ABSTRACT

After the Parallelism TS 2 was published in 2018, data-parallel types (basic_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.
4 Changes after TS feedback

4.1 improve ABI tags .............................................. 16
4.2 basic_simd_mask<sizeof(Ab), ABI> ........................ 16
4.3 Simplify/generalize casts ................................. 17
4.4 Add simd_mask generator constructor ................... 17
4.5 Default load/store flags to element_aligned .......... 17
4.6 Contiguous iterators for loads and stores ............... 18
4.7 constexpr everything ......................................... 18
4.8 Specify simd::size as integral_constant ................. 18
4.9 Replace where facilities ...................................... 18
4.10 Make use of int and size_t consistent ..................... 19
4.11 Add lvalue-qualifier to non-const subscript .......... 20
4.12 Rename simd_mask reductions ............................... 20
4.13 Rename hmin and hmax ....................................... 20
4.14 Added constraints on operators and functions to match their underlying element types ........ 21
4.15 Rename alignment flags and extend load/store flags for opt-in to conversions ..................... 21
4.16 Reduce overloads and rename split and concat .......... 21
4.17 Remove int exception from broadcast conversion rules .................................................. 22
4.18 Remove long double from vectorizable types ............. 22
4.19 Increase minimum supported width to 64 ................. 22
4.20 No std::hash<simd> .......................................... 23
4.21 No freestanding SIMD ......................................... 23

5 Outlook

5.1 element_reference is overspecified ....................... 23
5.2 Clean up math function overloads ......................... 24
5.3 Integration with ranges ....................................... 24
5.4 Formatting support .......................................... 24

6 Wording: Add Section 9 of N4808 with modifications .... 24

28.10 Data-parallel types [simd] (6.1.0) ...................... 25

A Acknowledgments .............................................. 55
B Bibliography .................................................. 56
1.1 changes from revision 0

Previous revision: P19280R0

- 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: P1928OR1

- 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: P1928OR2

- Propose alternative to `hmin` and `hmax`.
- Discuss `simd_mask` reductions wrt. consistency with `<bit>`. Propose better names to avoid ambiguity.
- Remove `some_of`.
- Add unary `-` to `simd_mask`.
- Discuss and ask for confirmation of masked "overloads" names and argument order.
- Resolve inconsistencies wrt. `int` and `size_t`: Change `fixed_size` and `resize_simd` NTPs from `int` to `size_t` (for consistency).
- Discuss conversions on loads and stores.
• Point to [P2509R0] as related paper.
• Generalize load and store from pointer to contiguous_iterator. (Section 4.6)
• Moved "element_reference is overspecified" to "Open questions".

1.4 changes from revision 3

Previous revision: P19280R3

• Remove wording diff.
• Add std::simd to the paper title.
• Update ranges integration discussion and mention formatting support via ranges (Section 5.4).
• 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 ([simd.unary], [simd.binary]).
• Rename mask reductions as decided in Issaquah.
• Remove R3 ABI discussion and add follow-up question.
• Add open question on first template parameter of simd_mask (Section 4.2).
• Overload loads and stores with mask argument ([simd ctor], [simd copy], [simd mask ctor], [simd mask copy]).
• Respecify basic_simd reductions to use a basic_simd_mask argument instead of const_ where_expression ([simd reductions]).
• Add basic_simd_mask operators returning a basic_simd ([simd mask unary], [simd mask conv])
• Add conditional operator overloads as hidden friends to basic_simd and basic_simd_mask ([simd cond], [simd mask cond]).
• Discuss std::hash for basic_simd (Section 4.20).
• Constrain some functions (e.g., min, max, clamp) to be totally_ordered ([simd reductions], [simd alg]).
• Asking for reconsideration of conversion rules.
• Rename load/store flags (Section 4.15).
• Extend load/store flags with a new flag for conversions on load/store. (Section 4.15).
• Update hmin/hmax discussion with more extensive naming discussion (Section 4.13).
• Discuss freestanding basic_simd (Section 4.21).
• Discuss split and concat (Section 4.16).
• Apply the new library specification style from P0788R3.

1.5

CHANGES FROM REVISION 4

Previous revision: P19280R4

• Added simd_select discussion.

1.6

CHANGES FROM REVISION 5

Previous revision: P19280R5

• Updated the wording for changes discussed in and requested by LEWG in Varna.
• Rename to simd_cat and simd_split.
• Drop simd_cat(array) overload.
• Replace simd_split by simd_split as proposed in P1928R4.
• Use indirectly_writable instead of output_iterator.
• Replace most size_t and int uses by simd-size-type signed integer type.
• Remove everything in 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 int/unsigned int exception from conversion rules of broadcast ctor.
• Rename loadstore_flags to simd_flags.
• Make simd_flags::operator| constexpr.
• Remove simd_flags::operator& and simd_flags::operator\&.
• Increase minimum SIMD width to 64.
• Rename hmin/hmax to 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.

1.7 CHANGES FROM REVISION 6

Previous revision: P1928R6

• Remove mask reduction precondition but ask LEWG for reversal of that decision (Section ??).
• Fix return type of basic_simd_mask unary operators.
• Fix bool overload of simd-select-impl(Section ??).
• Remove unnecessary implementation freedom in simd_split (Section ??).
• Use class instead of typename in template heads.
• Implement LEWG decision to SFINAE on values of constexpr-wrapper-like arguments to the broadcast ctor ([simd_ctor]).
• Add relational operators to basic_simd_mask as directed by LEWG ([simd.mask.comparison]).
• Update section on size_t vs. int usage (Section 4.10).
• Remove all open design questions, leaving LWG / wording questions.
• Add LWG question on implementation note (Section ??).
• Add constraint for BinaryOperation to reduce overloads ([simd.reductions]).
1.8 Changes from revision 7

Previous revision: P1928R7

- Include std::optional return value from reduce_min_index and reduce_max_index in the exploration.
- Fix \LaTeX markup errors.
- Remove repetitive mention of "exposition-only" before deduce-t.
- Replace "TU" with "translation unit".
- Reorder first paragraphs in the wording, especially reducing the note on compiling down to SIMD instructions.
- Replace cv-unqualified arithmetic types with a more precise list of types.
- Move the place where "supported" is defined.

1.9 Changes from revision 8

Previous revision: P1928R8

- Improve wording that includes the C++23 extended floating-point types in the set of vectorizable types ([simd.general] p.2).
- Improve wording that defines "selected indices" and "selected elements" ([simd.general] p.6).
- Remove superfluous introduction paragraph.
- Improve wording introducing the intent of ABI tags ([simd.abi] p.1)
- Consistently use size as a callable in the wording.
- Add missing type_identity_t for reduce ([simd.syn], [simd.reductions]).
- Spell out "iff" ([simd.abi] p.4).
- Fixed template argument to native-abi in the default template argument of basic_simd_-mask ([simd.syn]).
- Fixed default template argument to simd_mask to be consistent with simd ([simd.syn]).
- Add instructions to add <simd> to the table of headers in [headers].
- Add instructions to add a new subclause to the table in [numerics.general].
• Add instructions to add `<simd>` [diff.23.library].

• Add `simd-size-v` to the wording and replace `simd_size_v` to actually implement "Make `simd-size` exposition-only."

• Restored precondition (and removed `noexcept`) on `reduce_min_index` and `reduce_max_index` as directed by LEWG.

### 1.10 Changes from revision 9

Previous revision: P1928OR9

• Strike through wording removed by P3275 (non-const operator[]).

• Remove "exposition only" from detailed prose, it’s already marked as such in the synopsis.

• Reorder definition of `vectorizable type` above its first use.

• Commas, de-duplication, word order, s/may/can/ in a note.

