STD::SIMD — MERGE DATA-PARALLEL TYPES
FROM THE PARALLELISM TS 2

ABSTRACT

After the Parallelism TS 2 was published in 2018, data-parallel types (std::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
  1.1 Changes from revision 0 1
  1.2 Changes from revision 1 2
  1.3 Changes from revision 2 2
  1.4 Changes from revision 3 3
  1.5 Changes from revision 4 4
  1.6 Changes from revision 5 4

2 Straw Polls 5
  2.1 SG1 at Kona 2022 5
  2.2 LEWG at Issaquah 2023 5

3 Introduction 7
  3.1 Related papers 8
4 Changes after TS feedback

4.1 improve ABI tags .................................................. 9
4.2 Simplify/generalize casts .......................................... 9
4.3 Add simd_mask generator constructor .............................. 10
4.4 Default load/store flags to element_aligned ..................... 10
4.5 Contiguous iterators for loads and stores ...................... 10
4.6 constexpr everything ............................................. 10
4.7 Specify simd::size as integral_constant ........................ 10
4.8 Replace where facilities ......................................... 10
4.9 Make use of int and size_t consistent ........................... 13
4.10 Clean up math function overloads ............................... 13
4.11 Add lvalue-qualifier to non-const subscript ................... 13
4.12 Rename simd_mask reductions .................................... 13
4.13 Added constraints on operators and functions to match their underlying element types ................................. 14
4.14 Rename alignment flags and extend load/store flags for opt-in to conversions .............................................. 14

5 Open questions

5.1 Renaming hmin and hmax or else .................................. 15
5.2 Simplify fixed_size ................................................. 18
5.3 Basic mask type? .................................................. 19
5.4 Reconsider conversion rules ...................................... 21
5.5 Keep or remove split and concat or parts of it ................ 26
5.6 Integration with ranges .......................................... 28
5.7 Formatting support ............................................... 29
5.8 std::hash .......................................................... 29
5.9 Freestanding SIMD ............................................... 29
5.10 Correct place for simd in the IS? ............................... 30
5.11 element_reference is overspecified .............................. 30
5.12 Remove long double from vectorizable types ................... 30
5.13 Increase minimum supported width to 64 ....................... 30

6 Wording: Add Section 9 of N4808 with modifications ................................. 30
28.9 Data-Parallel Types [simd] (6.1.0) .............................. 31

A Acknowledgments .................................................. 61

B Bibliography ..................................................... 61
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 `resizesimd` 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, 28.9.8.2, 28.9.8.3).
• 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 (28.9.8.5, 28.9.8.6)
• 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 \texttt{hmin/hmax} discussion with more extensive naming discussion (Section 5.1).
- Discuss freestanding \texttt{simd} (Section 5.9).
- Discuss \texttt{split} and \texttt{concat} (Section 5.5).
- Apply the new library specification style from P0788R3.

### 1.5 Changes from revision 4

Previous revision: P1928R4

- Added \texttt{simd_select} discussion (Section 4.8.1).

### 1.6 Changes from revision 5

Previous revision: P1928R5

- Updated the wording for changes discussed in and requested by LEWG in Varna.
- Rename to \texttt{simd\_cat} and \texttt{simd\_split}.
- Drop \texttt{simd\_cat(array)} overload.
- Replace \texttt{ simd\_split} by \texttt{simd\_split} as proposed in P1928R4.
- Use \texttt{indirectly\_writable} instead of \texttt{output\_iterator}.
- Replace most \texttt{size\_t} and \texttt{int} uses by \texttt{simd\_size\_type} signed integer type.
- Remove everything in \texttt{simd\_abi} and the namespace itself.
- Reword section on ABI tags using exposition-only ABI tag aliases.
- Guarantee generator ctor calls callable exactly once per index.
- Remove \texttt{int/unsigned \_int exception from conversion rules of broadcast ctor}.
- Rename \texttt{loadstore\_flags} to \texttt{simd\_flags}.
- Make \texttt{simd\_flags::operator\_consteval}.
- Remove \texttt{simd\_flags::operator\&} and \texttt{simd\_flags::operator\^}.
- Increase minimum SIMD width to 64.
- Rename \texttt{hmin/hmax} to \texttt{reduce\_min and reduce\_max}. 
• Refactor simd_mask<T, Abi> to basic_simd_mask<Bytes, Abi> and replace all occurrences accordingly.

• Rename simd<T, Abi> to basic_simd<Bytes, Abi> and replace all occurrences accordingly.

• Remove long double from the set of vectorizable types.

• Remove is_abi_tag, is_simd, and is_simd_mask traits.

• Make simd_size exposition-only.

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)

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.2 lewg at issaquah 2023

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Poll: simd masked operations should look like (vote for as many options as you’d like):

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>where(u &gt; 0, v).copy_from(ptr)</td>
<td>12</td>
</tr>
<tr>
<td>v.copy_from(u &gt; 0, ptr)</td>
<td>1</td>
</tr>
<tr>
<td>v.copy_from_if(ptr, u &gt; 0)</td>
<td>2</td>
</tr>
<tr>
<td>v.copy_from(ptr, u &gt; 0)</td>
<td>14</td>
</tr>
<tr>
<td>v.copy_from(u &gt; 0, ptr)</td>
<td>3</td>
</tr>
<tr>
<td>v.copy_from_where(u &gt; 0, ptr)</td>
<td>4</td>
</tr>
<tr>
<td>v.copy_from_where(ptr, u &gt; 0)</td>
<td>11</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>where(u &gt; 0, v).copy_from(ptr)</td>
<td>5</td>
</tr>
<tr>
<td>v.copy_from(ptr, u &gt; 0)</td>
<td>12</td>
</tr>
<tr>
<td>v.copy_from_where(ptr, u &gt; 0)</td>
<td>6</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>
Poll: simd should be a range.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Poll: Rename popcount to reduce_count.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: Rename find_first_set/find_last_set to reduce_min_index/reduce_max_index.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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 indepen-
    dently.

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 ship-
ning 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 Changes after TS feedback

4.3  
ADD SIMD_MASK GENERATOR CONSTRUCTOR

This constructor was added:

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

For a discussion, see P1928R3 Section 4.3.

