Document Number: P1928R5
Date: 2023-06-17
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG
Target: C++26

std::simd — 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. Now there is sufficient feedback to improve and merge Section 9 of the Parallelism TS 2 into the IS working draft.

CONTENTS

1 CHANGELOG

1.1 changes from revision 0

Previous revision: P1928R0

- Target C++26, addressing SG1 and LEWG.
- Call for a merge of the (improved & adjusted) TS specification to the IS.
- Discuss changes to the ABI tags as consequence of TS experience; calls for polls to change the status quo.
- Add template parameter `T` to `simd_abi::fixed_size`.
- Remove `simd_abi::compatible`.
- Add (but ask for removal) `simd_abi::abi_stable`.
- Mention TS implementation in GCC releases.
- Add more references to related papers.
- Adjust the clause number for [numbers] to latest draft.
- Add open question: what is the correct clause for [simd]?
- Add open question: integration with ranges.
- Add `simd_mask` generator constructor.
- Consistently add simd and simd_mask to headings.
- Remove experimental and parallelism_v2 namespaces.
- Present the wording twice: with and without diff against N4808 (Parallelism TS 2).
- Default load/store flags to `element_aligned`.
- Generalize casts: conditionally `explicit` converting constructors.
- Remove named cast functions.

1.2 changes from revision 1

Previous revision: P1928R1

- Add floating-point conversion rank to condition of `explicit` for converting constructors.
- Call out different or equal semantics of the new ABI tags.
- Update introductory paragraph of Section 4; R1 incorrectly kept the text from R0.
- Define simd::size as a `constexpr` static data-member of type `integral_constant<size_t, N>`. This simplifies passing the size via function arguments and still be useable as a constant expression in the function body.
- Document addition of `constexpr` to the API.
- Add `constexpr` to the wording.
- Removed ABI tag for passing `simd` over ABI boundaries.
- Apply cast interface changes to the wording.
- Explain the plan: what this paper wants to merge vs. subsequent papers for additional features. With an aim of minimal removal/changes of wording after this paper.
- Document rationale and design intent for `where` replacement.

1.3 changes from revision 2

Previous revision: P1928R2

- Propose alternative to `hmin` and `hmax`.
- Discuss `simd_mask` reductions wrt. consistency with `<bit>`. Propose better names to avoid ambiguity.
- Remove `some_of`.
- Add unary ~ to `simd_mask`.
- Discuss and ask for confirmation of masked “overloads” names and argument order.
- Resolve inconsistencies wrt. `int` and `size_t`: Change `fixed_size` and `resize_simd` NTTPs from `int` to `size_t` (for consistency).
- Discuss conversions on loads and stores. (Section **??**)

- Point to [P2509R0] as related paper.
- Generalize load and store from pointer to `contiguous_iterator`. (Section 4.5)
- Moved "`element_reference` is overspecified" to "Open questions".

1.4 changes from revision 3

Previous revision: P1928R3

- Remove wording diff.
- Add std::simd to the paper title.
- Update ranges integration discussion and mention formatting support via ranges (Section 5.7).
- Fix: pass iterators by value not const-ref.
- Add lvalue-ref qualifier to subscript operators (Section 4.11).
- Constrain `simd` operators: require operator to be well-formed on objects of `value_type` (28.9.6.7, 28.9.7.1).
- Rename mask reductions as decided in Issaquah.
- Remove R3 ABI discussion and add follow-up question (Section 5.2).
- Add open question on first template parameter of `simd_mask` (Section 5.3).
- Overload loads and stores with mask argument (28.9.6.4, 28.9.6.5, 28.9.8.3, 28.9.8.4).
- Respecify `simd` reductions to use a `simd_mask` argument instead of `const_where_expression` (28.9.7.5).
- Add `simd_mask` operators returning a `simd` (28.9.8.6, 28.9.8.7)
- Add conditional operator overloads as hidden friends to `simd` and `simd_mask` (28.9.7.4, 28.9.9.4).
- Discuss `std::hash` for `simd` (Section 5.8).
- Constrain some functions (e.g., min, max, clamp) to be `totally_ordered` (28.9.7.5, 28.9.7.7).
- Asking for reconsideration of conversion rules (Section 5.4).
- Rename load/store flags (Section 4.14).
- Extend load/store flags with a new flag for conversions on load/store. (Section 4.14).

- Update `hmin/hmax` discussion with more extensive naming discussion (Section 5.1).
- Discuss freestanding `simd` (Section 5.9).
- Discuss `split` and `concat` (Section 5.5).
- Apply the new library specification style from P0788R3.

1.5 Changes from revision 4

Previous revision: P1928R4

- Added `simd_select` discussion (Section 4.8.1).

2 Straw Polls

2.1 SG1 at Kona 2022

Poll: After significant experience with the TS, we recommend that the next version (the TS version with improvements) of `std::simd` target the IS (C++26)

SF	F	N	A	SA
10	8	0	0	0

Poll: We like all of the recommended changes to `std::simd` proposed in p1928r1 (Includes making all of `std::simd` `constexpr`, and dropping an ABI stable type)
→ unanimous consent

Poll: Future papers and future revisions of existing papers that target `std::simd` should go directly to LEWG. (We do not believe there are SG1 issues with `std::simd` today.)

SF	F	N	A	SA
9	8	0	0	0

2.2 LEWG at Issaquah 2023

Poll: Change the default SIMD ABI tag to simd_abi::native instead of simd_abi::compatible.

SF	F	N	A	SA
16	12	0	0	1

Poll: Change simd_abi::fixed_size to not recommend implementations make it ABI compatible.

SF	F	N	A	SA
16	7	1	0	1

Poll: Make simd::size an integral_constant instead of a static member function.

SF	F	N	A	SA
9	8	7	1	0

Poll: simd masked operations should look like (vote for as many options as you'd like):

Option	Votes
where(u > 0, v).copy_from(ptr)	12
v.copy_from_if(u > 0, ptr)	1
v.copy_from_if(ptr, u > 0)	2
v.copy_from(ptr, u > 0)	14
v.copy_from(u > 0, ptr)	3
v.copy_from_where(u > 0, ptr)	4
v.copy_from_where(ptr, u > 0)	11

Poll: simd masked operations should look like (vote once for your favorite):

Option	Votes
where(u > 0, v).copy_from(ptr)	5
v.copy_from(ptr, u > 0)	12
v.copy_from_where(ptr, u > 0)	6

Poll: Make copy_to, copy_from, and the load constructor only do value-preserving conversions by default and require passing a flag to do non-value-preserving conversions.

SF	F	N	A	SA
14	9	1	0	0

Poll: SIMD types and operations should be value preserving, even if that means they're inconsistent with the builtin numeric types.

SF	F	N	A	SA
3	10	6	3	0

Poll: 2 * simd<float> should produce simd<double> (status quo: simd<float>).

SF	F	N	A	SA
1	5	9	6	1

Poll: Put SIMD types and operations into std:: and add the simd_ prefix to SIMD specific things (such as split and vector_aligned).

SF	F	N	A	SA
4	5	4	9	2

Poll: Put SIMD types and operations into a nested namespace in std::.

SF	F	N	A	SA
4	7	0	5	9

Poll: simd should be a range.

SF	F	N	A	SA
4	9	5	4	4

Poll: There should be an explicit way to get a view to a simd.

SF	F	N	A	SA
8	12	3	3	0

Poll: simd should have explicitly named functions for horizontal minimum and horizontal maximum.

SF	F	N	A	SA
4	5	7	4	2

Poll: Rename all_of/ any_of/none_of to reduce_all_of/reduce_any_of/reduce_none_of.

SF	F	N	A	SA
2	1	1	8	5

Poll: Rename all_of/any_of/none_of to reduce_and/reduce_or/reduce_nand.

SF	F	N	A	SA
2	6	2	4	3

Poll: Rename popcount to reduce_count.

SF	F	N	A	SA
4	9	2	1	2

Poll: Rename find_first_set/find_last_set to reduce_min_index/reduce_max_index.

SF	F	N	A	SA
2	7	3	2	3

3 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 release, a complete implementation of Section 9 of the TS was made available. Since GCC 11 a complete `simd` implementation of the TS is part of its standard library.

In the meantime the TS feedback progressed to a point where a merge should happen ASAP. This paper proposes to merge only the feature-set that is present in the Parallelism TS 2. (Note: The first revision of this paper did not propose a merge.) If, due to feedback, any of these features require a change, then this paper (P1928) is the intended vehicle. If a new feature is basically an addition to the wording proposed here, then it will progress in its own paper.

3.1 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.

P1068 R3 of the paper removed discussion/proposal of a `simd` based API because it was targeting C++23 with the understanding of `simd` not being ready for C++23. This is unfortunate as the presence of `simd` in the IS might lead to a considerably different assessment of the iterator/range-based API proposed in P1068.

P0917 The ability to write code that is generic wrt. arithmetic types and `simd` types is considered to be of high value (TS feedback). Conditional expressions via the `where` function were not all too well received. Conditional expressions via the conditional operator would provide a solution deemed perfect by those giving feedback (myself included).

Draft on non-member operator[] TODO

P2600 The fix for ADL is important to ensure the above two papers do not break existing code.

P0543 The paper proposing functions for saturation arithmetic expects `simd` overloads as soon as `simd` is merged to the IS.

P0553 The bit operations that are part of C++20 expects `simd` overloads as soon as `simd` is merged to the IS.

P2638 Intel's response to P1915R0 for `std::simd`

P2663 `std::simd<std::complex<T>>`.

P2664 Permutations for `simd`.

P2509 D'Angelo [P2509R0] proposes a "type trait to detect conversions between arithmetic-like types that always preserve the numeric value of the source object". This matches the *value-preserving* conversions the `simd` specification uses.

The papers P0350, P0918, P2663, P2664, and the `simd`-based P1068 fork currently have no shipping vehicle and are basically blocked on this paper.

4 CHANGES AFTER TS FEEDBACK

[P1915R0] (Expected Feedback from `simd` in the Parallelism TS 2) was published in 2019, asking for feedback to the TS. I received feedback on the TS via the GitHub issue tracker, e-mails, and personal conversations. There is also a lot of valuable feedback published in P2638 "Intel's response to P1915R0 for `std::simd`".

4.1 IMPROVE ABI TAGS

Summary:

- Change the default SIMD ABI tag to `simd_abi::native<T>` instead of `simd_abi::compatible<T>`.
- Change `simd_abi::fixed_size` to not recommend implementations make it ABI compatible.

For a discussion, see P1928R3 Section 4.1.
Follow-up open questions are discussed in Section 5.2 and Section 5.3.

4.2 SIMPLIFY/GENERALIZE CASTS

For a discussion, see P1928R3 Section 4.2.
Summary of changes wrt. TS:

1. `simd<T0, A0>` is convertible to `simd<T1, A1>` if `simd_size_v<T0, A0> == simd_size_v<T1, A1>`.
2. `simd<T0, A0>` is implicitly convertible to `simd<T1, A1>` if, additionally,
   - the conversion `T0` to `T1` is value-preserving, and
   - if both `T0` and `T1` are integral types, the integer conversion rank of `T1` is greater than or equal to the integer conversion rank of `T0`, and
   - if both `T0` and `T1` are floating-point types, the floating-point conversion rank of `T1` is greater than or equal to the floating-point conversion rank of `T0`.
3. `simd_mask<T0, A0>` is convertible to `simd_mask<T1, A1>` if `simd_size_v<T0, A0> == simd_size_v<T1, A1>`.
4. `simd_mask<T0, A0>` is implicitly convertible to `simd_mask<T1, A1>` if, additionally, `sizeof(T0) == sizeof(T1)`. (This point is irrelevant if Section 5.3 is accepted.)

5. `simd<T0, A0>` can be `bit_casted` to `simd<T1, A1>` if `sizeof(simd<T0, A0>) == sizeof(simd<T1, A1>)`.
6. `simd_mask<T0, A0>` can be `bit_casted` to `simd_mask<T1, A1>` if `sizeof(simd_mask<T0, A0>) == sizeof(simd_mask<T1, A1>)`.

4.3 ADD SIMD_MASK GENERATOR CONSTRUCTOR

This constructor was added:

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

For a discussion, see P1928R3 Section 4.3.

4.4 DEFAULT LOAD/STORE FLAGS TO ELEMENT_ALIGNED

Different to the TS, load/store flags default to `element_aligned`. For a discussion, see P1928R3 Section 4.4.