• Use text font for ""]" when defining a range of integers.

• Several small changes from LWG review on 2024-06-26.

• Reword `rebind simd` and `resize simd`.

• Remove mention of implementation-defined load/store flags.

• Remove paragraph about default initialization of `basic_simm`.

• Reword all constructor `Effects` from "Constructs an object ..." to "Initializes ...".

• Instead of writing "satisfies X" in `Constraints` and "models X" in `Preconditions`, say only "models X" in `Constraints`.

• Replace `is_trivial_v` with "is trivially copyable".

• First shot at improving generator function constraints.

• Reword constraints on unary and binary operators.

• Add missing/inconsistent `explicit` on load constructors.

• Fix preconditions of subscript operators.

• Reword effects of compound assignment operators.
2 Straw Polls

- Add that BinaryOperation may not modify input basic_simd.
- Fix definition of GENERALIZED_SUMs.

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></td>
<td>10</td>
<td>8</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></td>
<td>9</td>
<td>8</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></td>
<td>16</td>
<td>12</td>
<td>0</td>
<td>0</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></td>
<td>16</td>
<td>7</td>
<td>1</td>
<td>0</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></td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>1</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_if(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>9</td>
<td>2</td>
<td></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>

2.3 LEWG AT VARNA 2023

The conditional operator CPO should be called: (vote for as many options as you like)

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditional_operator</td>
<td>10</td>
</tr>
<tr>
<td>ternary</td>
<td>12</td>
</tr>
<tr>
<td>inline_if</td>
<td>0</td>
</tr>
<tr>
<td>iif</td>
<td>1</td>
</tr>
<tr>
<td>blend</td>
<td>2</td>
</tr>
<tr>
<td>select</td>
<td>14</td>
</tr>
<tr>
<td>choose</td>
<td>4</td>
</tr>
</tbody>
</table>
The conditional operator $CPO$ should be called: (vote once for your favorite)

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{conditional	extunderscore operator}$</td>
<td>2</td>
</tr>
<tr>
<td>$\text{ternary}$</td>
<td>8</td>
</tr>
<tr>
<td>$\text{select}$</td>
<td>10</td>
</tr>
</tbody>
</table>

Poll: The conditional operator $CPO$ should be called $\text{ternary}$

<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>7</td>
<td>0</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: The conditional operator $CPO$ should be called $\text{select}$

<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>9</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: The conditional operator $CPO$ should be called $\text{conditional	extunderscore operator}$

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: The conditional operator facility should not be user customizable, should work both scalar and SIMD types and should be marketed as part of the SIMD library.

<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>8</td>
<td>9</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The conditional operator facility should be called (vote once for your favorite):

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{simd	extunderscore ternary}$</td>
<td>4</td>
</tr>
<tr>
<td>$\text{simd	extunderscore bland}$</td>
<td>6</td>
</tr>
<tr>
<td>$\text{simd	extunderscore select}$</td>
<td>12</td>
</tr>
<tr>
<td>$\text{simd	extunderscore choose}$</td>
<td>0</td>
</tr>
</tbody>
</table>

Tuesday afternoon polls missing in minutes and/or GitHub issue.

Poll: Don’t publicly expose $\text{simd	extunderscore abi}$ ($\text{deduce	extunderscore t}$, $\text{fixed	extunderscore size}$, $\text{scalar}$, $\text{native}$). Preserve ABI tagging semantics. Rename $\text{simd}$ to $\text{basic	extunderscore simd}$. Add a $\text{simd}$ alias: $\text{simd<T, size_t N = basic	extunderscore simd<T>::size() >}$

= $\text{basic	extunderscore simd<T, __deduce	extunderscore t<T, N>>}$

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Poll: Spell the flags template std::simd_flags and spell the individual flags std::simd_flag_x.

<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>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Make simd_mask<T, N> an alias for basic_simd_mask<sizeof(TT), __deduce_t<T, N>>.

<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>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Remove simd_mask<T, N>::simd_type and make simd_mask<T, N> unary plus and unary minus return simd<I, N> where I is the largest standard signed integer type where sizeof(I) <= sizeof(T).

<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>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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>
<tbody>
<tr>
<td>4</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Replace all split/split_by functions by the proposed split function in P1928R4.

<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>8</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</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>
<tbody>
<tr>
<td>5</td>
<td>11</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 built-in numeric types (status quo, option 3 in P1928R4).

<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>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Remove broadcast constructor exceptions for int and unsigned int, and instead ensure constexpr_v arguments work correctly (ext: 2 * simd<float> will no longer compile).

<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>6</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: The broadcast constructor should take T directly and rely on language implicit conversion rules and optionally enabled compiler warnings to catch errors (ex: 2 * simd<float> will return
simd<float>, 3.14 * simd<float> will return simd<float> and may warn.

<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>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Poll: Remove is_simd, is_simd_v, is_simd_mask, and is_simd_mask_v.

<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>9</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Make simd_size exposition only and cause simd to have the size static data member if and only if T is a vectorizable type and Abi is an ABI tag.

<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>7</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Replacement name for memory_alignment and memory_alignment_v should feature a simd- prefix.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: There should be a marker in the name of memory_alignment and memory_alignment_v indicating that it applies only to loads and stores.

<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>3</td>
<td>9</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The name of memory_alignment should be (with memory_alignment_v having the same name followed by _v)

<table>
<thead>
<tr>
<th>Option</th>
<th>Votes</th>
</tr>
</thead>
<tbody>
<tr>
<td>simd_memory_alignment</td>
<td>2</td>
</tr>
<tr>
<td>simd_alignment</td>
<td>13</td>
</tr>
<tr>
<td>simd_loadstore_alignment</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: We’re interested in exploring rebind_simd and resize_simd as members of simd and simd_mask

<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>0</td>
<td>8</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Introduce an exposition only simd-size-t signed integer type and use this type consistently throughout P1928 (rather than size_t and int being used inconsistently).

<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>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Several Thursday morning polls missing in minutes and/or GitHub issue.

Poll: Simd reduce should not have a binary operator

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Poll: Modify P1928D6 ("simd") as described above, and then send the revised paper to library for C++26, to be confirmed with a library evolution electronic poll.

<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>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.4 LEWG TELECON 2024-01-16

Poll: Restore the precondition on reduce_min_index(empty_mask) and reduce_max_index(empty_mask) (TS status quo, UB).

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Return an unspecified value on reduce_min_index(empty_mask) and reduce_max_index(empty_mask).

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Poll: Return std::optional from reduce_min_index and 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>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Poll: Modify P1928R8 (Merge data-parallel types from the Parallelism TS 2) by restoring the TS specification for reduce_min_index/reduce_max_index and adding the change to 16.4.2.3 to list the header, and then send the revised paper to LWG for C++26 to be confirmed with a Library Evolution electronic poll.

<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>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

INTRODUCTION

[P0214R9] introduced std::experimental::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...
3 Introduction

P0214 evolved) implementation existed for the whole time P0214 progressed through the committee. Shortly after the GCC 9 release, a complete implementation of Section 9 of the TS was made available. Since GCC 11 a complete simd implementation of the TS is part of its standard library.

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

3.1 Related Papers

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

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

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

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

Draft on non-member operator[ ] TODO

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

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

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

P2638 Intel’s response to P1915R0 for std::simd

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

P2664 Permutations for simd.
4 Changes after TS feedback

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

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

4 Changes after TS feedback

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

4.1 Improve ABI tags

Summary:

• Change the default SIMD ABI tag to simd_abi::native<T> instead of simd_abi::compatible<T>.