4.4  
DEFAULT LOAD/STORE FLAGS TO ELEMENTAligned

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

4.5  
CONTINUOUS 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:

```cpp
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:
4 Changes after TS feedback

- 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_ALIGNMENT** 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:

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

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:
<table>
<thead>
<tr>
<th>TS name</th>
<th>Ideas</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::experimental::element_aligned</td>
<td>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</td>
<td>std::loadstore_default</td>
</tr>
<tr>
<td>std::experimental::vector_aligned</td>
<td>std::simd_aligned, std::simd_flag_aligned, std::simd_copy_aligned, std::loadstore_aligned, std::simd_flags::aligned</td>
<td>std::loadstore_aligned</td>
</tr>
<tr>
<td>std::experimental::overaligned&lt;N&gt;</td>
<td>std::overaligned&lt;N&gt;, std::simd_overaligned&lt;N&gt;, std::simd_flag_overaligned&lt;N&gt;, std::simd_copy_overaligned&lt;N&gt;, std::loadstore_overaligned&lt;N&gt;, std::simd_flags::overaligned&lt;N&gt;</td>
<td>std::loadstore_overaligned&lt;N&gt;</td>
</tr>
<tr>
<td>-</td>
<td>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</td>
<td>std::loadstore_convert</td>
</tr>
</tbody>
</table>

Table 1: Name alternatives for alignment flags
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:

```cpp
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.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

Poll: Rename to reduce_min and reduce_max.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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:

```cpp
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, 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>`.

<table>
<thead>
<tr>
<th></th>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 built-in 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 built-in 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.

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. Try to be as "consistently wrong" (i.e. consistent with built-in 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.

\(^1\) 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`?
5 Open questions

2. Variant of the above: Don’t make \texttt{simd} interconvertible, by defining the \texttt{simd} conversion constructor as in the wording below. However, the broadcast constructor is a simple \texttt{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 \textit{value-preserving} rules with (or without) exceptions for \texttt{int} and \texttt{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 \texttt{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 \texttt{explicit(not std::same\_as<another\_common\_type\_t<T, U>, T>)}.

In such an implementation \texttt{int * simd<float>} is still of type \texttt{simd<float>} (as is \texttt{simd<int> * simd<float>} and \texttt{simd<long\_long> * simd<float>}).

With choice 3 the \texttt{simd<int> * simd<float>} and \texttt{simd<long\_long> * simd<float>} cases are ill-formed.

However, choice 4 makes initialization of \texttt{simd<short>} (and similar) cumbersome, since no literal other than a \texttt{char} literal is convertible to the \texttt{simd}. This is true for the plain value-preserving approach as well, which is why it has exceptions for \texttt{int} and \texttt{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 \texttt{value} of the argument can be converted without narrowing. This would be possible with \texttt{integral\_constant} arguments, or rather anything with a \texttt{static constexpr} value data member. Then, \texttt{std::simd<short> x = std::cc<1>;} is well-formed while \texttt{std::simd<short> x = 1;} is not.\footnote{Then we only need a language feature to turn constant expressions into \texttt{integral\_constant/constexpr\_t/etc.} via opt-in on the constructor/function argument and \texttt{std::cc} becomes transparent.}

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 \texttt{value\_type} argument should be the more consistent choice for this scenario.

5.4.1 suggested polls

Poll: Base \texttt{simd} conversion rules on the non-promoting common-type method instead of value-preserving conversions (choice 4 in P1928R6).

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
SF & F & N & A & SA \\
\hline
\end{tabular}
\end{center}
5 Open questions

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar, uchar, short, simd&lt;schar&gt;, simd&lt;uchar&gt;, simd&lt;short&gt;</td>
<td>simd&lt;short&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ushort, uint, long, ulong, llong, ullong, float, double</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>simd&lt;short&gt; (ERROR)</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
</tr>
<tr>
<td>bool</td>
<td>ERROR</td>
<td>simd&lt;short&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;uchar&gt;, simd&lt;ushort&gt;, simd&lt;long&gt;, simd&lt;ulong&gt;</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;int&gt;, simd&lt;long&gt;, simd&lt;llong&gt;, simd&lt;u64&gt;, simd&lt;long&gt;, simd&lt;double&gt;</td>
<td></td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar, uchar, short, simd&lt;uchar&gt;, simd&lt;ushort&gt;</td>
<td>simd&lt;ushort&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long, ulong, llong, ullong, float, double</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
<tr>
<td>int, uint</td>
<td>simd&lt;ushort&gt; (ERROR)</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
</tr>
<tr>
<td>schar, short, bool</td>
<td>ERROR</td>
<td>simd&lt;ushort&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;schar&gt;, simd&lt;short&gt;</td>
<td>ERROR</td>
<td>simd&lt;ushort&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;int&gt;, simd&lt;long&gt;, simd&lt;llong&gt;, simd&lt;ulong&gt;, simd&lt;int&gt;, simd&lt;double&gt;</td>
<td></td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar, uchar, uint, simd&lt;uchar&gt;, simd&lt;uchar&gt;, simd&lt;ushort&gt;, simd&lt;int&gt;</td>
<td>simd&lt;int&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>simd&lt;int&gt; (ERROR)</td>
<td>simd&lt;int&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;long&gt;, simd&lt;ulong&gt;, simd&lt;l64&gt;, simd&lt;u64&gt;, simd&lt;long&gt;, simd&lt;double&gt;</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long, ulong, llong, ullong, float, double</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
<tr>
<td>schar, short, bool, simd&lt;schar&gt;, simd&lt;short&gt;, simd&lt;int&gt;, simd&lt;float&gt;</td>
<td>ERROR</td>
<td>simd&lt;int&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;float&gt;</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Binary operator results with simd<int> for operand 2
### 5 Open Questions

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uchar, ushort, uint, ulong, simd&lt;uchar&gt;, simd&lt;ushort&gt;, simd&lt;uint&gt;, simd&lt;ulong&gt;</td>
<td>simd&lt;ulong&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>simd&lt;ulong&gt; (ERROR)</td>
<td>simd&lt;ulong&gt;</td>
<td></td>
</tr>
<tr>
<td>schar, short, long, bool, simd&lt;schar&gt;, simd&lt;short&gt;, simd&lt;int&gt;, simd&lt;long&gt;</td>
<td>ERROR</td>
<td>simd&lt;ulong&gt;</td>
<td></td>
</tr>
<tr>
<td>llong, simd&lt;llong&gt;</td>
<td>ERROR</td>
<td>simd&lt;ulong&gt;</td>
<td></td>
</tr>
<tr>
<td>ulong</td>
<td>simd&lt;ulong&gt;</td>
<td>ERROR</td>
<td>simd&lt;ulong&gt;</td>
</tr>
<tr>
<td>simd&lt;ulong&gt;</td>
<td>simd&lt;ulong&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float, double</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;float&gt;, simd&lt;double&gt;</td>
<td>ERROR</td>
<td>simd&lt;operand1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>schar, uchar, short, ushort, float, simd&lt;schar&gt;, simd&lt;uchar&gt;, simd&lt;ushort&gt;, simd&lt;float&gt;</td>
<td>simd&lt;float&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint, long, ulong, llong, ullong, bool, simd&lt;int&gt;, simd&lt;long&gt;, simd&lt;ulong&gt;, simd&lt;llong&gt;, simd&lt;ulong&gt;</td>
<td>ERROR</td>
<td>simd&lt;float&gt;</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>simd&lt;float&gt; (ERROR)</td>
<td>simd&lt;float&gt;</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>ERROR</td>
<td>simd&lt;double&gt;</td>
<td></td>
</tr>
<tr>
<td>simd&lt;double&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>operand 1</th>
<th>P1928R6 (choice 3)</th>
<th>choice 4</th>
<th>choice 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>schar, uchar, short, ushort, int, uint, float, double, simd&lt;schar&gt;, simd&lt;uchar&gt;, simd&lt;ushort&gt;, simd&lt;int&gt;, simd&lt;uint&gt;, simd&lt;float&gt;, simd&lt;double&gt;</td>
<td>simd&lt;double&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long, ulong, llong, ullong, bool, simd&lt;long&gt;, simd&lt;ulong&gt;, simd&lt;llong&gt;, simd&lt;ulong&gt;, simd&lt;llong&gt;</td>
<td>ERROR</td>
<td>simd&lt;double&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Binary operator results with **simd<double>** for operand 2
Poll: Base `simd` binary operator result types on the non-promoting common-type method while simplifying the broadcast constructor (choice 2 in P1928R6).

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

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>.

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) re-
turns 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:

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

1. Let \( N \) be \( x\text{.size()} / V\text{.size()} \).

2. Returns:

   • If \( x\text{.size()} \% V\text{.size()} = 0 \), an array\( V, N \) with the \( i \text{th} \) simd element of the \( j \text{th} \) array element
     initialized to the value of the element in \( x \) with index \( i + j \times V\text{.size()} \).

   • 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 \( x\text{.size()} \% V\text{.size()} \). The \( i \text{th} \) simd element of the \( j \text{th} \)
     tuple element of type \( V \) is initialized to the value of the element in \( x \) with index \( i + j \times V\text{.size()} \).
     The \( i \text{th} \) simd element of the \( j \text{th} \) tuple with \( j \geq N \) is initialized to the value of the element in \( x \) with
     index \( i + N \times V\text{.size()} + \text{sum of the first } j - N \text{ values in the Sizes pack.}
```