4.5 CONTIGUOUS ITERATORS FOR LOADS AND STORES

Different to the TS, loads and stores use `contiguous_iterator` instead of pointers. For a discussion, see P1928R3 Section 4.5.

4.6 CONSTEXPR EVERYTHING

The merge adds `constexpr` to all functions. For a discussion, see P1928R3 Section 4.6.

4.7 SPECIFY SIMD::SIZE AS INTEGRAL_CONSTANT

Different to the TS, this paper uses a static data member `size` of type `std::integral_constant<std::size_t, N>` in `simd` and `simd_mask`. For a discussion, see P1928R3 Section 4.7.

4.8 REPLACE WHERE FACILITIES

The following load/store overloads have been added as a replacement for `std::experimental::where_expression::copy_from` and `std::experimental::const_where_expression::copy_to`:

- `simd::simd(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied from given range, otherwise use value-initialization)
- `simd::copy_from(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied from given range)

- `simd::copy_to(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied to given range)
- `simd_mask::simd_mask(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied from given range, otherwise use value-initialization)
- `simd_mask::copy_from(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied from given range)
- `simd_mask::copy_to(contiguous_iterator, const mask_type&, Flags = {})` (selected elements are copied to given range)

The `reduce`, `hmin`, and `hmax` overloads with `const_where_expression` argument have been replaced by overloads with `simd` and `simd_mask` arguments.

The following operators were added to `simd_mask`:

- `simd_mask::operator simd<U, A>() const noexcept`
- `simd_mask::operator+() const noexcept`
- `simd_mask::operator-() const noexcept`
- `simd_mask::operator~() const noexcept`
- `simd_mask::operator?:() const noexcept`
- `simd_mask operator?:(const simd_mask&, const simd_mask&, const simd_mask&) noexcept`
- `simd_mask operator?:(const simd_mask&, bool, bool) noexcept`
- `simd<nonpromoting_common_type_t<T0, T1> operator?:(const simd_mask&, const T0&, const T1&) noexcept`

The following operator was added to `simd`:

- `operator?:(const mask_type& mask, const simd& a, const simd& b) noexcept`

At this point, the use of `operator?:` obviously cannot work. As long as we don't have the language feature for overloading `?:` the following needs to happen (also reproduced at the beginning of the wording section):

1. Replace `operator?:` by `conditional_operator_impl`.
2. Add a `std::conditional_operator(cond, a, b)` CPO that tries the following:

- If `conditional_operator_impl(cond, a, b)` can be found via ADL calls `conditional_operator_impl(cond, a, b)` unqualified, otherwise
- if `cond ? a : b` is well-formed, return its result, otherwise
- if `std::common_type_t<decltype(a), decltype(b)>` exists, cast `a` and `b` to this common type and apply `?:`.

Obviously, the use of a function (or CPO) instead of overloading `operator?:` has a significant impact on generic code that would prefer to keep the semantics of `?:`, which doesn't evaluate an expression unless its result is actually needed. For a function, we cannot pass expressions but only their results. Relevant papers: [P0927R2], [D0917].

A list of alternative names for `std::conditional_operator`:

- `std::ternary(cond, a, b)`
- `std::inline_if(cond, a, b)`
- `std::iif(cond, a, b)`
- `std::blend(cond, a, b)`
- `std::select(cond, a, b)`
- `std::choose(cond, a, b)`

For a discussion, see P1928R3 Section 4.8 and 5.3.

4.8.1 simd_select instead of conditional_operator

LEWG feedback on Tuesday in Varna:

- Don't make it a CPO: user's shouldn't extend this.
- Make it "value based", i.e. don't bother about references for non-simd arguments.
- Call it `simd_select`.
- Come back ASAP.

Proposal:

```
namespace std
{
template <typename T, typename U>
  constexpr common_type_t<T, U>
  simd_select(bool c, T x, U y)
```

```
  { return c ? x : y; }

template <size_t N, typename Abi>
  constexpr auto
  simd_select(const basic_simd_mask<N, Abi>& k, const auto& x, const auto& y)
  -> decltype(simd-select-impl(k, x, y))
  { return simd-select-impl(k, x, y); }
}
```

Replace `operator?:` hidden friends with *`simd-select-impl`* hidden friends in `basic_simd` and `basic_simd_mask`.

Pro Continues to use the same conversion mechanisms as all the other (binary) operators (requires the use of hidden friends).

Con Harder to discover/understand/document.

Wording TBD.

4.9 Make use of int and size_t consistent

Different to the TS, this paper uses `size_t` for `std::experimental::simd_abi::fixed_size`, `std::experimental::fixed_size_simd`, `std::experimental::fixed_size_simd_mask`, `std::experimental::resize_simd`, and `std::experimental::simd_abi::max_fixed_size`. For a discussion, see P1928R3 Section 4.9.

4.10 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)`). At the very least, `std::abs(simd<`*`signed-integral`*`>)` should be specified.

Also, from implementation experience, "undefined behavior" for domain, pole, or range error is unnecessary. It could either be an unspecified result or even match the expected result of the function according to Annex F in the C standard. The latter could possibly be a recommendation, i.e. QoI. The intent is to avoid `errno` altogether, while still supporting floating-point exceptions (possibly depending on compiler flags).

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

4.11 Add lvalue-qualifier to non-const subscript

The `operator[]` overloads of `simd` and `simd_mask` returned a proxy reference object for non-`const` objects and the `value_type` for `const` objects. This made expressions such as `(x * 2)[0] = 1` well-formed. However, assignment to temporaries can only be an error in the code (or code obfuscation).

Both `operator[]` overloads should be lvalue-ref qualified to make `(x * 2)[0]` pick the const overload, which returns a prvalue that is not assignable.

4.12 Rename simd_mask reductions

Summary:

- The function `std::experimental::some_of` was removed.
- The function `std::experimental::popcount` was renamed to `std::reduce_count`.
- The function `std::experimental::find_first_set` was renamed to `std::reduce_min_index`.
- The function `std::experimental::find_last_set` was renamed to `std::reduce_max_index`.

For a discussion of this topic see P1928R3 Section 5.2.

4.13 Added constraints on operators and functions to match their underlying element types

Previously some operators (e.g., `operator<`) and functions which relied on some property of the element type (e.g., `min` relies on ordering) were unconstrained. Operations which were not permitted on individual elements were still available in the overload set for simd objects of those types. Constraints have been added where necessary to remove such operators and functions from the overload set where they aren't supported.

4.14 Rename alignment flags and extend load/store flags for opt-in to conversions

For some discussion, see P1928R3 Section 5.4.

In addition to the TS, the load/store flag mechanism is extended to enable combination of flags. The new flag enables conversions that are not *value-preserving* on loads and stores. Without the flag only *value-preserving* conversions are allowed.

4.14.1 Historical context

The existing flags

- `std::experimental::element_aligned`,
- `std::experimental::vector_aligned`, and
- `std::experimental::overaligned<N>`

had no need for combining since they are exclusive. Therefore, the TS included no such mechanism. We should include a flag combining mechanism in any case, in order to allow extensions without breaking implementations. For reference, the original proposal [**§6.2**, N4184] included a `streaming` flag and a flag to assist in emitting prefetch instructions. I implemented these flags in the Vc library more than 10 years ago. For sake of proving the viability without too much complexity, SG1 asked to reduce load store flags to what we finally shipped in the TS.

4.14.2 Renaming alignment flags

Now that the alignment flags move from `std::experimental::` to `std::`, we need to reconsider their names. See Table 1 for a list of names that were considered. On my choices:

`STD::LOADSTORE_DEFAULT` This is the default. There is no need to say anything about alignment. Alignment follows the assumptions of the rest of C++; there's nothing special. It is useful to have a name for the default. Some generic programming patterns need to decide on default vs. aligned depending on template parameters.

`STD::LOADSTORE_ALIGNED` This flags says that the memory pointed to by the iterator follows stricter alignment requirements. Specifically it follows `memory_alignment_v<T, U>` (`T` is the `simd<T>` element type and `U` is the array element type). We should consider renaming `memory_alignment(_v)` to `simd_alignment(_v)` or `simd_loadstore_alignment(_v)`.

`STD::LOADSTORE_OVERALIGNED<N>` This flags says that the memory pointed to by the iterator is aligned by `N` bytes.

`STD::LOADSTORE_CONVERT` This flag allows all conversions on load/store, in contrast to only *value-preserving* conversions. It doesn't enable *all* conversions, because value-preserving conversions are always enabled. But I cannot think of a more descriptive name that isn't at the same time also more confusing ("convert more", "convert anything").

Note that the wording also allows additional implementation-defined load and store flags.

The trait `is_simd_flag_type` has been removed because the flag parameter is now constrained via the `loadstore_flag` class template.

As a result, executing a not-value-preserving store on 16-Byte aligned memory now reads as:

TS

```
float *addr = ...;
void f(stdx::native_simd<double> x) {
  x.copy_to(addr, stdx::overaligned<16>);
}
```

P1928R5

```
float *addr = ...;
void f(std::simd<double> x) {
  x.copy_to(addr, std::loadstore_convert |
                  std::loadstore_overaligned<16>);
}
```

TS name	Ideas	Preference
`std::experimental::element_aligned`	`std::default_simd_flags`, `std::simd_element_aligned`, `std::simd_flag_element_aligned`, `std::simd_copy_element_aligned`, `std::loadstore_element_aligned`, `std::simd_flags::element_aligned` , `std::simd_unaligned`, `std::simd_flag_unaligned`, `std::simd_copy_unaligned`, `std::loadstore_unaligned`, `std::simd_flags::unaligned` , `std::simd_default`, `std::simd_flag_default`, `std::simd_copy_default`, `std::loadstore_default`, `std::simd_flags::default`	`std::loadstore_default`
`std::experimental::vector_aligned`	`std::simd_aligned`, `std::simd_flag_aligned`, `std::simd_copy_aligned`, `std::loadstore_aligned`, `std::simd_flags::aligned`	`std::loadstore_aligned`
`std::experimental::overaligned<N>`	`std::overaligned<N>`, `std::simd_overaligned<N>`, `std::simd_flag_overaligned<N>`, `std::simd_copy_overaligned<N>`, `std::loadstore_overaligned<N>`, `std::simd_flags::overaligned<N>`	`std::loadstore_overaligned<N>`
—	`std::simd_convert`, `std::simd_flag_convert`, `std::simd_copy_convert`, `std::loadstore_convert`, `std::simd_flags::convert` , `std::simd_cvt_any`, `std::simd_flag_cvt_any`, `std::simd_copy_cvt_any`, `std::loadstore_cvt_any`, `std::simd_flags::cvt_any` , `std::simd_any_cvt`, `std::simd_flag_any_cvt`, `std::simd_copy_any_cvt`, `std::loadstore_any_cvt`, `std::simd_flags::any_cvt`	`std::loadstore_convert`

Table 1: Name alternatives for alignment flags

5 OPEN QUESTIONS

5.1 RENAMING HMIN AND HMAX OR ELSE

The functions `hmin(simd)` and `hmax(simd)` are basically specializations of `reduce(simd)`. I received feedback asking for better names.

We could rename the TS functions to:

1. `reduce_min(simd)` and `reduce_max(simd)` (*my recommendation — unless convinced otherwise*)

   Pro Clearly states that a reduction operation happens (optimized implementation will use tree reduction). Also consistent with the `reduce_` prefix for `simd_mask` reductions.

   Con Generic name, but `simd`-specific (at this point).

2. `min_element(simd)` and `max_element(simd)`

   Pro Clearly states that it returns a single element, not a `simd`.

   Con Mismatching behavior to `std::ranges::min_element`, which returns an iterator.

3. `reduce_min_element(simd)` and `reduce_max_element(simd)`

   Pro Most explicit about performing a reduction and returning a single element.

   Con Rather verbose. Also the use of `min_element` in the name is again a mismatch with `std::ranges::min_element`.

4. `horizontal_min(simd)` and `horizontal_max(simd)`

   Pro States that a horizontal operation is applied.

   Con We don't use the term „horizontal" anywhere else.

5. `extract_min(simd)` and `extract_max(simd)`

   Pro The term "extract" states that a subset of elements (in this case a single element) will be extracted. This may or may not be a good matching name to the `extract` facility that we might add later.

   Con Inconsistent with `std::reduce(simd)` (maybe okay because slightly different: `std::reduce(simd)` returns an aggregate of all elements in the `simd`, whereas `extract_max` "searches" for a certain element and returns its value - though also typically implemented as a tree reduction).