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

• At the Varna LEWG meeting it was decided to remove the simd_abi namespace and all standard ABI tags altogether. Rationale: The initial goal was to let fixed_size be equivalent to std::experimental::simd_abi::deduce_t. This implies that std::experimental::fixed_size_simd<T, N> becomes the generic interface for deducing an efficient ABI tag. The next logical step is to give fixed_size_simd a shorter name and hide ABI tags. Consequently, std::simd<T, N = native-size> is an alias for std::basic_simd<T, Abi> now.

For a discussion, see P1928R3 Section 4.1 and P1928R4 Section 5.2.

4.2 Basic simd_mask<sizeof, ABI>

Following the polls by LEWG in Issaquah 2023, P1928R4 made mask types interconvertible. The next simplification was to make interconvertible types the same type instead. This is achieved by renaming the std::experimental::simd_mask class template to std::basic_simd_mask and changing the first template parameter from element type T to sizeof(T). An alias simd_mask<T, N> = basic_simd_mask<sizeof(T), native-size> provides the simpler to use API.

The resulting mask types are explicitly convertible if the SIMD width is equal, otherwise they are not convertible at all. Note that for some target hardware the (explicitly) convertible masks are convertible without any cost. However, that’s not the case for all targets, which is why the conversion is still marked explicit.
4.3 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<T0, A0>` can be bit_casted to `simd<T1, A1>` if `sizeof(simd<T0, A0>) == sizeof(simd<T1, A1>). (This point is irrelevant if Section 4.2 is accepted.)

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

4.4 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.

4.5 Default load/store flags to element_aligned

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

4.6 Contiguous Iterators for Loads and Stores
Different to the TS, loads and stores use `contiguous_iterator` instead of pointers. For a discussion, see P1928R3 Section 4.5.

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

4.8 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 `basic simd` and `basic simd_mask`. For a discussion, see P1928R3 Section 4.7.

4.9 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 `basic simd` and `basic simd_mask` arguments.

The following operators were added to `basic simd_mask`:

- `basic simd_mask::operator basic simd<U, A>()` const noexcept
- `simd-type basic simd_mask::operator+()` const noexcept
4 Changes after TS feedback

- simd-type basic_simd_mask::operator-() const noexcept
- simd-type basic_simd_mask::operator~() const noexcept

The following hidden friends were added to basic_simd_mask:

- basic_simd_mask simd-select-impl(const basic_simd_mask&, const basic_simd_mask&, const basic_simd_mask&) noexcept
- basic_simd_mask simd-select-impl(const basic_simd_mask&, bool, bool) noexcept
- simd<non-promoting-common-type<T0, T1>> simd-select-impl(const basic_simd_mask&, const T0&, const T1&) noexcept

The following hidden friend was added to basic_simd:

- basic_simd simd-select-impl(const mask_type& mask, const basic_simd& a, const basic_simd& b) noexcept

Instead of simd-select-impl we would have preferred to overload operator?: but that requires a language change first. As long as we don't have the language feature for overloading ?:, generic code must use an inferior function instead. Knowing that other libraries would benefit from an overloadable operator?: P1928R4 proposed a std::conditional_operator CPO that 3rd-party libraries could have extended. However, the use of a function (or CPO) instead of overloading operator?: cannot 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].

Therefore LEWG decided in Varna to define a std::simd_select function instead of a general CPO, with the following goals:

- Analogue semantics to ?:, but without lazy evaluation.
- User's should not be able to extend the facility.
- Make it “value based”, i.e. don't bother about references for non-simd arguments.

4.10 Make use of int and size_t consistent

Different to the TS, this paper uses simd-size-type instead of size_t for

- the SIMD width (number of elements),
- the generator constructor call argument,
- the subscript operator arguments, and
• the `basic simd_mask` reductions that return an integral value.

Alignments and values identifying a `sizeof` still use `size_t`.

The type `simd-size-type` is an exposition-only alias for a signed integer type. i.e. the implementation is free to choose any signed integer type.

The rationale given in the LEWG discussion was a desire to avoid type conversions when using the result of a `basic simd_mask` reduction as subscript argument. Since `<bit>` functions like `std::popcount`, `std::bit_width`, `std::countl_zero`, ... return `int`, the natural choice is to stick with that type and make subscript arguments use the same type. Since the SIMD width is also sometimes used in expressions in the subscript argument, the SIMD width should also have the same type.

### 4.11 ADD LVALUE-QUALIFIER TO NON-CONST SUBSCRIPT

The operator[] overloads of `basic simd` and `basic 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). Therefore, both operator[] overloads are now lvalue-ref qualified to make `(x * 2)[0]` pick the const overload, which returns a prvalue that is not assignable.

### 4.12 RENAME SIMD_MASK REDUCTIONS

Summary:

• The function `std::experimental::some_of` was removed.

• The function `std::experimental::popcount` was renamed to `std::reduce_count`.

• The function `std::experimental::find_first_set` was renamed to `std::reduce_min_index`.

• The function `std::experimental::find_last_set` was renamed to `std::reduce_max_index`.

For a discussion of this topic see P1928R3 Section 5.2.

### 4.13 RENAME HMIN AND HMAX

The functions `hmin(simd)` and `hmax(simd)` were renamed to `reduce_min` and `reduce_max` according to guidance from LEWG in Varna 2023.
ADDED CONSTRAINTS ON OPERATORS AND FUNCTIONS TO MATCH THEIR UNDERLYING ELEMENT TYPES

4.14

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 `basic_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.

4.15 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. A new flag enables conversions that are not value-preserving on loads and stores. (Without this new flag, only value-preserving conversions are allowed.) The new flags facility also keeps the design space open for adding new flags after C++26. The changes relative to the TS are shown in Table 1.

<table>
<thead>
<tr>
<th>TS</th>
<th>P1928R10</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::experimental::element_aligned</code></td>
<td><code>std::simd_flag_default</code></td>
</tr>
<tr>
<td><code>std::experimental::vector_aligned</code></td>
<td><code>std::simd_flag_aligned</code></td>
</tr>
<tr>
<td><code>std::experimental::overaligned&lt;N&gt;</code></td>
<td><code>std::simd_flag_overaligned&lt;N&gt;</code></td>
</tr>
<tr>
<td><code>implicit</code></td>
<td><code>std::simd_flag_convert</code></td>
</tr>
</tbody>
</table>