4 same as fixed_size_simd<float, 4> depending on Section 5.2
5 Open questions

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.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

Poll: Rename split to simd_split and concat to simd_cat.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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 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".

⁵ all kinds of ranges?
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).

5.12 **REMOVE LONG_DOUBLE FROM VECTORIZABLE TYPES**

Rationale: TS experience. It’s a headache. It’s not worth the specification and implementation effort.

Poll: Remove `long double` from vectorizable types

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

5.13 **INCREASE MINIMUM SUPPORTED WIDTH TO 64**

Rationale: AVX-512 `simd<char>::size() == 64`.

Poll: Increase the minimum supported `simd` width to 64

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
</table>

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 YYYYMM // 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 basic_simd and basic_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 and long double.

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 basic_simd_mask<Bytes, Abi> object mask, the selected indices signify the integers $i \in \{j \in \mathbb{N}_0 | j < \text{mask.size()} \land \text{mask}[j]\}$. Given an additional object data of type basic_simd<T, Abi> or basic_simd_mask<Bytes, 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 {
    using simd-size-type = see below; // exposition only
    template <typename T> constexpr size_t mask-element-size = see below; // exposition only
    template <size_t Bytes> using integer-from = see below; // exposition only

    // 28.9.3, simd ABI tags
    template<class T> using native-abi = see below; // exposition only
    template<class T, simd-size-type N> using deduce-t = see below; // exposition only

31
// 28.9.4, simd type traits
template<class T, class U = typename T::value_type> struct simd_alignment;
template<class T, class U = typename T::value_type>
    inline constexpr size_t simd_alignment_v = simd_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<simd-size-type N, class V> struct resize_simd { using type = see below; };
template<simd-size-type N, class V> using resize_simd_t = typename resize_simd<N, V>::type;

// 28.9.5, Load and store flags
  template <class... Flags> struct simd_flags;
inline constexpr simd_flags <> simd_flag_default {};
inline constexpr simd_flags<> simd_flag_convert {};
inline constexpr simd_flags<> simd_flag_aligned {};
  template< size_t N> requires (has_single_bit(N))
inline constexpr simd_flags<> simd_flag_overaligned {};

// 28.9.6, Class template basic_simd
  template<class T, class Abi = native-abi<T>> class basic_simd;
template<class T, simd-size-type N = basic_simd<T>::size()> using simd = basic_simd<T, deduce-t<T, N>>;

// 28.9.8, Class template basic_simd_mask
  template<size_t Bytes, class Abi = native-abi<T>> class basic_simd_mask;
template<class T, simd-size-type N = basic_simd_mask<sizeof(T)>::size()> using simd_mask = basic_simd_mask<sizeof(T), deduce-t<T, N>>;

// 28.9.7.6, basic_simd and basic_simd_mask creation
  template<class V, class Abi>
    constexpr auto simd_split (const basic_simd<typename V::value_type, Abi>& x) noexcept;
  template<class M, class Abi>
    constexpr auto simd_split (const basic_simd_mask<mask-element-size<M>, Abi>& x) noexcept;
  template<class T, class... Abis>
    constexpr basic_simd<T, deduce-t<T, (basic_simd<T, Abis>::size + ...)>>
    simd_cat (const basic_simd<T, Abis>&...) noexcept;
  template<size_t Bytes, class... Abis>
    constexpr basic_simd_mask<Bytes, deduce-t<integer-from<Bytes>, (basic_simd_mask<Bytes, Abis>::size() + ...)>
    simd_cat (const basic_simd_mask<Bytes, Abis>&...) noexcept;

// 28.9.9.5, basic_simd_mask reductions
  template<size_t Bytes, class Abi>
    constexpr bool all_of (const basic_simd_mask<Bytes, Abi>&) noexcept;
template< size_t Bytes, class Abi> constexpr bool any_of (const basic_simd_mask<Bytes, Abi>&) noexcept ;
template< size_t Bytes, class Abi> constexpr bool none_of (const basic_simd_mask<Bytes, Abi>&) noexcept ;
template< size_t Bytes, class Abi> constexpr simd-size-type reduce_count (const basic_simd_mask<Bytes, Abi>&) noexcept ;
template< size_t Bytes, class Abi> constexpr simd-size-type reduce_min_index (const basic_simd_mask<Bytes, Abi>&); 
template< size_t Bytes, class Abi> constexpr simd-size-type reduce_max_index (const basic_simd_mask<Bytes, Abi>&);

constexpr bool all_of(T) noexcept;
constexpr bool any_of(T) noexcept;
constexpr bool none_of(T) noexcept;
constexpr simd-size-type reduce_count(T) noexcept;
constexpr simd-size-type reduce_min_index(T); 
constexpr simd-size-type reduce_max_index(T);

// 28.9.7.5, basic_simd reductions

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

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

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

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

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

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

// 28.9.7.7, Algorithms
template<class T, class Abi>
constexpr basic_simd<T, Abi>
min(const basic_simd<T, Abi>& a, const basic_simd<T, Abi>& b) noexcept;

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

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

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

template<class T, class U>
constexpr auto simd_select (bool c, const T& a, const U& b) -> remove_cvref_t<decltype(c ? a : b)>;

template< size_t Bytes, class Abi, typename T, typename U>
constexpr auto simd_select (const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b);

1 The header <simd> defines class templates, tag types, trait types, and function templates for element-wise operations on data-parallel objects.
2 simd-size-type is an exposition-only alias for a signed integer type.
3 mask-element-size<basic_simd_mask<Bytes, Abi>> has the value Bytes.
4 integer-from<Bytes> is an alias for a signed integer type T so that sizeof(T) == Bytes.