Alternatively, we can remove `hmin` and `hmax`, with the goal of using different (more generic) facilities. With C++17, there was nothing equivalent to `std::plus<>` for minimum and maximum. Since the merge of Ranges (C++20), we have `std::ranges::min` and `std::ranges::max`. The `reduce(simd)`

specification requires the `binary_op` to be callable with two `simd` arguments, though (split initial argument in half, call `binary_op`, split again, call `binary_op`, ...until only a scalar is left). This doesn't work with `std::ranges::min` (and `max`) because it requires an lvalue reference as return type. If we added another `operator()` to `std::ranges::min`, then their use with `reduce(simd)` would be slightly inconsistent:

```
simd<unsigned> v = ...;
auto a = reduce(v, std::bit_and<>); // must type <>
auto b = reduce(v, std::ranges::min); // must *not* type <>
```

However, if `simd` will be an `input_range` (see Section 5.6) then the `std::ranges::min(ranges::input_range auto&&, ...)` overload matches and `std::ranges::min(simd)` works out of the box. We could then leave it up to QoI to recognize the opportunity for a SIMD implementation of the reduction.

5.1.1 Suggested polls

Poll: We want to do something about `hmin` and `hmax`; i.e. the TS status quo is not acceptable for the IS.

SF	F	N	A	SA

Poll: Rename to `reduce_min` and `reduce_max`.

SF	F	N	A	SA

Poll: Extend `std::ranges::min` and `max` to allow prvalue return types (and remove `hmin`/`hmax`).

SF	F	N	A	SA

Poll: Remove `hmin` and `hmax` expecting `simd` to become a range.

SF	F	N	A	SA

5.2 Simplify fixed_size

To understand the following discussion it is worth remembering that given an element type `T` and SIMD width `N`, the ABI tag is not necessarily distinct. For example, given AVX-512, `fixed_size_simd<float, 16>` could either be one `zmm` register or two `ymm` registers. Orthogonally, the `simd_mask` type could either be stored as a bit-mask or as a vector-mask. Thus, we already have four (somewhat)

reasonable choices for the ABI tag. If we consider different TUs compiled with different "`-m`" flags then the possible set of ABI tags for "`float, 16`" becomes even larger.

There are only three possible reasons why `simd_abi::fixed_size<T, N>` should not be an alias for `simd_abi::deduce_t<T, N>` (or rather a rename of `deduce_t`):

1. We want `N` in `fixed_size_simd<T, N>` to be deducible.
2. We want to support overloading via differently named ABI tags (i.e. if the user types a different name, then the actual type should be different).
3. We want the actual type name of `simd_abi::fixed_size<T, N>` to be recognizable as "fixed size" and not just hide behind some implementation-defined ABI tag.

In the TS, `fixed_size_simd<T, N>` is required to be deducible. However, after the changes we did to `fixed_size` and conversions, there is no good motivation for deducing `fixed_size_simd<T, N>` instead of deducing `simd<T, Abi>`. Let's just communicate that `simd<T, Abi>` is the one and only way to deduce arbitrary length `simd` types.

In the TS, a user could write the following overload set:

```
void f(stdx::native_simd<float>);

template <int N>
void f(stdx::fixed_size_simd<float, N>);
```

Besides `N` not being deducible anymore, if `N == stdx::native_simd<float>::size()`, then the overloads would clash, using the same argument type. However, again, after the changes we did to `fixed_size` and conversions, there is probably no good reason left for declaring such an overload set. Instead a user should write a single function template passing `simd<float, Abi>`, deducing `Abi` and potentially constraining the function using Concepts.

That leaves the question of diagnostics / debugging with regard to ABI tag names. This argument can also go the other way: By hiding the actual ABI tag used for implementing `fixed_size<T, N>` the user has a harder time understanding what is going on. So I don't think this argument has much weight.

5.2.1 SUGGESTED POLL

Consequently, I propose the following poll:
Poll: Remove `simd_abi::fixed_size`, rename `simd_abi::deduce_t` to `simd_abi::fixed_size`, and remove (no public API) `simd_abi::deduce`. Require `fixed_size_simd<T, simd_size_v<T>>` to be the same type as `simd<T>`. Require `fixed_size_simd<T, 1>` to be the same type as `simd<T, simd_abi::scalar>`.

SF	F	N	A	SA

5.3 basic mask type?

Now that `simd_mask` types have become interconvertible, the distinction between `simd_mask<T, Abi>` and `simd_mask<U, Abi>` with `sizeof(T) == sizeof(U)` is gone. Actually, the fact that the type is interconvertible makes it harder to use (because it is easier to construct ambiguities).

Solution: *Make `simd_mask<T, Abi>` an alias for `basic_simd_mask<sizeof(T), Abi>`.*

Pros
- reduces the number of template instantiations
- equivalent masks can now trivially be used on mixed type simd
- choosing the right template parameters for `simd_mask` is still simple and consistent with `simd`

Cons
- It may be surprising that `simd_mask<T, Abi>` and `simd_mask<U, Abi>` are not different types. E.g. if someone wants to distinguish an overload set (why, though?). This is resolved by the alternative solution below.
- The `simd_mask::simd_type` member type must be removed or defined to an integer type of the corresponding `sizeof`. I.e. the mapping from mask type to simd type changes or doesn't exist anymore.
- Unary minus and unary plus on `simd_mask` objects must return an integer simd type. (`-(x < 0)` with `x` of type `simd<float>` would be of type `simd<int>` instead of `simd<float>`)
- `sizeof(T) > sizeof(long long)` needs a different rule to determine the `simd_type` member and/or the unary operator return types. (E.g. `sizeof(long double) == 12` on Linux 32-bit, `sizeof(long double) == 16` on Linux 64-bit) Proposed solution: Define `simd_mask<B, Abi>::simd_type` as `simd<I, Abi>` with `I` the largest standard signed integer type where `sizeof(I) <= B`.

Alternative: *Define the class template `template <size_t Bytes, class Abi> class simd_mask;` and let users deal with the difference in template parameters.*

Note that `simd<T, Abi>::mask_type` exists and is basically equivalent to an alias template parameterized on `T` and `Abi`.

5.3.1 suggested polls

Poll: Respecify `simd_mask<T, Abi>` as alias for `basic_simd_mask<sizeof(T), Abi>`.

SF	F	N	A	SA

Poll: Specify `simd_mask` as `simd_mask<Bytes, Abi>`.

SF	F	N	A	SA

Poll: Define `simd_mask<B, Abi>::simd_type` as `simd<I, Abi>` with `I` the largest standard signed integer type where `sizeof(I) <= B`.

SF	F	N	A	SA

Poll: Do not define a `simd_type` member type, but use the above rule for the return type of unary minus and unary plus.

SF	F	N	A	SA

5.4 reconsider conversion rules

LEWG discussed conversion in Issaquah 2023. There was some consensus in favor of: "SIMD types and operations should be value preserving, even if that means they're inconsistent with the builtin numeric types." Comments on this poll were:

- WF: I generally like the direction, in the sense of safety, but I'm not sure of the consequences of such a design.
- N: I would rather be consistently wrong than inconsistent.
- WA: I don't want to be inconsistent with how things have been for the last 30 years.
- WF: I'm not sure if we've evaluated the performance impact of this.
- N: Just because we've made a mistake in the past doesn't mean we have to keep making it. I'm not convinced that the new approach is the right thing.

LEWG then noticed a specifically tricky case when mixing `int` and `float`. Which of these do we want?

- `int * simd<float> = simd<float>`
- `int * simd<float> = simd<double>`

There was no consensus on how to resolve this correctly. Note that `int` (and to some extent `unsigned int`) have a special exception in the *value-preserving* conversion rules. Otherwise, a lot of code would be much more cumbersome to write (especially for types that have no matching literals, e.g. `short`).

With the discussion on safety and correctness in C++ getting more traction, there is a plausible chance that we will see a set of types that exhibit "safer" conversion behavior in the future. A possible approach could be via [P2509R0], defining a trait for value-preserving conversions. This

trait could then be used to define a numeric type, wrapping the existing arithmetic types with conversions being conditionally `explicit` depending on that trait.

When that happens, we will likely want to make these types *vectorizable* and then inherit the conversion rules of those types.

So why not let `simd<T>` inherit conversion behavior via `std::constructible_from` and `std::convertible_to`? Then such future numeric types conversion behavior will just fall in place.

The problem is that builtin arithmetic types behave differently to class types.

- They are all interconvertible.
- When a binary operator is called with mixed types, there are rules ([expr.arith.conv]) to determine a common type that go beyond anything a class type can do.
- Integral promotions.
- And to top it off, there are no literals for `(un)signed char` and `(un)signed short` (i.e. you cannot initialize without an implicit or explicit conversion — except value-init).

We could make all `simd` types interconvertible and implement binary operators similar to `simd<another_common_type_t<T, U>>(simd<T>, simd<U>)`[1]. Though, we surely want to ignore integral promotions (otherwise `simd<char>() + simd<char>()` has surprisingly bad performance characteristics). This would be going back to the original Vc library that was the basis of the `simd` proposal (back in 2014). Meaning, it has been done and it can be done. But those binary operators are not really a pretty solution either. And error messages when something goes wrong even less.

The type trait `another_common_type<T, U>` could implement `common_type` without integer promotion (and assorted other tweaks to reduce surprises[2]).

The comment "I would rather be consistently wrong than inconsistent." above was missing this information. There is no reasonable possibility to be "consistently wrong" here. We have no other choice than being inconsistent. Our only choice is what that inconsistency looks like.

In Issaquah, the idea of a "value-preserving common type" was voiced. This would presumably mean that `int + float` can turn into `double`. The trouble with such a trait is how to determine the correct answer if none of the argument types is a value-preserving common type. The situation is manageable as long as `simd` only supports arithmetic types. But there exists a plausible future where fixed-point numbers, rational numbers, `_BitInt`-like numbers, …become `vectorizable`. While I believe a solution could be developed, I am not convinced that it would serve users well. The tricky cases are probably best flagged as compile-time errors and resolved via explicit cast.

Choices:

1 which is basically what my first paper on SIMD vector types proposed [N4184]

2 Who knew that `common_type_t<std::integral_constant<int, -1>, unsigned short>` is `unsigned short`?

1. Try to be as "consistently wrong" (i.e. consistent with builtin types) as possible, at the cost of interconvertible types (leads to ambiguity errors), more complex binary operators, and many subtle ways to unintentionally change types and potentially also pay in performance.

2. Variant of the above: Don't make `simd` interconvertible, by defining the `simd` conversion constructor as in the wording below. However, the broadcast constructor is a simple `simd(value_type x)`, without any further constraints. Binary operators derive the return type (and common type for evaluating the operations) from a non-promoting common type of the operands.

3. Stick to the *value-preserving* rules with (or without) exceptions for `int` and `unsigned int` as introduced in the TS.

4. Consider a new rule-set that derives conversion rules (i.e. constraints on broadcast and conversion constructor) from a non-promoting common type of the participating types. Binary operators are still a simple hidden friend `simd operator+(simd, simd)`, requiring that the operands determine one of the operand types as their common type.

Choice 4 would basically make conversions conditionally explicit via `explicit(not std::same_as<another_common_type_t<T, U>, T>)`.

In such an implementation `int * simd<float>` is still of type `simd<float>` (as is `simd<int> * simd<float>` and `simd<long long> * simd<float>`).

With choice 3 the `simd<int> * simd<float>` and `simd<long long> * simd<float>` cases are ill-formed.

However, choice 4 makes initialization of `simd<short>` (and similar) cumbersome, since no literal other than a `char` literal is convertible to the `simd`. This is true for the plain value-preserving approach as well, which is why it has exceptions for `int` and `unsigned int` on broadcast constructor arguments. This same exception could be used for choice 3, of course.

Or, instead, we could allow arbitrary types if the library can determine at compile-time that the *value* of the argument can be converted without narrowing. This would be possible with `integral_constant` arguments, or rather anything with a `static constexpr value` data member. Then, `std::simd<short> x = std::cc<1>;` is well-formed while `std::simd<short> x = 1;` is not.[3]

For a better understanding, Tables 2–7 present the result type of binary `operator+` for different conversion strategies. Note that *the broadcast constructor has no influence on the results of choice 2*. This is why a simple broadcast constructor expecting a `value_type` argument should be the more consistent choice for this scenario.