Table 1: Load/store flag changes

Note that the wording also allows additional implementation-defined load and store flags. The trait `std::experimental::is_simd_flag_type` has been removed because the flag parameter is now constrained via the `simd_flags` 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>);
}
```

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

4.16 REDUCE OVERLOADS AND RENAME SPLIT AND CONCAT

The `std::experimental::concat(array)` overload was removed in favor of using `std::apply`. The remaining `std::experimental::concat` function was renamed to `std::simd_cat` following the `std::tuple_cat` naming precedent.
The two `std::experimental::split` and one `std::experimental::split_by` functions from the TS were consolidated into a single `std::simd_split` function. The design intent for the `simd_split` function is to support the use case of splitting an "oversized" `basic_simd` into register-sized parts. Example: `simd<float, 20>` could be made up of one AVX-512 and one SSE register on an x86 target. `simd_split` is a simple interface for splitting `simd<float, 20>` into `simd<float>` and `basic_simd<float, impl-defined-abi-tag>`.

`std::simd_split<T>(x)` does the following: `simd_split<simd<float>>(x)` returns a tuple of as many `simd<float>` as `x.size()` allows plus an "epilogue" of one `impl-defined-abi-tag` object as necessary to return all elements of `x`. If no "epilogue" is necessary, the return type is an array instead of a tuple. Then `simd_split<simd<float>>(simd<float, 20>)` returns

- `tuple<simd<float>>, simd<float>, 4>` with AVX-512,
- `tuple<simd<float>>, simd<float>, simd<float>, 4>` with AVX, and
- `array<simd<float>, 5>` with SSE.

The `simd_split` function is overloaded for `basic_simd` and `basic_simd_mask`.

### 4.17 Remove int exception from broadcast conversion rules

LEWG discussed conversions in Issaquah 2023 and Varna 2023. P1928R4 Section 5.4 presented alternatives and their implications. LEWG decided in Varna to stick with value-preserving conversions as used in the TS. However, the exception for `int` and `unsigned int` conversions to `simd` were removed. Instead, `integral_constant`-like arguments, which will hopefully be available as literals in C++26, will be supported and their values (instead of types) determine whether the conversion is allowed.

### 4.18 Remove long double from vectorizable types

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

### 4.19 Increase minimum supported width to 64

The TS required a minimum of 32, with C++26 the minimum will be 64.

Rationale: AVX-514 `simd<char>::size() == 64`. And also `long double` is not a vectorizable type anymore.

---

1 same as `simd<float, 4>`. 
4.20 **NO STD::HASH<SIMD>**

No support for std::hash<simd<T>> was added.

Rationale: Is there a use case for std::hash<simd<T>>? In other words, is there a use case for using basic_simd<T> as a map key? Recall that we do not consider basic_simd<T> to be a product type [P0851R0]. If there’s no use case for hashing a basic_simd<T> 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 we prefer to wait for concrete use cases of hashing basic_simd objects before providing any standard interface. Specifically, at this point we do not want std::hash support for basic_simd.

4.21 **NO FREESTANDING SIMD**

simd will not be enabled 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 basic_simd to integers and the scalar ABI for freestanding. The utility of such a crippled basic_simd is highly questionable. Note that freestanding is just the baseline requirement and embedded targets are still free to add simd support.

5 **OUTLOOK**

5.1 **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).

As an alternative [P3275R0] discusses removal of non-const subscripts altogether. This would imply removal of element_reference, simplifying the wording by a good chunk.

5.2 **CLEAN UP MATH FUNCTION OVERLOADS**
The wording that produces basic_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(basic_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.

5.3 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".

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.

5.4 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.

6 Wording: Add Section 9 of N4808 with Modifications

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

In [headers], add the header <simd> to [tab:headers.cpp].
In [numerics.general], add a new row to [tab:numeric.summary]:

| [simd] | Data-parallel types | <simd> |

In [diff.23.library], modify:

1 **Affected subclause:** [headers]

   **Change:** New headers.

   **Rationale:** New functionality.

   **Effect on original feature:** The following C++ headers are new: `<debugging>`, `<hazard_pointer>`, `<linalg>`, `<rcu>`, `<simd>`, and `<text_encoding>`. Valid C++ 2023 code that includes headers with these names may be invalid in this revision of C++.

In [version.syn], add the following and adjust the placeholder value as needed so as to denote this proposal's date of adoption:

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

At the end of [numerics] (after §28.9 [linalg]), add the following new subclause:

(6.1) 28.10 Data-parallel types

(6.1.1) 28.10.1 General

1 [simd] defines data-parallel types and operations on these types. [Note: The intent is to support acceleration through data-parallel execution resources where available, such as SIMD registers and instructions or execution units driven by a common instruction decoder. — end note]

2 The set of **vectorizable types** comprises all standard integer types, character types, and the types `float` and `double` ([basic.fundamental]). In addition, `std::float16_t`, `std::float32_t`, and `std::float64_t` are vectorizable types if defined ([basic.extended.fp]).

3 The term **data-parallel type** refers to all supported ([simd.overview]) specializations of the `basic_simd` and `basic_simd_mask` class templates. A data-parallel object is an object of data-parallel type.

4 A data-parallel type consists of one or more 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. The elements in a data-parallel type are indexed from 0 to `width - 1`.

5 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.
6 Given a basic_simd_mask<Bytes, Abi> object mask, the selected indices signify the integers \(i\) in the range \([0, mask.size())\) for which \(mask[i]\) is true. Given an 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 \(i\).

7 The conversion from vectorizable type \(U\) to vectorizable type \(T\) is value-preserving if all possible values of \(U\) can be represented with type \(T\).

(6.1.2) 28.10.2 Header <simd> synopsis

```cpp
namespace std {
  using simd-size-type = see below; // exposition only
  template <class T, class Abi> constexpr simd-size-type simd-size-v = see below; // exposition only

  template <class T> constexpr size_t mask-element-size = see below; // exposition only
  template <size_t Bytes> using integer-from = see below; // exposition only

  template <class T>
  concept constexpr-wrapper-like =
      // exposition only
      convertible_to<T, decltype(T::value)> &&
      equality_comparable_with<T, decltype(T::value)> &&
      bool_constant<T>() == T::value::value &&
      bool_constant<static_cast<decltype(T::value)>(T()) == T::value::value; // exposition only

  // [simd.abi], 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

  // [simd.traits], 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;

  // [simd.flags], 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<simd-size-type N> requires (has_single_bit(N))
  inline constexpr simd_flags<> simd_flag_overaligned{};

  // [simd.class], 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>>;

// [simd.mask.class], Class template basic_simd_mask
template<
size_t Bytes, class Abi = native-abi<integer-from<Bytes>>>
class basic_simd_mask;
template<class T, simd-size-type N = basic_simd<T>::size()>
using simd_mask = basic_simd_mask<sizeof(T), deduce-t<T, N>>;

// [simd.creation], 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 Bs, class... Abis>
constexpr basic_simd_mask<Bs, deduce-t<integer-from<Bs>>, (basic_simd_mask<Bs, Abis>::size() + ...)>>
simd_cat(const basic_simd_mask<Bs, Abis>&... ) noexcept;

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

constexpr bool all_of(same_as<bool> auto) noexcept;
constexpr bool any_of(same_as<bool> auto) noexcept;
constexpr bool none_of(same_as<bool> auto) noexcept;
constexpr simd-size-type reduce_count(same_as<bool> auto) noexcept;
constexpr simd-size-type reduce_min_index(same_as<bool> auto);
constexpr simd-size-type reduce_max_index(same_as<bool> auto);

// [simd.reductions], basic_simd reductions
template<class T, class Abi, class BinaryOperation = plus>
constexpr T reduce(const basic_simd<T, Abi>&, BinaryOperation = {});

```cpp
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,
                   type_identity_t<T> identity_element,
                   BinaryOperation binary_op);
```

```cpp
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;
```

```cpp
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 ;
```

```cpp
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 ;
```

```cpp
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 ;
```

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

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

```cpp
template<class T, class Abi>
constexpr T reduce_min (const basic_simd<T, Abi>&,
                        const typename basic_simd<T, Abi>::mask_type &) noexcept ;
```

```cpp
template<class T, class Abi>
constexpr T reduce_max (const basic_simd<T, Abi>&) noexcept ;
```

```cpp
template<class T, class Abi>
constexpr T reduce_max (const basic_simd<T, Abi>&,
                        const typename basic_simd<T, Abi>::mask_type &) noexcept ;
```

```cpp
// [simd.alg]. Algorithms
```