(6.1.3) 28.9.3 simd ABI tags

template<class T> using native-abi = see below; // exposition only
template<class T, size_t N> using deduce-t = see below; // exposition only

1 An ABI tag is a type 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 basic_simd and basic_simd_mask.

[ 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 An implementation defines ABI tag types as necessary for the following exposition-only aliases.
4 The exposition-only alias deduce-t<T, N> results in a substitution failure if

- T is not a vectorizable type, or
- N is larger than an implementation-defined maximum.

The implementation-defined maximum for N is no smaller than 64.

5 Where present, the exposition-only alias deduce-t<T, N> names an ABI tag type that satisfies

- basic_simd<T, deduce-t<T, N> >::size == N, and
- basic_simd<T, deduce-t<T, N> > is default constructible (see 28.9.6.1).
native-abi<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-abi<T> alias. — end note ] [ Example: Consider a target architecture supporting the ABI tags __simd128 and __simd256, where hardware support for __simd256 only exists for floating-point types. The implementation therefore defines native-abi<T> as an alias for

- __simd256 if T is a floating-point type, and
- __simd128 otherwise.

— end example ]

The type of deduce-t<T, N> in TU1 differs from the type of deduce-t<T, N> in TU2 iff the type of native-abi<T> in TU1 differs from the type of native-abi<T> in TU2.

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

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

1 simd_alignment<T, U> shall have a member value if and only if

- T is a specialization of basic_simd_mask and U is bool, or
- T is a specialization of basic_simd and U is a vectorizable type.

2 If value is present, the type simd_alignment<T, U> is a BinaryTypeTrait with a base characteristic of integral_constant<sizeof_t, N> for some implementation-defined N (see 28.9.6.5 and 28.9.8.3). [ 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 ]

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

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

4 The member type is present if and only if

- V is either basic_simd<V, Abi0> or basic_simd_mask<Bytes, Abi0>, where V, Bytes, and Abi0 are deduced from V, and
- T is a vectorizable type, and
- simd_abi::deduce<T, basic_simd<V, Abi0>::size, Abi0> has a member type.

5 Let Abi1 denote the type deduce_t<T, basic_simd<U, Abi0>::size, Abi0>. Where present, the member typedef type names basic_simd<T, Abi1> if V is basic_simd<U, Abi0> or basic_simd_mask<sizeof(T), Abi1> if V is basic_simd_mask<Bytes, Abi0>.

```cpp
template<simd-size-type N, class V> struct resize_simd { using type = see below; };
```

6 The member type is present if and only if

- V is either basic_simd<T, Abi0> or basic_simd_mask<Bytes, Abi0>, where T, Bytes, and Abi0 are deduced from V, and
- simd_abi::deduce<T, N, Abi0> has a member type type.
Let \( \text{Abi} \) denote the type \( \text{deduce_t}(<T, N, \text{Abi}_0> \text{. Where present, the member typedef type names basic_simd<T, \text{Abi}_1> \text{ if V is basic_simd<T, Abi}_0> \text{ or basic_simd_mask<Bytes, Abi}_1> \text{ if V is basic_simd_mask<Bytes, Abi}_0>.} \)

(6.1.5) 28.9.5 Load and store flags

\[
\begin{align*}
&\text{inline constexpr simd_flags <see below> simd_flag_convert{};} \\
&\text{inline constexpr simd_flags <see below> simd_flag_aligned{}}; \\
&\text{template<\text{size_t N}> requires (has_single_bit<N>)} \\
&\quad \text{inline constexpr simd_flags <see below> simd_flag_overaligned{}};
\end{align*}
\]

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

(6.1.5.1) 28.9.5.1 Class template simd_flags overview

\[
\text{template <class... Flags> struct simd_flags { \\
// 28.9.5.2, simd_flags operators \\
\text{template <class... Other>} \\
\quad \text{friend constexpr auto operator|(simd_flags a, simd_flags<Other...> b);} \\
\};}
\]

The class template simd_flags acts like a integer bit-flag for types.

Constraints: Every type in Flags is a valid template argument to one of simd_flag_convert, simd_flag_aligned, simd_flag_overaligned, or to one of the implementation-defined load and store flags.

(6.1.5.2) 28.9.5.2 simd_flags operators

\[
\text{template <class... Other>} \\
\text{friend constexpr auto operator|(simd_flags a, simd_flags<Other...> b);} \\
\]

Returns: A specialization of simd_flags identifying all load and store flags identified either by a or b.

(6.1.6) 28.9.6 Class template basic_simd

(6.1.6.1) 28.9.6.1 Class template basic_simd overview

\[
\text{template<class T, class Abi> class basic_simd { \\
\quad \text{public:} \\
\quad \text{using value_type = T;} \\
\quad \text{using reference = see below;} \\
\quad \text{using mask_type = basic_simd_mask<sizeof(T), Abi>;} \\
\quad \text{using ab1_type = Abi;} \\
\quad \text{static constexpr integral_constant<simd-size-type, see below> size;}
\}}
\]
constexpr basic_simd() noexcept = default;

// 28.9.6.4, basic_simd constructors
template<class U> constexpr basic_simd(U&& value) noexcept;
template<class U, class UAbi>
    constexpr explicit(see below) basic_simd(const basic_simd<U, UAbi>&) noexcept;
template<class G> constexpr explicit basic_simd(G&& gen) noexcept;
template<typename It, class... Flags>
    constexpr basic_simd(It first, simd_flags<Flags...> = {});
template<typename It, class... Flags>
    constexpr basic_simd(It first, const mask_type & mask, simd_flags<Flags...> = {});

// 28.9.6.5, basic_simd copy functions
template<typename It, class... Flags>
    constexpr void copy_from(It first, simd_flags<Flags...> f = {});
template<typename It, class... Flags>
    constexpr void copy_from(It first, const mask_type & mask, simd_flags<Flags...> f = {});
template<typename Out, class... Flags>
    constexpr void copy_to(Out first, simd_flags<Flags...> f = {});
template<typename Out, class... Flags>
    constexpr void copy_to(Out first, const mask_type & mask, simd_flags<Flags...> f = {});

// 28.9.6.6, basic_simd subscript operators
constexpr reference operator[](simd-size-type) &;
constexpr value_type operator[](simd-size-type) const&;

// 28.9.6.7, basic_simd unary operators
constexpr basic_simd& operator++() noexcept;
constexpr basic_simd operator++(int) noexcept;
constexpr basic_simd& operator--() noexcept;
constexpr basic_simd operator--(int) noexcept;
constexpr mask_type operator!() const noexcept;
constexpr basic_simd operator~() const noexcept;
constexpr basic_simd operator+() const noexcept;
constexpr basic_simd operator-() const noexcept;