3 Then we only need a language feature to turn constant expressions into `integral_constant`/`constexpr_t`/etc. via opt-in on the constructor/function argument and `std::cc` becomes transparent.

operand 1	P1928R5 (choice 3)	choice 4	choice 2
schar, uchar, short, simd<schar>, simd<uchar>, simd<short>	simd<short>		
ushort, uint, long, ulong, llong, ullong, float, double	ERROR		simd<operand1>
int	simd<short> (ERROR)	ERROR	simd<operand1>
bool	ERROR	simd<short>	
simd<ushort>, simd<uint>, simd<ulong>, simd<ullong>	ERROR	simd<operand1>	
simd<int>, simd<long>, simd<llong>, simd<float>, simd<double>	simd<operand1>		

Table 2: Binary operator results with simd<short> for operand 2

operand 1	P1928R5 (choice 3)	choice 4	choice 2
uchar, ushort, simd<uchar>, simd<ushort>	simd<ushort>		
long, ulong, llong, ullong, float, double	ERROR		simd<operand1>
int, uint	simd<ushort> (ERROR)	ERROR	simd<operand1>
schar, short, bool	ERROR	simd<ushort>	
simd<schar>, simd<short>	ERROR	simd<ushort>	
simd<int>, simd<uint>, simd<long>, simd<ulong>, simd<llong>, simd<ullong>, simd<float>, simd<double>	simd<operand1>		

Table 3: Binary operator results with simd<ushort> for operand 2

operand 1	P1928R5 (choice 3)	choice 4	choice 2
uchar, ushort, uint, simd<uchar>, simd<ushort>, simd<uint>	simd<uint>		
int	simd<uint> (ERROR)	simd<uint>	
simd<long>, simd<ulong>, simd<llong>, simd<ullong>, simd<double>	simd<operand1>		
long, ulong, llong, ullong, float, double	ERROR		simd<operand1>
schar, short, bool, simd<schar>, simd<short>, simd<int>,	ERROR	simd<uint>	
simd<float>	ERROR	simd<operand1>	

Table 4: Binary operator results with simd<uint> for operand 2

operand 1	P1928R5 (choice 3)	choice 4	choice 2
uchar, ushort, uint, ulong, simd<uchar>, simd<ushort>, simd<uint>, simd<ulong>	simd<ulong>		
int	simd<ulong> (ERROR)	simd<ulong>	
schar, short, long, bool, simd<schar>, simd<short>, simd<int>, simd<long>	ERROR	simd<ulong>	
llong, simd<llong>	ERROR		simd<ullong>
ullong	simd<ulong>	ERROR	simd<ullong>
simd<ullong>	simd<ullong>		
float, double	ERROR		simd<operand1>
simd<float>, simd<double>	ERROR	simd<operand1>	

Table 5: Binary operator results with simd<ulong> for operand 2

operand 1	P1928R5 (choice 3)	choice 4	choice 2
schar, uchar, short, ushort, float, simd<schar>, simd<uchar>, simd<short>, simd<ushort>, simd<float>	simd<float>		
uint, long, ulong, llong, ullong, bool, simd<int>, simd<uint>, simd<long>, simd<ulong>, simd<llong>, simd<ullong>	ERROR	simd<float>	
int	simd<float> (ERROR)	simd<float>	
double	ERROR		simd<double>
simd<double>	simd<double>		

Table 6: Binary operator results with simd<float> for operand 2

operand 1	P1928R5 (choice 3)	choice 4	choice 2
schar, uchar, short, ushort, int, uint, float, double, simd<schar>, simd<uchar>, simd<short>, simd<ushort>, simd<int>, simd<uint>, simd<float>, simd<double>	simd<double>		
long, ulong, llong, ullong, bool, simd<long>, simd<ulong>, simd<llong>, simd<ullong>	ERROR	simd<double>	

Table 7: Binary operator results with simd<double> for operand 2

5.4.1 suggested polls

Poll: Base `simd` conversion rules on the non-promoting common-type method instead of value-preserving conversions (choice 4 in P1928R5).

SF	F	N	A	SA

Poll: Base `simd` binary operator result types on the non-promoting common-type method while simplifying the broadcast constructor (choice 2 in P1928R5).

SF	F	N	A	SA

Poll: Remove broadcast ctor exceptions for `int` and `unsigned int`, instead ensure `integral_constant`-like arguments work correctly.

SF	F	N	A	SA

5.5 keep or remove split and concat or parts of it

The TS includes the following functions for splitting `simd` and `simd_mask` objects:

1. `std::experimental::split<N0, N1, N2, ...>(x)` returns a `std::tuple` of `simd`/`simd_mask` (`x.size()` must be `N0 + N1 + N2 ...`)
2. `std::experimental::split<T>(x)` returns a `std::array<T, ...>` (`T` is a `simd`/`simd_mask` and `x.size()` must be a multiple of `V::size()`)
3. `std::experimental::split_by<N>(x)` returns a `std::array` of `simd`/`simd_mask` (`x.size()` must be a multiple of `N`)

For concatenation the TS includes:

- `std::experimental::concat(x, y, z)` returns a single `simd`/`simd_mask` object containing the elements of `x`, `y`, and `z` (requires arguments to have the same element type)
- `std::experimental::concat(array_of_simd_or_mask)` returns a single `simd`/`simd_mask` object containing the elements of the `simd`/`simd_mask` objects in the `std::array`.

An obvious candidate for removal is the `concat(array)` overload, since the same can be expressed using `std::apply`. If it is not removed then why are there no additional overloads for all the other tuple-like types?

The concatenation facility for `simd`/`simd_mask` is useful and necessary for some of the permutation examples presented in [P2664R2]. At this point there exists no good motivation for removal of `concat` since we would expect a later paper to re-add this facility otherwise.

For `split` and `split_by` we should reconsider the usefulness of each of the three overloads in light of the intent to add an "extract" function, which is supposed to return a slice of the input `simd`/`simd_mask` as a smaller `simd`/`simd_mask`. Such a function would be a replacement for the 1. `split` function. The 2. and 3. function are equivalent: `split_by<N>(T)` is the same as `split<resize_simd_t<V::size() / N, T>>(T)`; `split<U>(T)` is the same as `split_by<T::size() / U::size()>(T)`. We could therefore consider reducing `split` to a single interface.

The use case I came across when wanting to use `split` is the splitting of an "oversized" `simd` into register-sized parts. Example: `fixed_size_simd<float, 20>` could be made up of one AVX-512 and one SSE register on an x86 target. So I'd be interested in a simple interface of splitting `fixed_size_simd<float, 20>` into `simd<float>` and `simd<float, impl-defined-abi-tag>`[4]. `std::experimental::split<16, 4>(x)` would achieve that. But in generic code the derivation of these numbers is not trivial.

An extended `split<T>(x)` interface could instead do the following: `split<simd<float>>(x)` returns a `tuple` of as many `simd<float>` as `x.size()` allows plus an "epilogue" of `simd<float, impl-defined-abi-tag>` objects as are necessary to return all elements of `x`. Then `split<simd<float>>(fixed_size_simd<float, 20>)` returns

- `tuple<simd<float>, resize_simd_t<4, simd<float>>>` with AVX-512,
- `tuple<simd<float>, simd<float>, resize_simd_t<4, simd<float>>>` with AVX, and
- `tuple<simd<float>, simd<float>, simd<float>, simd<float>, simd<float>>` with SSE.

If no "epilogue" is necessary, the return type could be an `array` instead of a `tuple`, in which case the SSE case above would instead return

- `array<simd<float>, 5>`.

More precise, the only remaining `split` function could be:

```
template<class V, class Abi>
  constexpr auto split(const simd<typename V::value_type, Abi>& x) noexcept;
```

1 Let N be `x.size() / V::size()`.

2 *Returns:*

- If `x.size() % V::size() == 0`, an `array<V, N>` with the i^{th} `simd` element of the j^{th} `array` element initialized to the value of the element in `x` with index i `+` j `* V::size()`.

4 same as `fixed_size_simd<float, 4>` depending on Section 5.2

- Otherwise, a `tuple` of N objects of type `V` and one or more objects of types `resize_simd_t<Sizes, V>...` such that `(Sizes + ...)` is equal to `x.size() % V::size()`. The i^{th} `simd` element of the j^{th} `tuple` element of type `V` is initialized to the value of the element in `x` with index i `+` j `* V::size()`. The i^{th} `simd` element of the j^{th} `tuple` with $j \geq N$ is initialized to the value of the element in `x` with index i `+` N `* V::size() +` sum of the first $j - N$ values in the `Sizes` pack.

5.5.1 Renaming split and concat

The names `split` and `concat` are very general. We have naming precedent with `tuple_cat`. If we follow the `tuple_cat` precedent the functions should be called `simd_split` and `simd_cat`.

There is an alternative, where we envision making `std::concat` the name for concatenation of all kinds of things in the standard library: simds, strings, arrays, vectors, ...[5]. The `split` interface, taking a template type to determine the split granularity, is not as obvious to generalize. `split<array<int, 4>>(array<int, 7>)`? For strings the interface should likely look more like `vector<string_view> split(string_view, string_view)`.

5.5.2 Suggested polls

Poll: Remove `concat(array<simd>)` overload.

SF	F	N	A	SA

Poll: Replace all `split`/`split_by` functions by proposed `split` function.

SF	F	N	A	SA

Poll: Rename `split` to `simd_split` and `concat` to `simd_cat`.

SF	F	N	A	SA

5.6 Integration with ranges

`simd` itself is not a container [P0851R0]. The value of a data-parallel object is not an array of elements but rather needs to be understood as a single opaque value that happens to have means for reading and writing element values. I.e. `simd<int> x = {};` does not start the lifetime of `int` objects. This implies that `simd` cannot model a contiguous range. But `simd` can trivially model `random_access_range`. However, in order to model `output_range`, the iterator of every non-const

5 all kinds of ranges?

`simd` would have to return an `element_reference` on dereference. Without the ability of `element_reference` to decay to the element type (similar to how arrays decay to pointers on deduction), I would prefer to simply make `simd` model only `random_access_range`.

If `simd` is a range, then `std::vector<std::simd<float>> data` can be flattened trivially via `data | std::views::join`. This makes the use of "arrays of `simd<T>`" easier to integrate into existing interfaces the expect "array of `T`".

I plan to pursue adding iterators and conversions to array and from random-access ranges, specifically `span` with static extent, in a follow-up paper. I believe it is not necessary to resolve this question before merging `simd` from the TS.

5.7 formatting support

If `simd` *is a* range, as suggested above and to be proposed in a follow-up paper, then `simd` will automatically be formatted as a range. This seems to be a good solution unless there is a demand to format `simd` objects differently from `random_access_range`.

5.8 std::hash

Is there a use case for `std::hash<simd<T>>`? In other words, is there a use case for using `simd<T>` as a map key? Recall that we do not consider `simd` to be a product type [P0851R0]. If there's no use case for hashing a `simd` object as one, is there a use case for multiple look-ups into a map, parallelizing the lookup as much as possible?

Consider a hash map with `int` keys and the task of looking up multiple keys in arbitrary order (unordered). In this case, one might want to pass a `simd<int>`, compute the hashes of `simd<int>::size()` keys in parallel (using SIMD instructions), and potentially determine the addresses (or offsets in contiguous memory) of the corresponding values in parallel. The value lookup could then use a SIMD gather instruction.

If we consider this use case important (or at least interesting), is `std::hash<simd<T>>` the right interface to compute hashes element-wise? After all, `simd` operations act element-wise unless strong hints in the API suggest otherwise.

At this point I would prefer to wait for concrete use cases of hashing `simd` objects before providing any standard interface. Specifically, at this point *we do not want `std::hash` support for `simd`*.

5.9 freestanding simd

Should `simd` be enabled for freestanding?

It seems we maybe don't want to enable `simd` for freestanding. Kernel code typically wants to have a small state for more efficient context switching. Therefore floating-point and SIMD registers are not used. However, we could limit `simd` to integers and the scalar ABI for freestanding. The utility of such a crippled `simd` is highly questionable.

So the question we need answered is whether there are uses cases for freestanding `simd`, or whether *all* freestanding code would outlaw `simd` anyway. In other words, do we have or will we have users asking for `simd` in freestanding?