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

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

```cpp
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 ;
```

```cpp
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, class T, class U>  
constexpr auto simd_select (const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)  
noexcept -> decltype(simd-select-impl(c, a, b));

1 **simd-size-type** is an alias for a signed integer type.

2 **simd-size-v<T, Abi>** denotes the width of basic_simd<T, Abi>. [ Note: **simd-size-v<T, Abi>** does not require instantiation of basic_simd<T, Abi>. — end note ]

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) 28.10.3 simd ABI tags  
[ simd.abi ]

    template<class T> using native-abi = see below; // exposition only
    template<class T, simd-size-type 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 and compiler flags. — end note ] The ABI tag, together with a given element type, implies the width.

2 [ Note: The ABI tag is orthogonal to selecting the machine instruction set. The selected machine instruction set limits the usable ABI tag types, though (see [simd.overview]). 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 ]

An implementation defines ABI tag types as necessary for the following aliases.

4 **deduce-t<T, N>** is defined if

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

The implementation-defined maximum for N is not smaller than 64 and can differ depending on T. Where present, **deduce-t<T, N>** names an ABI tag type that satisfies

- simd-size-v<T, deduce-t<T, N>> == N, and
- basic_simd<T, deduce-t<T, N>> is supported (see [simd.overview]).

**Note:** I removed the paragraph saying “The type of **deduce-t<T, N>** in translation unit 1 differs from the type of **deduce-t<T, N>** in translation unit 2 if and only if the type of native-abi<T> in translation unit 1 differs from the type of native-abi<T> in translation unit 2.” after consulting Jens. He said I can’t reasonably say anything about working around ODR problems in an implementation. Implementations thus have to figure this out on their own.

6 **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 can change the type of the **native-abi<T>** alias. — end note ]
[Example: Consider a target architecture supporting the ABI tags \texttt{\_\_simd128} and \texttt{\_\_simd256}, where hardware support for \texttt{\_\_simd256} exists only for floating-point types. The implementation therefore defines \texttt{\_\_native-abi<T>} as an alias for
\begin{itemize}
  \item \texttt{\_\_simd256} if \(T\) is a floating-point type, and
  \item \texttt{\_\_simd128} otherwise.
\end{itemize}
— end example ]

(6.1.4) 28.10.4 simd type traits

\begin{verbatim}
template<class T, class U = typename T::value_type> struct simd_alignment {
  see below;

  simd_alignment<T, U> has a member value if and only if
  \begin{itemize}
    \item \(T\) is a specialization of \texttt{basic_simd_mask} and \(U\) is \texttt{bool}, or
    \item \(T\) is a specialization of \texttt{basic_simd} and \(U\) is a vectorizable type.
  \end{itemize}

  If value is present, the type \texttt{simd_alignment<T, U>} is a \texttt{BinaryTypeTrait} with a base characteristic of \texttt{integral_constant<size_t, N>} for some implementation-defined \(N\) (see \texttt{[simd.copy]} and \texttt{[simd.mask.copy]}). [Note: \texttt{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 ]

  The behavior of a program that adds specializations for \texttt{simd_alignment} is undefined.

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

  The member type is present if and only if
  \begin{itemize}
    \item \(V\) is a specialization of either \texttt{basic_simd} or \texttt{basic_simd_mask},
    \item \(T\) is a vectorizable type, and
    \item \texttt{deduce-t<T, V::size()>} has a member type type.
  \end{itemize}

  Let \texttt{Abi1} denote an ABI tag such that \texttt{basic_simd<T, Abi1>::size() == V::size()}. Where present, the member typedef \texttt{type names basic_simd<T, Abi1>} if \(V\) is a specialization of \texttt{basic_simd} or \texttt{basic_simd_mask<sizeof(T), Abi1>} if \(V\) is a specialization of \texttt{basic_simd_mask}.

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

  Let \(T\) denote
  \begin{itemize}
    \item \texttt{typename V::value_type} if \(V\) is a specialization of \texttt{basic_simd} or
    \item \texttt{integer-from<mask-element-size<V>>} if \(V\) is a specialization of \texttt{basic_simd_mask}.
  \end{itemize}

  The member type is present if and only if
  \begin{itemize}
    \item \(V\) is a specialization of either \texttt{basic_simd} or \texttt{basic_simd_mask}, and
    \item \texttt{deduce-t<T, N>} has a member type type.
  \end{itemize}

  Let \texttt{Abi1} denote an ABI tag such that \texttt{basic_simd<T, Abi1>::size() == V::size()}. Where present, the member typedef \texttt{type names basic_simd<T, Abi1>} if \(V\) is a specialization of \texttt{basic_simd} or \texttt{basic_simd_mask<sizeof(T), Abi1>} if \(V\) is a specialization of \texttt{basic_simd_mask}.
\end{verbatim}
28.10.5 Load and store flags

```cpp
inline constexpr simd_flags<see below> simd_flag_convert{};
inline constexpr simd_flags<see below> simd_flag_aligned{};
template<size_t N> requires (has_single_bit(N))
inline constexpr simd_flags<see below> simd_flag_overaligned{};
```

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

28.10.5.1 Class template `simd_flags` overview

```cpp
template <class... Flags> struct simd_flags {
  // [simd.flags.oper]. simd_flags operators
  template <class... Other>
  friend constexpr auto operator|(simd_flags a, simd_flags<Other...> b);
};
```

[Note: The class template `simd_flags` acts like a integer bit-flag for types. — end note]

Constraints: Every type in `Flags` is a valid template argument to one of `simd_flag_convert`, `simd_flag_aligned`, or `simd_flag_overaligned`.

28.10.5.2 `simd_flags` operators

```cpp
template <class... Other>
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`.

28.10.6 Class template `basic_simd`

28.10.6.1 Class template `basic_simd` overview

```cpp
template<class T, class Abi> class basic_simd {
  public:
    using value_type = T;
    using reference = see below;
    using mask_type = basic_simd_mask<sizeof(T), Abi>;
    using abi_type = Abi;

    static constexpr integral_constant< simd-size-type, simd-size-v<T, Abi>> size {};

    constexpr basic_simd() noexcept = default;

    // [simd ctor]. basic_simd constructors
    template<class U> constexpr basic_simd(U& value) noexcept;
};
```
template<class U, class UAbi>
    constexpr explicit(const basic_simd<U, UAbi>&) noexcept;

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

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

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

// [simd.copy], basic_simd copy functions

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

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

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

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

// [simd.subscr], basic_simd subscript operators

constexpr reference operator[](simd-size-type) \\=
    constexpr value_type operator[](simd-size-type) const;

// [simd.unary], 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;

// [simd.binary], 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;
friend constexpr basic_simd operator<<(const basic_simd &, simd-size-type) noexcept;
friend constexpr basic_simd operator>>(const basic_simd &, simd-size-type) noexcept;

The specializations of class template `basic_simd` are data-parallel types.

Every specialization of `basic_simd` is a complete type. The types `basic_simd<T, deduce-t, N>` for all vectorizable `T` and with `N` in the range of `[1, 64]` are supported. It is implementation-defined whether any other `basic_simd<T, Abi>` specializations with vectorizable `T` are supported.

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

FixMe: drop the note?

If `basic_simd<T, Abi>` is not supported, the specialization has a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. In addition only the `value_type`, `abi_type`, and `mask_type` members are present.

If `basic_simd<T, Abi>` is supported, `basic_simd<T, Abi>` is trivially copyable.

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

```cpp
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) 28.10.6.2 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.

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;

—constexpr reference operator++() && noexcept;
—constexpr value_type operator++(int) && noexcept;
—constexpr reference operator-=(const U) && 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.
Constraints: `std::forward<U>(x)` is well-formed.

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

Returns: A copy of `this`.

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

Constraints: `static_cast<value_type>() @ std::forward<U>()` is well-formed.

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

Returns: A copy of `this`.