// 28.9.7.1, basic_simd binary operators
friend constexpr basic_simd operator+(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator-(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator*(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator/(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator%(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator&(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator|(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator^(const basic_simd&, const basic_simd&) noexcept;
friend constexpr basic_simd operator<<(const basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd operator>>(const basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd operator<<(const basic_simd &, simd-size-type) noexcept;
friend constexpr basic_simd operator>>(const basic_simd &, simd-size-type) noexcept;

// 28.9.7.2, basic_simd compound assignment
friend constexpr basic_simd & operator+=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator-=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator*=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator/=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator%=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator&=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator|=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator^=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator<<=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator>>=( basic_simd &, const basic_simd &) noexcept;
friend constexpr basic_simd & operator<<=( basic_simd &, simd-size-type) noexcept;
friend constexpr basic_simd & operator>>=( basic_simd &, simd-size-type) noexcept;

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

// 28.9.7.4, basic_simd conditional operators
friend constexpr basic_simd simd-select-impl(const mask_type &, const basic_simd &, const basic_simd &) noexcept;
};

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

2 Every specialization of basic_simd is a complete type. The specialization basic_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.

It is implementation-defined whether basic_simd<T, Abi> is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. —end note]

If basic_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<basic_simd<T, Abi>>, and
- is_nothrow_move_assignable_v<basic_simd<T, Abi>>, and
- is_nothrow_default_constructible_v<basic_simd<T, Abi>>.
[Example: Consider an implementation that defines the ABI tags __simd_x and __gpu_y. When the compiler is invoked to translate 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:

- basic_simd<T, simd_abi::__gpu_y> is not supported for any T and has a deleted constructor.
- basic_simd<long double, simd_abi::__simd_x> is not supported and has a deleted constructor.
- basic_simd<double, simd_abi::__simd_x> is supported.
- basic_simd<long double, simd_abi::scalar> is supported.

— end example ]

Default initialization 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]

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

constexpr explicit operator implementation-defined() const;
constexpr explicit basic_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:

```cpp
using V = basic_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 basic_simd width

```cpp
static constexpr integral_constant<simd-size-type, see below> size;
```

1 size is an integral_constant<@simd-size-type@, N> with N equal to the number of elements in a basic_simd object.

[Note: This member is present even if the particular basic_simd specialization is not supported. — end note]

(6.1.6.3) 28.9.6.3 Element references

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

Class reference is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name.

```cpp
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;

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

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

Constraints: declval<typename>() = std::forward<U>(x) is well-formed.

Effects: Replaces the referred to element in basic_simd or basic_simd_mask with static_cast<typename>(std::forward<U>(x)).

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

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

Returns: A copy of *this.

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

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

Effects: Applies the indicated operator to the referred to element in basic_simd or basic_simd_mask.

Returns: A copy of *this.

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

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

Effects: Applies the indicated operator to the referred to element in basic_simd or basic_simd_mask.

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;

Effects: Exchanges the values a and b refer to.

(6.1.6.4) 28.9.6.4 basic_simd constructors

template<class U> constexpr basic_simd(U&&) noexcept;

Let From denote the type remove_cvref_t<U>.

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.

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) basic_simd(const basic_simd<U, UAbi>& x) noexcept;

Constraints: basic_simd<U, UAbi>::size == size().

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

Remarks: The constructor is explicit
• if the conversion from \(U\) to \(\text{value\_type}\) is not value-preserving, or
• if both \(U\) and \(\text{value\_type}\) are integral types and the integer conversion rank ([conv.rank]) of \(U\) is greater than the integer conversion rank of \(\text{value\_type}\), or
• if both \(U\) and \(\text{value\_type}\) are floating-point types and the floating-point conversion rank ([conv.rank]) of \(U\) is greater than the floating-point conversion rank of \(\text{value\_type}\).

\[
\text{template<class G> constexpr basic\_simd(G&& \text{gen}) noexcept ;}
\]

Constraints:
\(\text{basic\_simd}([\text{gen}]\text{(integral\_constant\_size\_t, i}()\text{)})\) is well-formed for all \(i\) in the range of \([0, \text{size}())\).

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

The calls to \(\text{gen}\) are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by \(\text{gen}\) ([algorithms.parallel.exec]). \(\text{gen}\) is invoked exactly once for each \(i\).

\[
\text{template<typername It, class... Flags> constexpr basic\_simd(It \text{first}, simd\_flags<Flags...> = {});} 
\]

Constraints:
• \(\text{iter\_value\_t<It}\) is a vectorizable type, and
• \(\text{It}\) satisfies \text{contiguous\_iterator}.

Mandates: If the template parameter pack \(\text{Flags}\) does not contain the type identifying \text{simd\_flag\_convert}, then the conversion from \(\text{iter\_value\_t<It}\) to \(\text{value\_type}\) is value-preserving.

Preconditions:
• \([\text{first, first} + \text{size}())\) is a valid range.
• \(\text{It}\) models \text{contiguous\_iterator}.
• If the template parameter pack \(\text{Flags}\) contains the type identifying \text{simd\_flag\_aligned}, \text{addressof}(*\text{first}) shall point to storage aligned by \text{simd\_alignment\_v<basic\_simd, iter\_value\_t<It>}}.
• If the template parameter pack \(\text{Flags}\) contains the type identifying \text{simd\_flag\_overaligned<N>}, \text{addressof}(*\text{first}) shall point to storage aligned by \(N\).

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

Throws: Nothing.

\[
\text{template<typername It, class... Flags> constexpr basic\_simd(It \text{first}, const mask\_type\& mask, simd\_flags<Flags...> = {});} 
\]

Constraints:
• \(\text{iter\_value\_t<It}\) is a vectorizable type, and
• \(\text{It}\) satisfies \text{contiguous\_iterator}.

Mandates: If the template parameter pack \(\text{Flags}\) does not contain the type identifying \text{simd\_flag\_convert}, then the conversion from \(\text{iter\_value\_t<It}\) to \(\text{value\_type}\) is value-preserving.

Preconditions:
For all selected indices $i$, $[\text{first}, \text{first} + i)$ is a valid range.

It models contiguous_iterator.

If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd, iter_value_t<It>>.

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

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

**Throws:** Nothing.

---

(6.1.6.5) 28.9.6.5 basic_simd copy functions

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

**Constraints:**

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

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

**Preconditions:**

- $[\text{first}, \text{first} + \text{size()})$ is a valid range.
- It models contiguous_iterator.

- If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd, iter_value_t<It>>.

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

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

**Throws:** Nothing.

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

**Constraints:**

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

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

**Preconditions:**

- For all selected indices $i$, $[\text{first}, \text{first} + i)$ is a valid range.
It models contiguous_iterator.

If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd, iter_value_t<It>>.

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

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

**Throws:** Nothing.

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

**Constraints:**

- iter_value_t<Out> is a vectorizable type, and
- Out satisfies contiguous_iterator, and
- Out satisfies indirectly_writable<value_type>.