5.10 Correct place for simd in the IS?

While `simd` is certainly very important for numerics and therefore fits into the "Numerics library" clause, it is also more than that. E.g. `simd` can be used for vectorization of text processing. In principle `simd` should be understood similar to fundamental types. Is the "General utilities library" clause a better place? Or rename "Concurrency support library" to "Parallelism and concurrency support library" and put it there? Alternatively, add a new library clause?

I am seeking feedback before making a recommendation.

5.11 element_reference is overspecified

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

This change is not reflected in the wording, pending encouragement from WG21 (mostly LWG).

6 Wording: Add Section 9 of N4808 with modifications

The following section presents the wording to be applied against the C++ working draft.

This wording overloads `operator?:` even though that's not possible (yet). As long as we don't have the language feature for overloading `?:` the following needs to happen:

1. Replace `operator?:` by `conditional_operator_impl`.
2. Add a `std::conditional_operator(cond, a, b)` CPO that tries the following:
   - If `conditional_operator_impl(cond, a, b)` can be found via ADL calls `conditional_operator_impl(cond, a, b)` unqualified, otherwise
   - if `cond ? a : b` is well-formed, return its result, otherwise
   - if `std::common_type_t<decltype(a), decltype(b)>` exists, cast `a` and `b` to this common type and apply `?:`.

In [version.syn], add

```
#define __cpp_lib_simd YYYYMML // also in <simd>
```

Adjust the placeholder value as needed so as to denote this proposal's date of adoption.

Add a new subclause after §28.8 [numerics.numbers]

(6.1) 28.9 Data-Parallel Types [simd]

(6.1.1) 28.9.1 General [simd.general]

1 The simd subclause defines data-parallel types and operations on these types. A data-parallel type consists of elements of an underlying vectorizable 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 The term *data-parallel type* refers to all *supported* (28.9.6.1) specializations of the `simd` and `simd_mask` class templates. A *data-parallel object* is an object of *data-parallel type*.

3 The set of *vectorizable types* comprises all cv-unqualified arithmetic types other than `bool`.

4 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.

5 Given a `simd_mask<T, Abi>` object `mask`, the *selected indices* signify the integers $i \in \{j \in \mathbb{N}_0 | j <$ `mask.size()` $\wedge$ `mask[`j`]`$\}$. Given an additional object `data` of type `simd<T, Abi>` or `simd_mask<T, Abi>`, the *selected elements* signify the elements `data[`i`]` for all selected indices i.

6 The conversion from vectorizable type `U` to vectorizable type `T` is *value-preserving* if all possible values of `U` can be represented with type `T`.

7 [*Note:* 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. — *end note*]

(6.1.2) 28.9.2 Header `<simd>` synopsis [simd.syn]

```
namespace std {
  namespace simd_abi {
    using scalar = see below;
    template<class T, size_t N> using fixed_size = see below;
    template<class T> inline constexpr size_t max_fixed_size = 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;
  }

  // 28.9.4, 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, class Abi = simd_abi::native<T>> struct simd_size;
template<class T, class Abi = simd_abi::native<T>>
  inline constexpr size_t simd_size_v = simd_size<T,Abi>::value;
```

// TODO: Rename to simd_alignment/simd_loadstore_alignment?

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

// 28.9.5, Load and store flags

```
template <class... Flags> struct loadstore_flags;
inline constexpr loadstore_flags<> loadstore_default{};
inline constexpr loadstore_flags<see below> loadstore_convert{};
inline constexpr loadstore_flags<see below> loadstore_aligned{};
template<size_t N> requires (has_single_bit<N>)
  inline constexpr loadstore_flags<see below> loadstore_overaligned{};
```

// 28.9.6, Class template `simd`

```
template<class T, class Abi = simd_abi::native<T>> class simd;
template<class T, size_t N> using fixed_size_simd = simd<T, simd_abi::fixed_size<T, N>>;
```

// 28.9.8, Class template `simd_mask`

```
template<class T, class Abi = simd_abi::native<T>> class simd_mask;
template<class T, size_t N> using fixed_size_simd_mask = simd_mask<T, simd_abi::fixed_size<T, N>>;
```

// 28.9.7.6, `simd` *and* `simd_mask` *creation*

```
template<size_t... Sizes, class T, class Abi>
  constexpr tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd<T, Abi>&) noexcept;
template<size_t... Sizes, class T, class Abi>
  constexpr tuple<simd_mask<T, simd_mask_abi::deduce_t<T, Sizes>>...>
    split(const simd_mask<T, Abi>&) noexcept;
template<class V, class Abi>
  constexpr array<V, simd_size_v<typename V::value_type, Abi> / V::size()>
    split(const simd<typename V::value_type, Abi>&) noexcept;
template<class V, class Abi>
  constexpr array<V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>
    split(const simd_mask<typename V::simd_type::value_type, Abi>&) noexcept;
```

```
template<size_t N, class T, class A>
  constexpr array<resize_simd_t<simd_size_v<T, A> / N, simd<T, A>>, N>
    split_by(const simd<T, A>& x) noexcept;
template<size_t N, class T, class A>
  constexpr array<resize_simd_t<simd_size_v<T, A> / N, simd_mask<T, A>>, N>
    split_by(const simd_mask<T, A>& x) noexcept;

template<class T, class... Abis>
  constexpr simd<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>>
    concat(const simd<T, Abis>&...) noexcept;
template<class T, class... Abis>
  constexpr simd_mask<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>>
    concat(const simd_mask<T, Abis>&...) noexcept;

template<class T, class Abi, size_t N>
  constexpr resize_simd_t<simd_size_v<T, Abi> * N, simd<T, Abi>>
    concat(const array<simd<T, Abi>, N>& arr) noexcept;
template<class T, class Abi, size_t N>
  constexpr resize_simd_t<simd_size_v<T, Abi> * N, simd_mask<T, Abi>>
    concat(const array<simd_mask<T, Abi>, N>& arr) noexcept;
```

// 28.9.9.5, `simd_mask` *reductions*

```
template<class T, class Abi> constexpr bool all_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr bool any_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr bool none_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr int reduce_count(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr int reduce_min_index(const simd_mask<T, Abi>&);
template<class T, class Abi> constexpr int reduce_max_index(const simd_mask<T, Abi>&);

constexpr bool all_of(T) noexcept;
constexpr bool any_of(T) noexcept;
constexpr bool none_of(T) noexcept;
constexpr int reduce_count(T) noexcept;
constexpr int reduce_min_index(T);
constexpr int reduce_max_index(T);
```

// 28.9.7.5, `simd` *reductions*

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(const simd<T, Abi>&, BinaryOperation = {});

template<class T, class Abi, class BinaryOperation>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     T identity_element, BinaryOperation binary_op);
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     plus<> binary_op = {}) noexcept;
```

```
  template<class T, class Abi>
    constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                       multiplies<> binary_op) noexcept;
  template<class T, class Abi>
    constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                       bit_and<> binary_op) noexcept;
  template<class T, class Abi>
    constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                       bit_or<> binary_op) noexcept;
  template<class T, class Abi>
    constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                       bit_xor<> binary_op) noexcept;

  template<class T, class Abi>
    constexpr T hmin(const simd<T, Abi>&) noexcept;
  template<class T, class Abi>
    constexpr T hmin(const simd<T, Abi>&, const typename simd<T, Abi>::mask_type&) noexcept;
  template<class T, class Abi>
    constexpr T hmax(const simd<T, Abi>&) noexcept;
  template<class T, class Abi>
    constexpr T hmax(const simd<T, Abi>&, const typename simd<T, Abi>::mask_type&) noexcept;

  // 28.9.7.7, Algorithms
  template<class T, class Abi>
    constexpr simd<T, Abi>
      min(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    constexpr simd<T, Abi>
      max(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    constexpr pair<simd<T, Abi>, simd<T, Abi>>
      minmax(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    constexpr simd<T, Abi>
      clamp(const simd<T, Abi>& v,
            const simd<T, Abi>& lo,
            const simd<T, Abi>& hi);
}
```

1 The header <simd> defines class templates, tag types, trait types, and function templates for element-wise operations on data-parallel objects.

(6.1.3) 28.9.3 simd ABI tags [simd.abi]

```
namespace simd_abi {
  using scalar = see below;
  template<class T, size_t N> using fixed_size = see below;
  template<class T> inline constexpr size_t max_fixed_size = implementation-defined;
```

```
  template<class T> using native = implementation-defined;
}
```

1 An *ABI tag* is a type in the `std::simd_abi` namespace that indicates a choice of size and binary representation for objects of data-parallel type. [*Note:* The intent is for the size and binary representation to depend on the target architecture. — *end note*] 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`.

2 [*Note:* The ABI tag is orthogonal to selecting the machine instruction set. The selected machine instruction set limits the usable ABI tag types, though (see 28.9.6.1). The ABI tags enable users to safely pass objects of data-parallel type between translation unit boundaries (e.g. function calls or I/O). — *end note*]

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_size_v<T, simd_abi::scalar>` equals 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_size_v<T, simd_abi::fixed_size<N>>` equals `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 28.9.6.1), 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 [*Note:* 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. — *end note*]

7 The type of `fixed_size<T, N>` in TU1 differs from the type of `fixed_size<T, N>` in TU2 iff the type of `native<T>` in TU1 differs from the type of `native<T>` in TU2.

8 An implementation may define additional *extended ABI tag* types in the `std::simd_abi` namespace, to support other forms of data-parallel computation.

9 `native<T>` is an implementation-defined alias for an ABI tag. [*Note:* 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 with ISA extensions, compiler flags may change the type of the `native<T>` alias. — *end note*] [*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*]

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

10 The member `type` shall be present if and only if

- `T` is a vectorizable type, and
- `simd_abi::fixed_size<N>` is supported (see 28.9.3), and
- every type in the `Abis` pack is an ABI tag.

11 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 28.9.6.1).

If `N` is `1`, the member typedef `type` is `simd_abi::scalar`. [*Note:* Implementations can base the choice on `Abis`, but can also ignore the `Abis` arguments. — *end note*]

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

(6.1.4) 28.9.4 `simd` type traits [simd.traits]

```
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, class Abi = simd_abi::native<T>> struct simd_size { see below };
```

7 `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`.

[*Note:* The rules are different from those in (28.9.6.1): The member `value` is present even if `simd<T, Abi>` is not supported for the currently targeted system. — *end note*]

8 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.

9 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 };
```

10 `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.

11 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 28.9.6.5 and 28.9.8.4). [*Note:* `value` identifies the alignment restrictions on pointers used for (converting) loads and stores for the give type `T` on arrays of type `U`. — *end note*]

12 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; };
```

13 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
- `simd_abi::deduce<T, simd_size_v<U, Abi0>, Abi0>` has a member type `type`.

14 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<U, Abi0>`.

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

15 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`.

16 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>`.

(6.1.5) 28.9.5 Load and store flags [simd.flags]

```
inline constexpr loadstore_flags<see below> loadstore_convert{};
inline constexpr loadstore_flags<see below> loadstore_aligned{};
template<size_t N> requires (has_single_bit<N>)
  inline constexpr loadstore_flags<see below> loadstore_overaligned{};
```

1 The template arguments to `loadstore_flags` are unspecified types used by the implementation to identify the different load and store flags.

2 There may be additional implementation-defined load and store flags.

(6.1.5.1) 28.9.5.1 Class template `loadstore_flags` overview [simd.flags.overview]

```
template <class... Flags> struct loadstore_flags {
  // 28.9.5.2, loadstore_flags operators
  template <class... Other>
    friend constexpr auto operator|(loadstore_flags, loadstore_flags<Other...>);

  template <class... Other>
    friend constexpr auto operator&(loadstore_flags, loadstore_flags<Other...>);
```

```
  template <class... Other>
    friend constexpr auto operator^(loadstore_flags, loadstore_flags<Other...>);
};
```

1 The class template `loadstore_flags` acts like a integer bit-flag for types.

2 *Constraints*: Every type in `Flags` is a valid template argument to one of `loadstore_convert`, `loadstore_aligned`, `loadstore_overaligned`, or to one of the implementation-defined load and store flags.

(6.1.5.2) 28.9.5.2 `loadstore_flags` operators [simd.flags.oper]

```
template <class... Other>
  friend constexpr auto operator|(loadstore_flags a, loadstore_flags<Other...> b);
```

1 *Returns:* A specialization of `loadstore_flags` identifying all load and store flags identified either by `a` or `b`.

```
template <class... Other>
  friend constexpr auto operator&(loadstore_flags a, loadstore_flags<Other...> b);
```

2 *Returns:* A specialization of `loadstore_flags` identifying all load and store flags identified both by `a` and `b`.