```cpp
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`.

```cpp
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.

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

Let From denote the type remove_cvref_t<U>.

**Constraints:** From satisfies convertible_to<value_type>, and either

- From satisfies constexpr-wrapper-like ([simd.syn]) and the actual value of From::value after conversion to value_type will fit into value_type and will produce the original value when converted back to decltype(From::value), or
- From is a vectorizable type and the conversion from From::value_type is value-preserving ([simd.general]), or
- From is not an arithmetic type and does not satisfy constexpr-wrapper-like.

**Effects:** Initializes each element 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:** simd-size-v <U, UAbi> == size() is true.

**Effects:** Initializes the \( i \)th element with static_cast<T>(x[i]) for all \( i \) in the range of \([0, \text{size}())\).

**Remarks:** The expression inside explicit evaluates to true if either

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

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

**Constraints:** basic_simd(gen(integral_constant<simd-size-type, i>()) is well-formed and the return type of gen(integral_constant<simd-size-type, i>()) satisfies convertible_to<value_type> for all \( i \) in the range of \([0, \text{size}())\).

**Effects:** Initializes the \( i \)th element with gen(integral_constant<simd-size-type, i>()) for all \( i \) in the range of \([0, \text{size}())\).

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

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

**Constraints:**

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

- 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 \(\text{simd\_alignment}\_v<\text{basic\_simd}, \text{iter\_value\_t<It>>}\).

- If the template parameter pack \(\text{Flags}\) contains the type identifying \(\text{simd\_flag\_overaligned}<N>\), \(\text{addressof}(*\text{first})\) points to storage aligned by \(N\).

**Effects:** Initializes the \(i\)th element with \(\text{static\_cast<T>}(\text{first}[i])\) for all \(i\) in the range of \([0, \text{size()}]\).

```cpp
template<class It, class... Flags>
constexpr explicit basic_simd(It first, const mask_type & mask, simd_flags<Flags...> = {});
```

**Constraints:**
- \(\text{iter\_value\_t<It>}\) is a vectorizable type, and
- \(\text{It}\) models \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 \(\text{value\_t}\) is value-preserving.

**Preconditions:**
- For all selected indices \(i, [\text{first}, \text{first} + i + 1]\) is a valid range.
- 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 \(\text{simd\_alignment}\_v<\text{basic\_simd}, \text{iter\_value\_t<It>>}\).
- If the template parameter pack \(\text{Flags}\) contains the type identifying \(\text{simd\_flag\_overaligned}<N>\), \(\text{addressof}(*\text{first})\) points to storage aligned by \(N\).

**Effects:** Initializes the \(i\)th element with \(\text{mask}[i] \ ? \ \text{static\_cast<T>}(\text{first}[i]) : T()\) for all \(i\) in the range of \([0, \text{size()}]\).

(6.1.6.4) 28.10.6.4 basic_simd copy functions

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

**Constraints:**
- \(\text{iter\_value\_t<It>}\) is a vectorizable type, and
- \(\text{It}\) models \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 \(\text{value\_t}\) is value-preserving.

**Preconditions:**
- \([\text{first}, \text{first} + \text{size()}]\) is a valid range.
- 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 \(\text{simd\_alignment}\_v<\text{basic\_ simd}, \text{iter\_value\_t<It>>}\).
- If the template parameter pack \(\text{Flags}\) contains the type identifying \(\text{simd\_flag\_overaligned}<N>\), \(\text{addressof}(*\text{first})\) points to storage aligned by \(N\).

**Effects:** Replaces the elements of the \text{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 \([0, \text{size()}]\).
template<class 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 models 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 \( i \), \([first, first + i + 1]\) is a valid range.
• 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) points 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 \( i \).

template<class 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 models contiguous_iterator, and
• Out models 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.
• 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) points to storage aligned by \( N \).

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

template<class 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 models contiguous_iterator, and
• Out models 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 $i$, $[\text{first}, \text{first} + i + 1)$ is a valid range.
- 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) points to storage aligned by $N$.

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

(6.1.6.5) 28.10.6.5 basic_simd subscript operators

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

- Preconditions: $i < \text{size()}$.
- Returns: A reference (see [simd.reference]) referring to the $i^{th}$ element.
- Throws: Nothing.

```cpp
constexpr value_type operator[](simd.size-type i) const&;
```

- Preconditions: $i >= 0 \&\& i < \text{size()}$ is true.
- Returns: The value of the $i^{th}$ element.
- Throws: Nothing.

(6.1.6.6) 28.10.6.6 basic_simd unary operators

Effects in [simd.unary] are applied as unary element-wise operations.

```cpp
constexpr basic_simd& operator++() noexcept;
```

- Constraints: requires (value_type a) { ++a; } is true.
- Effects: Increments every element by one.
- Returns: *this.

```cpp
constexpr basic_simd& operator++(int) noexcept;
```

- Constraints: requires (value_type a) { a++; } is true.
- Effects: Increments every element by one.
- Returns: A copy of *this before incrementing.