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

**Preconditions:**

- [first, first + size()) is a valid range.
- Out models contiguous_iterator.
- Out models indirectly_writable<value_type>.
- If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd, iter_value_t<Out>>.
- If the template parameter pack Flags contains the type identifying simd_flag_overaligned<N>, addressof(*first) shall point to storage aligned by N.

**Effects:** Copies all basic_simd elements as if first\[i\] = static_cast<iter_value_t<Out>>((operator[]\(i\)(first\[i\]))) for all \(i\) in the range of \([0, \text{size}())\).

**Throws:** Nothing.

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

**Constraints:**

- iter_value_t<Out> is a vectorizable type, and
- Out satisfies contiguous_iterator, and
- Out satisfies indirectly_writable<value_type>.

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

**Preconditions:**

- For all selected indices \(i\), [first, first + \(i\)) is a valid range.
• Out models contiguous_iterator.
• Out models indirectly_writable<value_type>.
• If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd, iter_value_t<Out>>.
• If the template parameter pack Flags contains the type identifying simd_flag_overaligned<N>, addressof(*first) shall point to storage aligned by N.


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

Throws: Nothing.

### 28.9.6.6 basic_simd subscript operators

```cpp
constexpr reference operator[](simd-size-type i) &;  
Preconditions: i < size().
Returns: A reference (see 28.9.6.3) referring to the ith element.
Throws: Nothing.
```

```cpp
constexpr value_type operator[](simd-size-type i) const&;  
Preconditions: i < size().
Returns: The value of the ith element.
Throws: Nothing.
```

### 28.9.6.7 basic_simd unary operators

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

```cpp
constexpr basic_simd& operator++() noexcept;  
Constraints: Application of unary ++ to objects of type value_type is well-formed.
Effects: Increments every element by one.
Returns: *this.
```

```cpp
constexpr basic_simd operator++(int) noexcept;  
Constraints: Application of unary ++ to objects of type value_type is well-formed.
Effects: Increments every element by one.
Returns: A copy of *this before incrementing.
```

```cpp
constexpr basic_simd& operator--() noexcept;
```
Constraints: Application of unary -- to objects of type value_type is well-formed.
Effects: Decrements every element by one.
Returns: *this.

constexpr basic_simd operator--(int) noexcept;

Constraints: Application of unary -- to objects of type value_type is well-formed.
Effects: Decrements every element by one.
Returns: A copy of *this before decrementing.

constexpr mask_type operator!() const noexcept;

Constraints: Application of unary ! to objects of type value_type is well-formed.
Returns: A basic_simd_mask object with the i-th element set to !operator[](i) for all i in the range of [0, size()).

constexpr basic_simd operator~() const noexcept;

Constraints: Application of unary ~ to objects of type value_type is well-formed.
Returns: A basic_simd object where each bit is the inverse of the corresponding bit in *this.

constexpr basic_simd operator+(() const noexcept;

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

constexpr basic_simd operator-() const noexcept;

Constraints: Application of unary - to objects of type value_type is well-formed.
Returns: A basic_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 basic_simd non-member operations
[simd.nonmembers]

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

friend constexpr basic_simd operator+(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator-(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator*(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator/(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator%(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator&(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator|(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator^(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator<<(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr basic_simd operator>>(const basic_simd & lhs, const basic_simd & rhs) noexcept;
Constraints: Application of the indicated operator to objects of type `value_type` is well-formed.

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

```
friend constexpr basic_simd operator<<(const basic_simd & v, simd-size-type n) noexcept;
friend constexpr basic_simd operator>>(const basic_simd & v, simd-size-type n) noexcept;
```

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

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

(6.1.7.2) 28.9.7.2 basic_simd compound assignment

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

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

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

Returns: `lhs`.

```
friend constexpr basic_simd & operator<<(basic_simd & lhs, simd-size-type n) noexcept;
friend constexpr basic_simd & operator>>(basic_simd & lhs, simd-size-type n) noexcept;
```

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

Effects: Equivalent to: `return operator@=(lhs, basic_simd(n));`

(6.1.7.3) 28.9.7.3 basic_simd compare operators

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

(6.1.7.3) 28.9.7.3 basic_simd compare operators

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

**Returns:** A `basic_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 basic_simd conditional operators**

```
friend constexpr basic_simd simd-select-impl(const mask_type & mask, const basic_simd & a, const basic_simd & b) noexcept ;
```

**Returns:** A `basic_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 basic_simd reductions**

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

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

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

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

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

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

**Preconditions:** The results of `all_of(x == binary_op(basic_simd<T, A1>(identity_element), basic_simd<T, A1>(x))) and all_of(basic_simd<T, A1>(x) == binary_op(x, basic_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`.

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

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

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

**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 basic_simd<T, Abi>& x, const typename basic_simd<T, Abi>::mask_type & mask, multiplies<> binary_op) noexcept;

**Returns:** If none_of(x), returns 1. Otherwise, returns \(\text{GENERALIZED\_SUM}(\text{binary\_op}, x[i], \ldots)\) for all selected indices \(i\).

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

**Constraints:** \(\text{is\_integral\_v<T>}\) is true.

**Returns:** If none_of(mask), returns \(-T()\). Otherwise, returns \(\text{GENERALIZED\_SUM}(\text{binary\_op}, x[i], \ldots)\) for all selected indices \(i\).

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

**Constraints:** \(\text{is\_integral\_v<T>}\) is true.

**Returns:** If none_of(mask), returns \(T()\). Otherwise, returns \(\text{GENERALIZED\_SUM}(\text{binary\_op}, x[i], \ldots)\) for all selected indices \(i\).

template<class T, class Abi> constexpr T reduce_min(const basic_simd<T, Abi>& x) noexcept;

**Constraints:** \(T\) satisfies totally_ordered.

**Preconditions:** \(T\) models totally_ordered.

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

template<class T, class Abi> constexpr T reduce_min(const basic_simd<T, Abi>& x) noexcept;

**Constraints:** \(T\) satisfies totally_ordered.

**Preconditions:** \(T\) models totally_ordered.

**Returns:** If none_of(mask), returns \(\text{numeric\_limits<T>::max()}\). 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 reduce_max(const basic_simd<T, Abi>& x) noexcept;

**Constraints:** \(T\) satisfies totally_ordered.

**Preconditions:** \(T\) models totally_ordered.

**Returns:** The value of an element \(x[j]\) for which \(x[i] < x[j]\) is false for all \(i\) in the range of \([0, \text{size()}]\).
template<class T, class Abi>
constexpr T reduce_max(const basic_simd<T, Abi>&, const typename basic_simd<T, Abi>::mask_type &) noexcept;

Constraints: T satisfies totally_ordered.

Preconditions: T models totally_ordered.

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 basic_simd and basic_simd_mask creation

template<class V, class Abi>
constexpr auto simd_split(const basic_simd<typename V::value_type, Abi>& x) noexcept;

Constraints:

• V is a specialization of basic_simd.

• M is a specialization of basic_simd_mask.

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

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().

