\[\text{[simd.mask.where]}\]

template<class T, class Abi>
where_expression<simd_mask<T, Abi>, simd<T, Abi>>
where(const typename simd<T, Abi>::mask_type& k, simd<T, Abi>& v) noexcept;
template<class T, class Abi>
const_where_expression<simd_mask<T, Abi>, simd<T, Abi>>
where(const typename simd<T, Abi>::mask_type& k, const simd<T, Abi>& v) noexcept;
template<class T, class Abi>
where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
where(const type_identity_t<simd_mask<T, Abi>>& k, simd_mask<T, Abi>& v) noexcept;
template<class T, class Abi>
const_where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
where(const type_identity_t<simd_mask<T, Abi>>& k, const simd_mask<T, Abi>& v) noexcept;

Returns: An object (26.9.5) with mask and data initialized with \( k \) and \( v \) respectively.
template<class T>
where_expression<bool T>
  where(see below k, % v) noexcept;
template<class T>
const_where_expression<bool, T>
  where(see below k, const % v) noexcept;

2 Remarks: The functions shall not participate in overload resolution unless
  • T is neither a simd nor a simd_mask specialization, and
  • the first argument is of type bool.

3 Returns: An object (26.9.5) with mask and data initialized with k and v respectively.

A BIBLIOGRAPHY