```cpp
constexpr basic_simd& operator--() noexcept;
```
Constraints: requires (value_type a) { --a; } is true.

Effects: Decrements every element by one.

Returns: *this.

constexpr basic_simd operator--(int) noexcept;

Constraints: requires (value_type a) { a--; } is true.

Effects: Decrements every element by one.

Returns: A copy of *this before decrementing.

constexpr mask_type operator!() const noexcept;

Constraints: requires (const value_type a) { !a; } is true.

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

constexpr basic_simd operator~() const noexcept;

Constraints: requires (const value_type a) { ~a; } is true.

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

constexpr basic_simd operator+() const noexcept;

Constraints: requires (const value_type a) { +a; } is true.

Returns: *this.

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;
Let \( \text{op} \) be the operator.

**Constraints:** requires \((\text{value_type} \ a, \text{value_type} \ b) \ { \ a \ \text{op} \ b; \} \) is true.

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

friend constexpr \( \text{basic_simd} \) \( \text{operator}<<\)\((\text{const basic_simd} & v, \text{simd-size-type} \ n) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}>>\)\((\text{const basic_simd} & v, \text{simd-size-type} \ n) \) noexcept;

Let \( \text{op} \) be the operator.

**Constraints:** requires \((\text{value_type} \ a, \text{value_type} \ b) \ { \ a \ \text{op} \ b; \} \) is true.

**Returns:** A \( \text{basic_simd} \) 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) 28.10.7.2 \( \text{basic_simd} \) compound assignment

friend constexpr \( \text{basic_simd} \) \( \text{operator}+=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}-=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}*=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}/=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}%=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}&=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}|=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}^=\)\((\text{basic_simd} & \text{lhs}, \text{const basic_simd} & \text{rhs}) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}<<=\)\((\text{basic_simd} & \text{lhs}, \text{simd-size-type} \ n) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}>>=\)\((\text{basic_simd} & \text{lhs}, \text{simd-size-type} \ n) \) noexcept;

Let \( \text{op} \) be the operator.

**Constraints:** requires \((\text{value_type} \ a, \text{value_type} \ b) \ { \ a \ \text{op} \ b; \} \) is true.

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

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

friend constexpr \( \text{basic_simd} \) \( \text{operator}<<=\)\((\text{basic_simd} & \text{lhs}, \text{simd-size-type} \ n) \) noexcept;
friend constexpr \( \text{basic_simd} \) \( \text{operator}>>=\)\((\text{basic_simd} & \text{lhs}, \text{simd-size-type} \ n) \) noexcept;

Let \( \text{op} \) be the operator.

**Constraints:** requires \((\text{value_type} \ a, \text{value_type} \ b) \ { \ a \ \text{op} \ b; \} \) is true.

**Effects:** Equivalent to: return \( \text{operator op (lhs, basic_simd(n))} \);

(6.1.7.3) 28.10.7.3 \( \text{basic_simd} \) compare operators

[simd.cassign]
friend constexpr mask_type operator==(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr mask_type operator!=(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr mask_type operator>=(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr mask_type operator<=(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr mask_type operator>(const basic_simd & lhs, const basic_simd & rhs) noexcept;
friend constexpr mask_type operator<(const basic_simd & lhs, const basic_simd & rhs) noexcept;

1 Let \( \text{op} \) be the operator.
2 Constraints: requires \((\text{value	extunderscore type} \ a, \text{value	extunderscore type} \ b) \{ \ a \ \text{op} \ b; \} \text{ is true.} \)
3 Returns: A \text{basic	extunderscore simd	extunderscore mask} object initialized with the results of applying the indicated operator to \( \text{lhs} \) and \( \text{rhs} \) as a binary element-wise operation.

(6.1.7.4) 28.10.7.4 basic_simd conditional operators [simd.cond]

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

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

(6.1.7.5) 28.10.7.5 basic_simd reductions [simd.reductions]

1 In [simd.reductions], \text{BinaryOperation} shall be a binary element-wise operation.

\[\text{NOTE: In principle, reductions over signed integral \( T \) have a precondition. It could actually be nice to call this out here, and implement a precondition check in simd reductions.}\]

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

2 Constraints: BinaryOperation satisfies invocable<simd<\( T \), 1>, simd<\( T \), 1>>.

3 Mandates: binary\_op can be invoked with two arguments of type basic\_simd<\( T \), \( A1 \)> returning basic\_\text{simd}<\( T \), \( A1 \)> for every \( A1 \) that is an ABI tag type.

4 Preconditions: BinaryOperation does not modify \( x \).

5 Returns: GENERALIZED_SUM(binary\_op, simd<\( T \), 1>(x[i]), ...)\{0\} for all \( i \) in the range of \([0, \text{size}())\) ([numerics.defns]).

6 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, type_identity_t<\( T \)> identity_element, BinaryOperation binary_op);

7 Constraints: BinaryOperation satisfies invocable<simd<\( T \), 1>, simd<\( T \), 1>>.

8 Mandates: binary\_op can be invoked with two arguments of type basic\_simd<\( T \), \( A1 \)> returning basic\_\text{simd}<\( T \), \( A1 \)> for every \( A1 \) that is an ABI tag type.

9 Preconditions:
• BinaryOperation does not modify x.

• For every A1 that is an ABI tag type and for all finite values y representable by T, the results of \( \text{all}_-\text{of}(y == \text{binary}_\text{op}(\text{basic}_\text{simd}<T, A1>(\text{identity}_\text{element}), \text{basic}_\text{simd}<T, A1>(y))) \) and \( \text{all}_-\text{of}(\text{basic}_\text{simd}<T, A1>(y) == \text{binary}_\text{op}(y, \text{basic}_\text{simd}<T, A1>(\text{identity}_\text{element}))) \) are true.

**Returns:** If none_of(mask), returns identity_element. Otherwise, returns \( \text{GENERALIZED}_\text{SUM}(\text{binary}_\text{op}, \text{simd}<T, 1>(x[i]), \ldots)[0] \) for all selected indices i.

**Throws:** Any exception thrown from \text{binary}_\text{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 \( \text{GENERALIZED}_\text{SUM}(\text{binary}_\text{op}, x[i], \ldots) \) 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, multiplies <> binary_op ) noexcept;
```

**Returns:** If none_of(x), returns 1. Otherwise, returns \( \text{GENERALIZED}_\text{SUM}(\text{binary}_\text{op}, x[i], \ldots) \) 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:** is_integral_v<T> is true.

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

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

**Constraints:** T 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, size()).
template<class T, class Abi>
constexpr T reduce_min(
    const basic_simd<T, Abi>&, const typename basic_simd<T, Abi>::mask_type&) noexcept;

Constraints: T models `totally_ordered`.

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

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

Constraints: T models `totally_ordered`.

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

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

template<class T, class Abi>
constexpr auto simd_split (const basic_simd_mask<mask-element-size<T>, Abi>& x) noexcept;

Constraints:
- For the first overload T is a specialization of `basic_simd`.
- For the second overload T is a specialization of `basic_simd_mask`.

Let `N` be `x.size() / T::size()`.

Returns:
- If `x.size() % T::size() == 0`, an array<T, N> with the i\textsuperscript{th} `basic_simd` or `basic_simd_mask` element of the j\textsuperscript{th} array element initialized to the value of the element in x with index `i + j * T::size()`.
- Otherwise, a tuple of `N` objects of type T and one object of type resize_simd_t<x.size() % T::size(), T>. The i\textsuperscript{th} `basic_simd` or `basic_simd_mask` element of the j\textsuperscript{th} tuple element of type T is initialized to the value of the element in x with index `i + j * T::size()`. The i\textsuperscript{th} `basic_simd` or `basic_simd_mask` element of the N\textsuperscript{th} tuple element is initialized to the value of the element in x with index `i + N * T::size()`.
template<class T, class... Abis>
constexpr simd<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 $i + \text{the sum of the width of the first } j \text{ parameters in the xs pack}$. 

(6.1.7.7) 28.10.7.7 Algorithms

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

Constraints: T models `totally_ordered`.

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

```cpp
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 models `totally_ordered`.

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

```cpp
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 models `totally_ordered`.

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

```cpp
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);
```

Constraints: 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 `std::clamp(v[i], lo[i], 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, class T, class U>
constexpr auto simd_select (const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
  noexcept -> decltype (simd-select-impl(c, a, b));

   Returns: As-if `simd-select-impl(c, a, b)`.

(6.1.7.8) 28.10.7.8 basic_simd math library [simd.math]

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 `rebind_simd_t<U, basic_simd<T, Abi>>`.
- All arguments corresponding to `U*`, where `U` is integral, shall be of type `rebind_simd_t<U, basic_simd<T, Abi>>*`.
- 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<T, Abi>::mask_type`. Otherwise, the return type is `rebind_simd_t<R, basic_simd<T, Abi>>`, with `R` denoting the corresponding return type.

It is unspecified whether a call to these overloads with arguments that are all convertible to `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]

FixMe: 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.

(6.1.8) 28.10.8 Class template `basic_simd_mask` [simd.mask.class]

(6.1.8.1) 28.10.8.1 Class template `basic_simd_mask overview` [simd.mask.overview]
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 integral_constant<simd-size-type, simd-size-v<integer-from<Bytes>, Abi>> size {};

    constexpr basic_simd_mask() noexcept = default;

    // [simd.mask ctor], 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<class It, class... Flags>
        constexpr basic_simd_mask(It first, Flags = {});
    template<class It, class... Flags>
        constexpr basic_simd_mask(It first, const basic_simd_mask & mask, simd_flags<Flags...> = {});

    // [simd.mask copy], basic_simd_mask copy functions
    template<class It, class... Flags>
        constexpr void copy_from(It first, simd_flags<Flags...> = {});
    template<class It, class... Flags>
        constexpr void copy_from(It first, const basic_simd_mask & mask, simd_flags<Flags...> = {});
    template<class Out, class... Flags>
        constexpr void copy_to(Out first, simd_flags<Flags...> = {}); const;
    template<class Out, class... Flags>
       constexpr void copy_to(Out first, const basic_simd_mask & mask, simd_flags<Flags...> = {}) const;

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

    // [simd.mask unary], basic_simd_mask unary operators
    constexpr basic_simd_mask operator!() const noexcept;
    constexpr basic_simd<integer-from<Bytes>, Abi> operator+() const noexcept;
    constexpr basic_simd<integer-from<Bytes>, Abi> operator-() const noexcept;
    constexpr basic_simd<integer-from<Bytes>, Abi> operator~() const noexcept;

    // [simd.mask conv], basic_simd_mask conversion operators
    template <class U, class A>
        constexpr basic_simd<integer-from<Bytes>, Abi> operator basic_simd<U, A>() const noexcept;

    // [simd.mask binary], basic_simd_mask binary operators
    friend constexpr basic_simd_mask

47
6 Wording: Add Section 9 of N4808 with modifications

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;

// [simd.mask.cassign], 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;

// [simd.mask.comparison], 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;
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;

// [simd.mask.cond], 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&, same_as<bool> auto, same_as<bool> auto) noexcept;
template <class T0, class T1>
    friend constexpr simd<
        see below, size()>
    simd-select-impl(const basic_simd_mask&, const T0&, const T1&) noexcept;

1 The specializations of class template basic_simd_mask are data-parallel types with element type bool.
2 Every specialization of basic_simd_mask is a complete type. The specialization basic_simd_mask<Bytes, Abi> is not supported if the type integer-from<Bytes> does not exist or if it is not a vectorizable type.