• 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 ≥ 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.

template<class T, class... Abis>
constexpr basic_simd<T, see below<T, (basic_simd<T, Abis>::size + ...)>>> simd_cat(const basic_simd<T, Abis>&... xs) noexcept;

Returns: A data-parallel object initialized with the concatenated values in the xs pack of data-parallel objects: The i^{th} basic_simd/basic_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.

(6.1.7.7) 28.9.7.7 Algorithms
constraints: T satisfies totally_ordered.

preconditions: T models totally_ordered.

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

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

constraints: T satisfies totally_ordered.

preconditions: T models totally_ordered.

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

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

constraints: T satisfies totally_ordered.

preconditions: T models totally_ordered.

returns: a pair initialized with

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

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

constraints: T satisfies totally_ordered.

preconditions: T models totally_ordered.

preconditions: No element in lo shall be greater than the corresponding element in hi.

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

template<class T, class U>
constexpr auto simd_select(bool c, const T& a, const U& b) -> remove_cvref_t<decltype(c ? a : b)>;

returns: as-if \(c ? a : b\).

template<size_t Bytes, class Abi, typename T, typename U>
constexpr auto simd_select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b);

returns: as-if simd-select-impl(c, a, b).
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 `basic_simd<T, Abi>`, where `is_floating_point_v<T>` is true, then:

- All arguments corresponding to `double` parameters shall be convertible to `basic_simd<T, Abi>`.
- All arguments corresponding to `double*` parameters shall be of type `basic_simd<T, Abi>*`.
- All arguments corresponding to parameters of integral type `U` shall be convertible to `simd<U, basic_simd<T, Abi>::size>`.
- All arguments corresponding to `U*`, where `U` is integral, shall be of type `simd<U, basic_simd<T, Abi>::size>*`.

If the corresponding return type is `double`, the return type of the additional overloads is `basic_simd<T, Abi>`. Otherwise, if the corresponding return type is `bool`, the return type of the additional overload is `basic_simd_mask<T, Abi>`. Otherwise, the return type is `simd<R, basic_simd<T, Abi>::size>`, with `R` denoting the corresponding return type.

It is unspecified whether a call to these overloads with arguments that are all convertible to `basic_simd<T, Abi>` but are not of type `basic_simd<T, Abi>` is well-formed.

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.

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]

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

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

```cpp
template< size_t Bytes, class Abi> class basic_simd_mask {
public:
    using value_type = bool;
    using reference = see below;
    using abi_type = Abi;

    static constexpr auto size = basic_simd<integer-from<Bytes>, Abi>::size;

    constexpr basic_simd_mask() noexcept = default;

    // 28.9.8.2, basic_simd_mask constructors
    constexpr explicit basic_simd_mask(value_type) noexcept;
    template< size_t UBytes, class UAbi>
```
constexpr explicit basic_simd_mask(const basic_simd_mask<UBytes, UAbi>&) noexcept;

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

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

template<typename It, class... Flags>
constexpr basic_simd_mask(It first, const basic_simd_mask<mask, simd_flags<Flags...> > = {});

// 28.9.8.3, basic_simd_mask copy functions
template<typename It, class... Flags>
constexpr void copy_from(It first, simd_flags<Flags...> = {});

template<typename It, class... Flags>
constexpr void copy_from(It first, const basic_simd_mask<mask, simd_flags<Flags...> > = {});

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

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

// 28.9.8.4, basic_simd_mask subscript operators
constexpr reference operator[](simd-size-type) &;
constexpr value_type operator[](simd-size-type) const&;

// 28.9.8.5, basic_simd_mask unary operators
constexpr basic_simd_mask operator!( ) const noexcept;
constexpr simd_type operator+() const noexcept;
constexpr simd_type operator-() const noexcept;
constexpr simd_type operator~() const noexcept;

// 28.9.8.6, basic_simd_mask conversion operators
template<class U, class A>
constexpr explicit(sizeof( U) != Bytes) operator basic_simd<U, A>() const noexcept;

// 28.9.9.1, basic_simd_mask binary operators
friend constexpr basic_simd_mask operator&&(const basic_simd_mask &, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask operator||(const basic_simd_mask &, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask operator&(const basic_simd_mask &, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask operator|(const basic_simd_mask &, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask operator^(const basic_simd_mask &, const basic_simd_mask&) noexcept;

// 28.9.9.2, basic_simd_mask compound assignment
friend constexpr basic_simd_mask & operator&=( basic_simd_mask &, const basic_simd_mask &) noexcept;
friend constexpr basic_simd_mask & operator|=( basic_simd_mask &, const basic_simd_mask &) noexcept;
friend constexpr basic_simd_mask & operator^=( basic_simd_mask &, const basic_simd_mask &) noexcept;

// 28.9.9.3, basic_simd_mask comparisions
friend constexpr basic_simd_mask operator==(const basic_simd_mask &, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask operator!=(const basic_simd_mask &, const basic_simd_mask&) noexcept;
6 Wording: Add Section 9 of N4808 with modifications

// 28.9.4. basic_simd_mask conditional operators
friend constexpr basic_simd_mask simd-select-impl(const basic_simd_mask&, const basic_simd_mask&, const basic_simd_mask&) noexcept;
friend constexpr basic_simd_mask simd-select-impl(const basic_simd_mask&, bool, bool) noexcept;
template <class T0, class T1>
  friend constexpr basic_simd<see below, Abi>
  simd-select-impl(const basic_simd_mask&, const T0&, const T1&) noexcept;
};

1 The class template basic_simd_mask is a data-parallel type with the element type bool. The width of a given basic_simd_mask specialization is a constant expression, determined by the template parameters. Specifically, basic_simd_mask<T, Abi>::size() == basic_simd<T, Abi>::size().

2 Every specialization of basic_simd_mask is a complete type. The specialization basic_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).

   It is implementation-defined whether basic_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 basic_simd_mask<Bytes, 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<basic_simd_mask<Bytes, Abi>>, and
   • is_nothrow_move_assignable_v<basic_simd_mask<Bytes, Abi>>, and
   • is_nothrow_default_constructible_v<basic_simd_mask<Bytes, Abi>>.

3 Default initialization performs no initialization 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 basic_simd_mask:
   constexpr explicit operator implementation-defined() const;
   constexpr explicit basic_simd_mask(const implementation-defined& init) const;

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

28.9.8.2 basic_simd_mask constructors [simd.mask.ctor]

constexpr explicit basic_simd_mask(value_type x) noexcept;

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

template<size_t UBytes, class UAbi>
constexpr explicit basic_simd_mask(const basic_simd_mask<UBytes, UAbi>& x) noexcept;

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

3 Effects: Constructs an object of type basic_simd_mask where the i\textsuperscript{th} element equals x[i] for all i in the range of [0, size()].
template<class G> constexpr explicit basic_simd_mask(G&& gen) noexcept;

Constraints: static_cast<bool>(gen(integral_constant<size_t, i>())) is well-formed for all $i$ in the range of $[0, \text{size()}]$.