```
template <class... Other>
  friend constexpr auto operator^(loadstore_flags a, loadstore_flags<Other...> b);
```

3 *Returns:* A specialization of `loadstore_flags` identifying all load and store flags identified either by `a` or by `b`, but not both.

(6.1.6) 28.9.6 Class template `simd` [simd.class]

(6.1.6.1) 28.9.6.1 Class template `simd` overview [simd.overview]

```
template<class T, class Abi> class simd {
public:
  using value_type = T;
  using reference = see below;
  using mask_type = simd_mask<T, Abi>;
  using abi_type = Abi;

  static constexpr typename simd_size<T, Abi>::type size;

  constexpr simd() noexcept = default;

  // 28.9.6.4, simd constructors
  template<class U> constexpr simd(U&& value) noexcept;
  template<class U, class UAbi>
    constexpr explicit(see below) simd(const simd<U, UAbi>&) noexcept;
```

```
template<class G> constexpr explicit simd(G&& gen) noexcept;
template<typename It, class... Flags>
  constexpr simd(It first, loadstore_flags<Flags...> = {});
template<typename It, class... Flags>
  constexpr simd(It first, const mask_type& mask, loadstore_flags<Flags...> = {});
```

// 28.9.6.5, `simd` *copy functions*

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, loadstore_flags<Flags...> f = {});
template<typename It, class... Flags>
  constexpr void copy_from(It first, const mask_type& mask, loadstore_flags<Flags...> f = {});
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, loadstore_flags<Flags...> f = {}) const;
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, const mask_type& mask, loadstore_flags<Flags...> f = {}) const;
```

// 28.9.6.6, `simd` *subscript operators*

```
constexpr reference operator[](size_t) &;
constexpr value_type operator[](size_t) const&;
```

// 28.9.6.7, `simd` *unary operators*

```
constexpr simd& operator++() noexcept;
constexpr simd operator++(int) noexcept;
constexpr simd& operator--() noexcept;
constexpr simd operator--(int) noexcept;
constexpr mask_type operator!() const noexcept;
constexpr simd operator~() const noexcept;
constexpr simd operator+() const noexcept;
constexpr simd operator-() const noexcept;
```

// 28.9.7.1, `simd` *binary operators*

```
friend constexpr simd operator+(const simd&, const simd&) noexcept;
friend constexpr simd operator-(const simd&, const simd&) noexcept;
friend constexpr simd operator*(const simd&, const simd&) noexcept;
friend constexpr simd operator/(const simd&, const simd&) noexcept;
friend constexpr simd operator%(const simd&, const simd&) noexcept;
friend constexpr simd operator&(const simd&, const simd&) noexcept;
friend constexpr simd operator|(const simd&, const simd&) noexcept;
friend constexpr simd operator^(const simd&, const simd&) noexcept;
friend constexpr simd operator<<(const simd&, const simd&) noexcept;
friend constexpr simd operator>>(const simd&, const simd&) noexcept;
friend constexpr simd operator<<(const simd&, int) noexcept;
friend constexpr simd operator>>(const simd&, int) noexcept;
```

// 28.9.7.2, `simd` *compound assignment*

```
friend constexpr simd& operator+=(simd&, const simd&) noexcept;
```

```
  friend constexpr simd& operator-=(simd&, const simd&) noexcept;
  friend constexpr simd& operator*=(simd&, const simd&) noexcept;
  friend constexpr simd& operator/=(simd&, const simd&) noexcept;
  friend constexpr simd& operator%=(simd&, const simd&) noexcept;
  friend constexpr simd& operator&=(simd&, const simd&) noexcept;
  friend constexpr simd& operator|=(simd&, const simd&) noexcept;
  friend constexpr simd& operator^=(simd&, const simd&) noexcept;
  friend constexpr simd& operator<<=(simd&, const simd&) noexcept;
  friend constexpr simd& operator>>=(simd&, const simd&) noexcept;
  friend constexpr simd& operator<<=(simd&, int) noexcept;
  friend constexpr simd& operator>>=(simd&, int) noexcept;

  // 28.9.7.3, simd compare operators
  friend constexpr mask_type operator==(const simd&, const simd&) noexcept;
  friend constexpr mask_type operator!=(const simd&, const simd&) noexcept;
  friend constexpr mask_type operator>=(const simd&, const simd&) noexcept;
  friend constexpr mask_type operator<=(const simd&, const simd&) noexcept;
  friend constexpr mask_type operator>(const simd&, const simd&) noexcept;
  friend constexpr mask_type operator<(const simd&, const simd&) noexcept;

  // 28.9.7.4, simd conditional operators
  friend constexpr simd operator?:(const mask_type&, 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` is 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 28.9.3.

If `Abi` is an extended ABI tag, it is implementation-defined whether `simd<T, Abi>` is supported. [*Note:* The intent is for implementations to decide on the basis of the currently targeted system. — *end note*]

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 intialization performs no initialization of the elements; value-initialization initializes each element with `T()`. [*Note:* Thus, default initialization leaves the elements in an indeterminate state. *— end note*]

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

```
constexpr explicit operator implementation-defined() const;
constexpr 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]

(6.1.6.2) 28.9.6.2 `simd` width [simd.width]

```
static constexpr typename simd_size<T, Abi>::type size;
```

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

(6.1.6.3) 28.9.6.3 Element references [simd.reference]

1 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.

2 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;

  constexpr operator value_type() const noexcept;

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

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

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

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

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

constexpr operator value_type() const noexcept;
```

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

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

4 *Constraints*: `declval<value_type&>() = std::forward<U>(x)` is well-formed.

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

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

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

7 *Constraints*: `declval<value_type&>() @= std::forward<U>(x)` (where `@=` denotes the indicated compound assignment operator) is well-formed.

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

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

```
constexpr reference operator++() && noexcept;
constexpr reference operator--() && noexcept;
```

10 *Constraints*: The indicated operator can be applied to objects of type `value_type`.

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

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

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

13 *Remarks:* The indicated operator can be applied to objects of type `value_type`.

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

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

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

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

(6.1.6.4) 28.9.6.4 `simd` constructors [simd.ctor]

```
template<class U> constexpr simd(U&&) noexcept;
```

1 Let `From` denote the type `remove_cvref_t<U>`.

2 *Constraints*:

- `From` is a vectorizable type and the conversion from `From` to `value_type` is value-preserving, or
- `From` is not an arithmetic type and is implicitly convertible to `value_type`, or
- `From` is `int`, or
- `From` is `unsigned int` and `is_unsigned_v<value_type>` is `true`.

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

```
template<class U, class UAbi> constexpr explicit(see below) simd(const simd<U, UAbi>& x) noexcept;
```

4 *Constraints*: `simd_size_v<U, UAbi> == size()`.

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

6 *Remarks:* The constructor is `explicit`

- if the conversion from `U` to `value_type` is not value-preserving, or
- if both `U` and `value_type` are integral types and the integer conversion rank ([conv.rank]) of `U` is greater than the integer conversion rank of `value_type`, or
- if both `U` and `value_type` are floating-point types and the floating-point conversion rank ([conv.rank]) of `U` is greater than the floating-point conversion rank of `value_type`.

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

7 *Constraints*: `simd(gen(integral_constant<size_t, i>()))` is well-formed for all i in the range of `[0, size())`.

8 *Effects*: Constructs an object where the i^{th} element is initialized to `gen(integral_constant<size_t, i>())`.

9 The calls to `gen` are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by `gen` ([algorithms.parallel.exec]).

```
template<typename It, class... Flags>
  constexpr simd(It first, loadstore_flags<Flags...> = {});
```

10 *Constraints*:

- `iter_value_t<It>` is a vectorizable type, and
- `It` satisfies `contiguous_iterator`.

11 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `iter_value_t<It>` to `value_type` is value-preserving.

12 *Preconditions*:

- `[first, first + size())` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<It>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

13 *Effects*: Constructs an object where the i^{th} element is initialized to `static_cast<T>(first[`i`])` for all i in the range of `[0, size())`.

14 *Throws:* Nothing.

```
template<typename It, class... Flags>
  constexpr simd(It first, const mask_type& mask, loadstore_flags<Flags...> = {});
```

15 *Constraints*:

- `iter_value_t<It>` is a vectorizable type, and
- `It` satisfies `contiguous_iterator`.

16 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `iter_value_t<It>` to `value_type` is value-preserving.

17 *Preconditions*:

- For all selected indices i, `[first, first +` i`)` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<It>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

18 *Effects*: Constructs an object where the i^{th} element is initialized to `mask[`i`] ? static_cast<T>(first[`i`]) : T()` for all i in the range of `[0, size())`.

19 *Throws:* Nothing.

(6.1.6.5) 28.9.6.5 `simd` copy functions [simd.copy]

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, loadstore_flags<Flags...> f = {});
```

1 *Constraints*:

- `iter_value_t<It>` is a vectorizable type, and
- `It` satisfies `contiguous_iterator`.

2 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `iter_value_t<It>` to `value_type` is value-preserving.

3 *Preconditions*:

- `[first, first + size())` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<It>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

4 *Effects*: Replaces the elements of the `simd` object such that the i^{th} element is assigned with `static_cast<T>(first[`i`])` for all i in the range of `[0, size())`.

5 *Throws:* Nothing.

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, const mask_type& mask, loadstore_flags<Flags...> f = {});
```

6 *Constraints*:

- `iter_value_t<It>` is a vectorizable type, and
- `It` satisfies `contiguous_iterator`.

7 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `iter_value_t<It>` to `value_type` is value-preserving.

8 *Preconditions*:

- For all selected indices i, `[first, first +` i`)` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<It>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

9 *Effects*: Replaces the selected elements of the `simd` object such that the i^{th} element is replaced with `static_cast<T>(first[`i`])` for all selected indices i.

10 *Throws:* Nothing.

```
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, loadstore_flags<Flags...> f = {}) const;
```

11 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<Out>` is a vectorizable type, and
- `Out` satisfies `contiguous_iterator`, and
- `Out` satisfies `output_iterator<value_type>`.

12 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `value_type` to `iter_value_t<Out>` is value-preserving.

13 *Preconditions*:

- `[first, first + size())` is a valid range.
- `Out` models `contiguous_iterator`.
- `Out` models `output_iterator<value_type>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<Out>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

14 *Effects*: Copies all `simd` elements as if `first[`i`] = static_cast<iter_value_t<Out>>(operator[](`i`))` for all i in the range of `[0, size())`.

15 *Throws:* Nothing.

```
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, const mask_type& mask, loadstore_flags<Flags...> f = {}) const;
```

16 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<Out>` is a vectorizable type, and
- `Out` satisfies `contiguous_iterator`, and
- `Out` satisfies `output_iterator<value_type>`.

17 *Mandates*: If the template parameter pack `Flags` does not contain the type identifying `loadstore_convert`, then the conversion from `value_type` to `iter_value_t<Out>` is value-preserving.

18 *Preconditions*:

- For all selected indices i, `[first, first +` i`)` is a valid range.
- `Out` models `contiguous_iterator`.
- `Out` models `output_iterator<value_type>`.

- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd, iter_value_t<Out>>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

19 *Effects*: Copies the selected elements as if `first[`i`] = static_cast<iter_value_t<Out>>(operator[](`i`))` for all selected indices i.

20 *Throws:* Nothing.

(6.1.6.6) 28.9.6.6 `simd` subscript operators [simd.subscr]

```
constexpr reference operator[](size_t i) &;
```

1 *Preconditions*: `i < size()`.

2 *Returns:* A `reference` (see 28.9.6.3) referring to the i^{th} element.

3 *Throws:* Nothing.

```
constexpr value_type operator[](size_t i) const&;
```

4 *Preconditions*: `i < size()`.

5 *Returns:* The value of the i^{th} element.

6 *Throws:* Nothing.

(6.1.6.7) 28.9.6.7 `simd` unary operators [simd.unary]

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

```
constexpr simd& operator++() noexcept;
```

2 *Constraints*: Application of unary `++` to objects of type `value_type` is well-formed.

3 *Effects*: Increments every element by one.

4 *Returns:* `*this`.

```
constexpr simd operator++(int) noexcept;
```

5 *Constraints*: Application of unary `++` to objects of type `value_type` is well-formed.

6 *Effects*: Increments every element by one.

7 *Returns:* A copy of `*this` before incrementing.