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>>`.

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]

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`:

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

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

### 28.10.8.2 basic_simd_mask constructors

```cpp
constexpr explicit basic_simd_mask(value_type x) noexcept;
```

**Effects:** Initializes each element with `x`.

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

**Constraints:** `simd-size-v<u, UAbi> == size()`.

**Effects:** Initializes the `i`th element with `x[i]` for all `i` in the range of `[0, size())`.

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

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

**Effects:** Initializes the `i`th element with `gen(integral_constant<simd-size-type, i>())` for all `i` in the range of `[0, size())`.

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

```cpp
template<class It, class... Flags>
cconstexpr basic_simd_mask(It first, simd_flags<Flags...> = {});
```

**Constraints:**
- `iter_value_t<It>` is of type `bool`, and
- `It` models `contiguous_iterator`.

**Preconditions:**
- `[first, first + size())` is a valid range.
6 Wording: Add Section 9 of N4808 with modifications

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)` points to storage aligned by `N`.

**Effects:** Initializes the \(i\)th element with `first[i]` for all \(i\) in the range of \([0, \text{size()}]\).

**Throws:** Nothing.

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

**Constraints:**
- `iter_value_t<It>` is of type `bool`, and
- `It` models `contiguous_iterator`.

**Preconditions:**
- For all selected indices \(i\), \([\text{first}, \text{first} + i + 1]\) is a valid range.
- 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)` points to storage aligned by `N`.

**Effects:** Initializes the \(i\)th element with `mask[i] ? first[i] : false` for all \(i\) in the range of \([0, \text{size()}]\).

**Throws:** Nothing.

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

**Constraints:**
- `iter_value_t<It>` is of type `bool`, and
- `It` models `contiguous_iterator`.

**Preconditions:**
- \([\text{first}, \text{first} + \text{size()}]\) is a valid range.
- 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)` points to storage aligned by `N`.

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

**Throws:** Nothing.

```cpp
template<class It, class... Flags>
constexpr void copy_from (It first, const basic_simd_mask & mask, simd_flags<Flags...> = {});
```
6 Wording: Add Section 9 of N4808 with modifications

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

Preconditions:
- For all selected indices `i`, `[first, first + i + 1)` is a valid range.
- 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) points to storage aligned by `N`.

Effects: Replaces the selected elements of the `basic_simd_mask` object such that the `ith` element is replaced with `first[i]` for all selected indices `i`.

Throws: Nothing.

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

Constraints:
- `iter_value_t<Out>` is of type `bool`, and
- `Out` models `contiguous_iterator`, and
- `Out` models `indirectly_writable<value_type>`.

Preconditions:
- `[first, first + size())` is a valid range.
- 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) points 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, size())`.

Throws: Nothing.

```cpp
template<class 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` models `contiguous_iterator`, and
- `Out` models `indirectly_writable<value_type>`.

Preconditions:
- For all selected indices `i`, `[first, first + i + 1)` is a valid range.
- 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>`.

51
• If the template parameter pack Flags contains the type identifying simd_flag_overaligned<N>, addressof(*first) points 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) 28.10.4.4 basic_sind_mask subscript operators

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

Preconditions: i < size().

Returns: A reference (see [simd.reference]) referring to the i-th element.

Throws: Nothing.

constexpr value_type operator[](simd-size i) const &;

Preconditions: i >= 0 && i < size() is true.

Returns: The value of the i-th element.

Throws: Nothing.

(6.1.8.5) 28.10.5 basic_simd_mask unary operators

constexpr basic_simd_mask operator!() const noexcept;

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

constexpr basic_simd<integer-from<Bytes>, Abi> operator+() const noexcept;
constexpr basic_simd<integer-from<Bytes>, Abi> operator-() const noexcept;
constexpr basic_simd<integer-from<Bytes>, Abi> 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) 28.10.6 basic_simd_mask conversion operators

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


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

(6.1.9) 28.10.9 Non-member operations

(6.1.9.1) 28.10.9.1 basic_simd_mask binary operators
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) 28.10.9.2 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.

(6.1.9.3) 28.10.9.3 basic_simd_mask comparisons [simd.mask.comparison]

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;
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.

(6.1.9.4) 28.10.9.4 basic_simd_mask conditional operators [simd.mask.cond]
friend constexpr basic_simd_mask simd-select-impl(
    const basic_simd_mask & mask, const basic_simd_mask & a, const basic_simd_mask & b) noexcept;

1 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()).

friend constexpr basic_simd_mask
simd-select-impl(const basic_simd_mask & mask, same_as<bool> auto a, same_as<bool> auto b) noexcept;

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

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

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

4 Constraints:
   • $U$ is a vectorizable type, and
   • sizeof($U$) == Bytes, and
   • $T0$ satisfies convertible_to<simd<$U$, size()>>, and
   • $T1$ satisfies convertible_to<simd<$U$, size()>>.

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

(6.1.9.5) 28.10.9.5 basic_simd_mask reductions

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

1 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;

2 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;

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

(template<size_t Bytes, class Abi>
constexpr simd-size-type reduce_count(const basic_simd_mask<Bytes, Abi>& k) noexcept;
\( \text{Returns: The number of boolean elements in } k \text{ that are true.} \)

```cpp
template<\text{size\_t} \text{Bytes}, \text{class} \text{Abi}>
\text{constexpr} \text{simd\_size\_type} \text{reduce\_min\_index}(\text{const basic\_simd\_mask<Bytes, Abi>&} k);
```

\( \text{Preconditions: any\_of(k) is true.} \)

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

```cpp
template<\text{size\_t} \text{Bytes}, \text{class} \text{Abi}>
\text{constexpr} \text{simd\_size\_type} \text{reduce\_max\_index}(\text{const basic\_simd\_mask<Bytes, Abi>&} k);
```

\( \text{Preconditions: any\_of(k) is true.} \)

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

```cpp
\text{constexpr bool all\_of(same\_as<bool> auto) noexcept ;}
\text{constexpr bool any\_of(same\_as<bool> auto) noexcept ;}
\text{constexpr bool none\_of(same\_as<bool> auto) noexcept ;}
\text{constexpr simd\_size\_type reduce\_count(same\_as<bool> auto x) noexcept ;}
```

\( \text{Returns: all\_of and any\_of return their arguments; none\_of returns the negation of its argument; reduce\_count returns the integral representation of } x. \)

```cpp
\text{constexpr simd\_size\_type reduce\_min\_index(same\_as<bool> auto y)};
\text{constexpr simd\_size\_type reduce\_max\_index(same\_as<bool> auto z)};
```

\( \text{Preconditions: The value of the argument is true.} \)

\( \text{Returns: 0.} \)

### ACKNOWLEDGMENTS

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