Effects: Constructs an object where the $i$th element is initialized to $\text{gen(integral_constant<size_t, i>()})$. The calls to $\text{gen}$ are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by $\text{gen}(\{\text{algorithms.parallel.exec}\})$.

template<typename It, class... Flags>
constexpr basic_simd_mask(It first, simd_flags<Flags...> = {});

Constraints:
- $\text{iter_value_t<It>}$ is of type $\text{bool}$, and
- It satisfies $\text{contiguous_iterator}$.

Preconditions:
- $[\text{first}, \text{first} + \text{size()}]$ is a valid range.
- It models $\text{contiguous_iterator}$.
- If the template parameter pack $\text{Flags}$ contains the type identifying $\text{simd_flag_aligned}$, $\text{addressof(*first)}$ shall point to storage aligned by $\text{simd_alignment_v<basic_simd_mask>}$.
- If the template parameter pack $\text{Flags}$ contains the type identifying $\text{simd_flag_overaligned<N>}$, $\text{addressof(*first)}$ shall point to storage aligned by $N$.

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

Throws: Nothing.

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

Constraints:
- $\text{iter_value_t<It>}$ is of type $\text{bool}$, and
- It satisfies $\text{contiguous_iterator}$.

Preconditions:
- For all selected indices $i$, $[\text{first}, \text{first} + i]$ is a valid range.
- It models $\text{contiguous_iterator}$.
- If the template parameter pack $\text{Flags}$ contains the type identifying $\text{simd_flag_aligned}$, $\text{addressof(*first)}$ shall point to storage aligned by $\text{simd_alignment_v<basic_simd_mask>}$.
- If the template parameter pack $\text{Flags}$ contains the type identifying $\text{simd_flag_overaligned<N>}$, $\text{addressof(*first)}$ shall point to storage aligned by $N$.

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

Throws: Nothing.
28.9.8.3 basic_simd_mask copy functions

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

Constraints:
- iter_value_t<It> is of type bool, and
- It satisfies contiguous_iterator.

Preconditions:
- [first, first + size()) is a valid range.
- It models contiguous_iterator.
- If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd_mask>.
- If the template parameter pack Flags contains the type identifying simd_flag_overaligned<N>, addressof(*first) shall point to storage aligned by N.

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

Throws: Nothing.

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

Constraints:
- iter_value_t<It> is of type bool, and
- It satisfies contiguous_iterator.

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

Effects: Replaces the selected elements of the basic_simd_mask object such that the \( i \)\textsuperscript{th} element is replaced with first\([i]\) for all selected indices \( i \).

Throws: Nothing.

template<
  typename Out,
  class... Flags>
constexpr void copy_to (Out first, simd_flags<Flags...> = {} const;
• Out satisfies contiguous_iterator, and
• Out satisfies indirectly_writable<value_type>.

**Preconditions:**
• \([\text{first, first + size()})\) is a valid range.
• Out models contiguous_iterator.
• Out models indirectly_writable<value_type>.
• If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd_mask>.
• If the template parameter pack Flags contains the type identifying simd_flag_overaligned<N>, addressof(*first) shall point to storage aligned by N.

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

**Throws:** Nothing.

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

**Constraints:**
• iter_value_t<Out> is of type bool, and
• Out satisfies contiguous_iterator, and
• Out satisfies indirectly_writable<value_type>.

**Preconditions:**
• For all selected indices \(i\), \([\text{first, first + i})\) is a valid range.
• Out models contiguous_iterator.
• Out models indirectly_writable<value_type>.
• If the template parameter pack Flags contains the type identifying simd_flag_aligned, addressof(*first) shall point to storage aligned by simd_alignment_v<basic_simd_mask>.
• If the template parameter pack Flags contains the type identifying simd_flag_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.

(6.1.8.4) 28.9.8.4 basic_simd_mask subscript operators

```cpp
constexpr reference operator[](simd-size-type i) &;
```

**Preconditions:** \(i < \text{size()}\).

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

**Throws:** Nothing.
constexpr value_type operator[](simd-size-type i) const &;

Preconditions: i < size().

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

Throws: Nothing.

(6.1.8.5) 28.9.8.5 basic_simd_mask unary operators [simd.mask.unary]

cconstexpr basic_simd_mask operator!() const noexcept;

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

cconstexpr simd_type operator+() const noexcept;
cconstexpr simd_type operator-() const noexcept;
cconstexpr 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).

(6.1.8.6) 28.9.8.6 basic_simd_mask conversion operators [simd.mask.conv]
template <class U, class A>
cconstexpr explicit(sizeof(U) != Bytes) operator basic_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 basic_simd_mask binary operators [simd.mask.binary]

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

Returns: A basic_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 basic_simd_mask compound assignment [simd.mask.cassign]
friend constexpr basic_simd_mask & operator&=(basic_simd_mask& lhs, const basic_simd_mask& rhs) noexcept;
friend constexpr basic_simd_mask & operator|=(basic_simd_mask& lhs, const basic_simd_mask& rhs) noexcept;
friend constexpr basic_simd_mask & operator^=(basic_simd_mask& lhs, const basic_simd_mask& rhs) noexcept;

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

**Returns:** lhs.

### 28.9.9.3 basic_simd_mask comparisons

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

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

### 28.9.9.4 basic_simd_mask conditional operators

friend constexpr basic_simd simd-select-impl(const basic_simd_mask& mask, const basic_simd_mask& a, const basic_simd_mask& b) noexcept;
friend constexpr basic_simd simd-select-impl(const basic_simd_mask& mask, bool a, bool b) noexcept;

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

### 28.9.9.5 basic_simd_mask reductions

template <class T0, class T1>
friend constexpr basic_simd<see below, Abi> simd-select-impl(const basic_simd_mask& mask, const T0& a, const T1& b) noexcept;

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.

**Constraints:**
- U is a vectorizable type, and
- sizeof(U) == sizeof(T), and
- T0 satisfies convertible_to<basic_simd<U, Abi>>, and
- T1 satisfies convertible_to<basic_simd<U, Abi>>.

**Returns:** A basic_simd<U, Abi> object where the i\(^{th}\) element equals mask[i] ? a[i] : b for all i in the range of [0, size()).
template< size_t Bytes, class Abi> constexpr bool all_of (const basic_simd_mask<Bytes, Abi>& k) noexcept ;

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

template< size_t Bytes, class Abi> constexpr bool any_of (const basic_simd_mask<Bytes, Abi>& k) noexcept ;

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

template< size_t Bytes, class Abi> constexpr bool none_of (const basic_simd_mask<Bytes, Abi>& k) noexcept ;

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

template< size_t Bytes, class Abi> constexpr bool all_of (T) noexcept ;
constexpr bool any_of (T) noexcept ;
constexpr bool none_of (T) noexcept ;
constexpr simd-size-type reduce_count(T) noexcept ;

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.

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

constexpr simd-size-type reduce_min_index(T);
constexpr simd-size-type reduce_max_index(T);

Preconditions: The value of the argument is true.

Returns: 0.

Throws: Nothing.

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

ACKNOWLEDGMENTS

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

B

BIBLIOGRAPHY