```
constexpr simd& operator--() noexcept;
```

8 *Constraints*: Application of unary `--` to objects of type `value_type` is well-formed.

9 *Effects*: Decrements every element by one.

10 *Returns:* `*this`.

```
constexpr simd operator--(int) noexcept;
```

11 *Constraints*: Application of unary `--` to objects of type `value_type` is well-formed.

12 *Effects*: Decrements every element by one.

13 *Returns:* A copy of `*this` before decrementing.

```
constexpr mask_type operator!() const noexcept;
```

14 *Constraints*: Application of unary `!` to objects of type `value_type` is well-formed.

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

```
constexpr simd operator~() const noexcept;
```

16 *Constraints*: Application of unary `~` to objects of type `value_type` is well-formed.

17 *Returns:* A `simd` object where each bit is the inverse of the corresponding bit in `*this`.

```
constexpr simd operator+() const noexcept;
```

18 *Constraints*: Application of unary `+` to objects of type `value_type` is well-formed.

19 *Returns:* `*this`.

```
constexpr simd operator-() const noexcept;
```

20 *Constraints*: Application of unary `-` to objects of type `value_type` is well-formed.

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

(6.1.7) 28.9.7 `simd` non-member operations [simd.nonmembers]

(6.1.7.1) 28.9.7.1 `simd` binary operators [simd.binary]

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

1 *Constraints*: Application of the indicated operator to objects of type `value_type` is well-formed.

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

```
friend constexpr simd operator<<(const simd& v, int n) noexcept;
friend constexpr simd operator>>(const simd& v, int n) noexcept;
```

3 *Constraints*: Application of the indicated operator to objects of type `value_type` is well-formed.

4 *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, size())`.

(6.1.7.2) 28.9.7.2 `simd` compound assignment [simd.cassign]

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

1 *Constraints*: Application of the indicated operator to objects of type `value_type` is well-formed.

2 *Effects*: These operators apply the indicated operator to `lhs` and `rhs` as an element-wise operation.

3 *Returns:* `lhs`.

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

4 *Constraints*: Application of the indicated operator to objects of type `value_type` is well-formed.

5 *Effects*: Equivalent to: `return operator@=(lhs, simd(n));`

(6.1.7.3) 28.9.7.3 `simd` compare operators [simd.comparison]

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

1 *Constraints*: Application of the indicated operator to objects of type `value_type` is well-formed.

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

(6.1.7.4) 28.9.7.4 `simd` conditional operators [simd.cond]

```
friend constexpr simd operator?:(const mask_type& mask, const simd& a, const simd& b) noexcept;
```

1 *Returns:* A `simd` object where the i^{th} element equals `mask[`i`] ? a[`i`] : b[`i`]` for all i in the range of `[0, size())`.

(6.1.7.5) 28.9.7.5 `simd` reductions [simd.reductions]

1 In this subclause, `BinaryOperation` shall be a binary element-wise operation.

```
template<class T, class Abi, class BinaryOperation = plus<>>
  constexpr T reduce(const simd<T, Abi>& x, BinaryOperation binary_op = {});
```

2 *Mandates*: `binary_op` can be invoked with two arguments of type `simd<T, A1>` returning `simd<T, A1>` for every `A1` that is an ABI tag type.

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

4 *Throws:* Any exception thrown from `binary_op`.

```
template<class T, class Abi, class BinaryOperation>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     T identity_element, BinaryOperation binary_op);
```

5 *Mandates*: `binary_op` can be invoked with two arguments of type `simd<T, A1>` returning `simd<T, A1>` for every `A1` that is an ABI tag type.

6 *Preconditions*: The results of `all_of(x == binary_op(simd<T, A1>(identity_element), simd<T, A1>(x)))` and `all_of(simd<T, A1>(x) == binary_op(x, simd<T, A1>(identity_element)))` shall be `true` for every `A1` that is an ABI tag type and for all finite values `x` representable by `T`.

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

8 *Throws:* Any exception thrown from `binary_op`.

```
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     plus<> binary_op = {}) noexcept;
```

9 *Returns:* If `none_of(mask)`, returns `T()`. Otherwise, returns *GENERALIZED_SUM*`(binary_op, x[`i`], ...)` for all selected indices i.

```
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     multiplies<> binary_op) noexcept;
```

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

```
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     bit_and<> binary_op) noexcept;
```

11 *Constraints*: `is_integral_v<T>` is `true`.

12 *Returns:* If `none_of(mask)`, returns `~T()`. Otherwise, returns *GENERALIZED_SUM*`(binary_op, x[`i`], ...)` for all selected indices i.

```
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     bit_or<> binary_op) noexcept;
template<class T, class Abi>
  constexpr T reduce(const simd<T, Abi>& x, const typename simd<T, Abi>::mask_type& mask,
                     bit_xor<> binary_op) noexcept;
```

13 *Constraints*: `is_integral_v<T>` is `true`.

14 *Returns:* If `none_of(mask)`, returns `T()`. Otherwise, returns *GENERALIZED_SUM*`(binary_op, x[`i`], ...)` for all selected indices i.

```
template<class T, class Abi> constexpr T hmin(const simd<T, Abi>& x) noexcept;
```

15 *Constraints*: `T` satisfies `totally_ordered`.

16 *Preconditions*: `T` models `totally_ordered`.

17 *Returns:* The value of an element `x[`j`]` for which `x[`i`] < x[`j`]` is `false` for all i in the range of `[0, size())`.

```
template<class T, class Abi>
  constexpr T hmin(const simd<T, Abi>&, const typename simd<T, Abi>::mask_type&) noexcept;
```

18 *Constraints*: `T` satisfies `totally_ordered`.

19 *Preconditions*: `T` models `totally_ordered`.

20 *Returns:* If `none_of(mask)`, returns `numeric_limits<T>::max()`. Otherwise, returns the value of a selected element `x[`j`]` for which `x[`i`] < x[`j`]` is `false` for all selected indices i.

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

21 *Constraints*: `T` satisfies `totally_ordered`.

22 *Preconditions*: `T` models `totally_ordered`.

23 *Returns:* The value of an element `x[`j`]` for which `x[`j`] < x[`i`]` is `false` for all i in the range of `[0, size())`.

```
template<class T, class Abi>
  constexpr T hmax(const simd<T, Abi>&, const typename simd<T, Abi>::mask_type&) noexcept;
```

24 *Constraints*: `T` satisfies `totally_ordered`.

25 *Preconditions*: `T` models `totally_ordered`.

26 *Returns:* If `none_of(mask)`, returns `numeric_limits<V::value_type>::lowest()`. Otherwise, returns the value of a selected element `x.data[`j`]` for which `x.data[`j`] < x.data[`i`]` is `false` for all selected indices i.

(6.1.7.6) 28.9.7.6 `simd` and `simd_mask` creation [simd.creation]

```
template<size_t... Sizes, class T, class Abi>
  constexpr tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd<T, Abi>& x) noexcept;
template<size_t... Sizes, class T, class Abi>
  constexpr tuple<simd_mask<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd_mask<T, Abi>& x) noexcept;
```

1 *Constraints*: The sum of all values in the `Sizes` pack is equal to `simd_size_v<T, Abi>`.

2 *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 + sum of the first j values in the `Sizes` pack.

```
template<class V, class Abi>
  constexpr array<V, simd_size_v<typename V::value_type, Abi> / V::size()>
    split(const simd<typename V::value_type, Abi>& x) noexcept;
template<class V, class Abi>
  constexpr array<V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>
    split(const simd_mask<typename V::simd_type::value_type, Abi>& x) noexcept;
```

3 *Constraints*:

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

4 *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 `+` j `*` `V::size()`.

```
template<size_t N, class T, class A>
  constexpr array<resize_simd<simd_size_v<T, A> / N, simd<T, A>>, N>
    split_by(const simd<T, A>& x) noexcept;
template<size_t N, class T, class A>
  constexpr array<resize_simd<simd_size_v<T, A> / N, simd_mask<T, A>>, N>
    split_by(const simd_mask<T, A>& x) noexcept;
```

5 *Constraints*: `simd_size_v<T, A>` is an integral multiple of `N`.

6 *Returns:* An array `arr`, where `arr[`i`][`j`]` is initialized by `x[`i `* (simd_size_v<T, A> / N) +` j`]`.

```
template<class T, class... Abis>
  constexpr simd<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>> concat(
    const simd<T, Abis>&... xs) noexcept;
template<class T, class... Abis>
  constexpr simd_mask<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>> concat(
    const simd_mask<T, Abis>&... xs) noexcept;
```

7 *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 + the sum of the width of the first j parameters in the `xs` pack.

```
template<class T, class Abi, size_t N>
  constexpr resize_simd<simd_size_v<T, Abi> * N, simd<T, Abi>>
    concat(const array<simd<T, Abi>, N>& arr) noexcept;
template<class T, class Abi, size_t N>
  constexpr resize_simd<simd_size_v<T, Abi> * N, simd_mask<T, Abi>>
    concat(const array<simd_mask<T, Abi>, N>& arr) noexcept;
```

8 *Returns:* A data-parallel object, the i^{th} element of which is initialized by `arr[`i` / simd_size_v<T, Abi>][`i` % simd_size_v<T, Abi>]`.

(6.1.7.7) 28.9.7.7 Algorithms [simd.alg]

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

1 *Constraints*: `T` satisfies `totally_ordered`.

2 *Preconditions*: `T` models `totally_ordered`.

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

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

4 *Constraints*: `T` satisfies `totally_ordered`.

5 *Preconditions*: `T` models `totally_ordered`.

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

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

7 *Constraints*: `T` satisfies `totally_ordered`.

8 *Preconditions*: `T` models `totally_ordered`.

9 *Returns:* A `pair` initialized with

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

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

10 *Constraints*: `T` satisfies `totally_ordered`.

11 *Preconditions*: `T` models `totally_ordered`.

12 *Preconditions*: No element in `lo` shall be greater than the corresponding element in `hi`.

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

(6.1.7.8) 28.9.7.8 `simd` math library [simd.math]

1 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 `simd<T, Abi>` but are not of type `simd<T, Abi>` is well-formed.

2 Each function overload produced by the above rules applies the indicated `<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.

3 The result is unspecified if a domain, pole, or range error occurs when the input argument(s) are applied to the indicated `<cmath>` function. [*Note:* Implementations are encouraged to follow the C specification (especially Annex F). — *end note*]

4 TODO: Allow `abs(simd<signed-integral>)`.

5 If `abs` is called with an argument of type `simd<X, Abi>` for which `is_unsigned_v<X>` is `true`, the program is ill-formed.

(6.1.8) 28.9.8 Class template `simd_mask` [simd.mask.class]

(6.1.8.1) 28.9.8.1 Class template `simd_mask` overview [simd.mask.overview]

```
template<class T, class Abi> class simd_mask {
public:
  using value_type = bool;
  using reference = see below;
  using simd_type = simd<T, Abi>;
  using abi_type = Abi;

  static constexpr typename simd_size<T, Abi>::type size;

  constexpr simd_mask() noexcept = default;

  // 28.9.8.3, simd_mask constructors
```

```
constexpr explicit simd_mask(value_type) noexcept;
template<class U, class UAbi>
  constexpr explicit(sizeof(U) != sizeof(T)) simd_mask(const simd_mask<U, UAbi>&) noexcept;
template<class G> constexpr explicit simd_mask(G&& gen) noexcept;
template<typename It, class... Flags>
  constexpr simd_mask(It first, Flags = {});
template<typename It, class... Flags>
  constexpr simd_mask(It first, const simd_mask& mask, loadstore_flags<Flags...> = {});
```

// 28.9.8.4, `simd_mask` *copy functions*

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, loadstore_flags<Flags...> = {});
template<typename It, class... Flags>
  constexpr void copy_from(It first, const simd_mask& mask, loadstore_flags<Flags...> = {});
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, loadstore_flags<Flags...> = {}) const;
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, const simd_mask& mask, loadstore_flags<Flags...> = {}) const;
```

// 28.9.8.5, `simd_mask` *subscript operators*

```
constexpr reference operator[](size_t) &;
constexpr value_type operator[](size_t) const&;
```

// 28.9.8.6, `simd_mask` *unary operators*

```
constexpr simd_mask operator!() const noexcept;
constexpr simd_type operator+() const noexcept;
constexpr simd_type operator-() const noexcept;
constexpr simd_type operator~() const noexcept;
```

// 28.9.8.7, `simd_mask` *conversion operators*

```
template <class U, class A>
  constexpr explicit(sizeof(U) != sizeof(T)) operator simd<U, A>() const noexcept;
```

// 28.9.9.1, `simd_mask` *binary operators*

```
friend constexpr simd_mask operator&&(const simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask operator||(const simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask operator&(const simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask operator|(const simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask operator^(const simd_mask&, const simd_mask&) noexcept;
```

// 28.9.9.2, `simd_mask` *compound assignment*

```
friend constexpr simd_mask& operator&=(simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask& operator|=(simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask& operator^=(simd_mask&, const simd_mask&) noexcept;
```

// 28.9.9.3, `simd_mask` *comparisons*

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

    // 28.9.9.4, simd_mask conditional operators
    friend constexpr simd_mask operator?:(const simd_mask&, const simd_mask&, const simd_mask&) noexcept;
    friend constexpr simd_mask operator?:(const simd_mask&, bool, bool) noexcept;
    template <class T0, class T1>
      friend constexpr simd<see below, Abi>
        operator?:(const simd_mask&, const T0&, const T1&) noexcept;
  };
```

1 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()`.

2 Every specialization of `simd_mask` is 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 (28.9.3).

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

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>>`.

3 Default initialization performs no intialization of the elements; value-initialization initializes each element with `false`. [*Note:* Thus, default initialization leaves the elements in an indeterminate state. — *end note*]

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

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

5 The member type `reference` has the same interface as `simd<T, Abi>::reference`, except its `value_type` is `bool`. (28.9.6.3)

(6.1.8.2) 28.9.8.2 `simd_mask` width [simd.mask.width]

```
static constexpr typename simd_size<T, Abi>::type size;
```

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

(6.1.8.3) 28.9.8.3 `simd_mask` constructors [simd.mask.ctor]

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

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

```
template<class U, class UAbi>
  constexpr explicit(sizeof(U) != sizeof(T)) simd_mask(const simd_mask<U, UAbi>& x) noexcept;
```

2 *Constraints*: `simd_size_v<U, UAbi> == size()`.

3 *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())`.

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

4 *Constraints*: `static_cast<bool>(gen(integral_constant<size_t, i>()))` is well-formed for all i in the range of `[0, size())`.

5 *Effects*: Constructs an object where the i^{th} element is initialized to `gen(integral_constant<size_t, i>())`.

6 The calls to `gen` are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by `gen` ([algorithms.parallel.exec]).

```
template<typename It, class... Flags>
  constexpr simd_mask(It first, loadstore_flags<Flags...> = {});
```

7 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<It>` is of type `bool`, and
- `It` satisfies `contiguous_iterator`.

8 *Preconditions*:

- `[first, first + size())` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

9 *Effects*: Constructs an object where the i^{th} element is initialized to `first[`i`]` for all i in the range of `[0, size())`.

10 *Throws:* Nothing.

```
template<typename It, class... Flags>
  constexpr simd_mask(It first, const simd_mask& mask, loadstore_flags<Flags...> = {});
```

11 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<It>` is of type `bool`, and
- `It` satisfies `contiguous_iterator`.

12 *Preconditions*:

- For all selected indices i, `[first, first + i)` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

13 *Effects*: Constructs an object where the i^{th} element is initialized to `mask[i] ? first[i] : false` for all i in the range of `[0, size())`.

14 *Throws:* Nothing.

(6.1.8.4) 28.9.8.4 `simd_mask` copy functions [simd.mask.copy]

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, loadstore_flags<Flags...> = {});
```

1 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<It>` is of type `bool`, and
- `It` satisfies `contiguous_iterator`.

2 *Preconditions*:

- `[first, first + size())` is a valid range.
- `It` models `contiguous_iterator`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

3 *Effects*: Replaces the elements of the `simd_mask` object such that the i^{th} element is replaced with `first[i]` for all i in the range of `[0, size())`.

4 *Throws:* Nothing.

```
template<typename It, class... Flags>
  constexpr void copy_from(It first, const simd_mask& mask, loadstore_flags<Flags...> = {});
```

5 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<It>` is of type `bool`, and
- `It` satisfies `contiguous_iterator`.

6 *Preconditions*:

- For all selected indices i, `[first, first + i)` is a valid range.
- `It` models `contiguous_iterator`.

- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

7 *Effects*: Replaces the selected elements of the `simd_mask` object such that the i^{th} element is replaced with `first[`i`]` for all selected indices i.

8 *Throws:* Nothing.

```
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, loadstore_flags<Flags...> = {}) const;
```

9 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<Out>` is of type `bool`, and
- `Out` satisfies `contiguous_iterator`, and
- `Out` satisfies `output_iterator<value_type>`.

10 *Preconditions*:

- `[first, first + size())` is a valid range.
- `Out` models `contiguous_iterator`.
- `Out` models `output_iterator<value_type>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

11 *Effects*: Copies all `simd_mask` elements as if `first[`i`] = operator[](`i`)` for all i in the range of `[0, size())`.

12 *Throws:* Nothing.

```
template<typename Out, class... Flags>
  constexpr void copy_to(Out first, const simd_mask& mask, loadstore_flags<Flags...> = {}) const;
```

13 *Constraints*:

- `is_simd_flag_type_v<Flags>` is `true`, and
- `iter_value_t<Out>` is of type `bool`, and
- `Out` satisfies `contiguous_iterator`, and
- `Out` satisfies `output_iterator<value_type>`.

14 *Preconditions*:

- For all selected indices i, `[first, first +` i`)` is a valid range.
- `Out` models `contiguous_iterator`.
- `Out` models `output_iterator<value_type>`.
- If the template parameter pack `Flags` contains the type identifying `loadstore_aligned`, `addressof(*first)` shall point to storage aligned by `memory_alignment_v<simd_mask>`.

- If the template parameter pack `Flags` contains the type identifying `loadstore_overaligned<N>`, `addressof(*first)` shall point to storage aligned by `N`.

Effects: Copies the selected elements as if `first[`i`] = operator[](`i`)` for all selected indices i.

Throws: Nothing.

(6.1.8.5) 28.9.8.5 `simd_mask` subscript operators [simd.mask.subscr]

```
constexpr reference operator[](size_t i) &;
```

Preconditions: `i < size()`.

Returns: A `reference` (see 28.9.6.3) referring to the i^{th} element.

Throws: Nothing.

```
constexpr value_type operator[](size_t i) const&;
```

Preconditions: `i < size()`.

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

Throws: Nothing.

(6.1.8.6) 28.9.8.6 `simd_mask` unary operators [simd.mask.unary]

```
constexpr simd_mask operator!() const noexcept;
```

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

```
constexpr simd_type operator+() const noexcept;
constexpr simd_type operator-() const noexcept;
constexpr simd_type operator~() const noexcept;
```

Constraints: Application of the indicated unary operator to objects of type `T` is well-formed.

Returns: The result of applying the indicated operator to `static_cast<simd_type>(*this)`.

(6.1.8.7) 28.9.8.7 `simd_mask` conversion operators [simd.mask.conv]

```
template <class U, class A>
  constexpr explicit(sizeof(U) != sizeof(T)) operator simd<U, A>() const noexcept;
```

Constraints: `simd_size_v<U, A> == simd_size_v<T, Abi>`.

Returns: An object where the i^{th} element is initialized to `static_cast<U>(operator[](`i`))`.

(6.1.9) 28.9.9 Non-member operations [simd.mask.nonmembers]

(6.1.9.1) 28.9.9.1 `simd_mask` binary operators [simd.mask.binary]

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

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

(6.1.9.2) 28.9.9.2 `simd_mask` compound assignment [simd.mask.cassign]

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

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

2 *Returns:* `lhs`.

(6.1.9.3) 28.9.9.3 `simd_mask` comparisons [simd.mask.comparison]

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

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

(6.1.9.4) 28.9.9.4 `simd_mask` conditional operators [simd.mask.cond]

```
friend constexpr simd operator?:(const simd_mask& mask, const simd_mask& a, const simd_mask& b) noexcept;
```

1 *Returns:* A `simd_mask` object where the i^{th} element equals `mask[`i`] ? a[`i`] : b[`i`]` for all i in the range of `[0, size())`.

```
friend constexpr simd_mask operator?:(const simd_mask& mask, bool a, bool b) noexcept;
```

2 *Returns:* A `simd_mask` object where the i^{th} element equals `mask[`i`] ? a : b` for all i in the range of `[0, size())`.

```
template <class T0, class T1>
  friend constexpr simd<see below, Abi>
    operator?:(const simd_mask& mask, const T0& a, const T1& b) noexcept;
```

3 Let `U` be the common type of `T0` and `T1` without applying integral promotions on integral types with integer conversion rank less than the rank of `int`.

4 *Constraints*:

- `U` is a vectorizable type, and
- `sizeof(U) == sizeof(T)`, and
- `T0` satisfies `convertible_to<simd<U, Abi>>`, and
- `T1` satisfies `convertible_to<simd<U, Abi>>`.

Returns: A `simd<U, Abi>` object where the i^{th} element equals `mask[`i`] ? a : b` for all i in the range of `[0, size())`.

(6.1.9.5) 28.9.9.5 `simd_mask` reductions [simd.mask.reductions]

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

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

```
template<class T, class Abi> constexpr int reduce_min_index(const simd_mask<T, Abi>& k);
```

Preconditions: `any_of(k)` returns `true`.

Returns: The lowest element index i where `k[`i`]` is `true`.

Throws: Nothing.

```
template<class T, class Abi> constexpr int reduce_max_index(const simd_mask<T, Abi>& k);
```

Preconditions: `any_of(k)` returns `true`.

Returns: The greatest element index i where `k[`i`]` is `true`.

Throws: Nothing.

```
constexpr bool all_of(T) noexcept;
constexpr bool any_of(T) noexcept;
constexpr bool none_of(T) noexcept;
constexpr int reduce_count(T) noexcept;
```

11 *Returns:* `all_of` and `any_of` return their arguments; `none_of` returns the negation of its argument; `reduce_count` returns the integral representation of its argument.

12 *Remarks:* The parameter type `T` is an unspecified type that is only constructible via implicit conversion from `bool`.

```
constexpr int reduce_min_index(T);
constexpr int reduce_max_index(T);
```

13 *Preconditions*: The value of the argument is `true`.

14 *Returns:* `0`.

15 *Throws:* Nothing.

16 *Remarks:* The parameter type `T` is an unspecified type that is only constructible via implicit conversion from `bool`.

A Acknowledgments

Thanks to Daniel Towner, Jeff Garland, and Nicolas Morales for discussions and/or pull requests on this paper.

B Bibliography

[P2509R0] Giuseppe D'Angelo. *P2509R0: A proposal for a type trait to detect value-preserving conversions*. ISO/IEC C++ Standards Committee Paper. 2021. URL: https://wg21.link/p2509r0.

[P0927R2] James Dennett and Geoff Romer. *P0927R2: Towards A (Lazy) Forwarding Mechanism for C++*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0927r2.

[D0917] Matthias Kretz. *D0917: Making operator?: overloadable*. ISO/IEC C++ Standards Committee Paper. 2023. URL: https://web-docs.gsi.de/~mkretz/D0917.pdf.

[N4184] Matthias Kretz. *N4184: SIMD Types: The Vector Type & Operations*. ISO/IEC C++ Standards Committee Paper. 2014. URL: https://wg21.link/n4184.

[P0214R9] Matthias Kretz. *P0214R9: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0214r9.

[P0350R0] Matthias Kretz. *P0350R0: Integrating datapar with parallel algorithms and executors*. ISO/IEC C++ Standards Committee Paper. 2016. URL: https://wg21.link/p0350r0.

[P0851R0] Matthias Kretz. *P0851R0: simd<T> is neither a product type nor a container type*. ISO/IEC C++ Standards Committee Paper. 2017. URL: https://wg21.link/p0851r0.

[P1915R0] Matthias Kretz. *P1915R0: Expected Feedback from `simd` in the Parallelism TS 2*. ISO/IEC C++ Standards Committee Paper. 2019. URL: https://wg21.link/p1915r0.

[P0918R2] Tim Shen. *P0918R2: More simd<> Operations*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0918r2.

[P2664R2] Daniel Towner and Ruslan Arutyunyan. *P2664R2: Proposal to extend std::simd with permutation API*. ISO/IEC C++ Standards Committee Paper. 2023. URL: https://wg21.link/p2664r2.