ABSTRACT

After the Parallelism TS 2 was published in 2018, data-parallel types (simd<T>) have been implemented and used. Now there is sufficient feedback to improve and merge Section 9 of the Parallelism TS 2 into the IS working draft.

CONTENTS

1 Changelog .............................. 1
  1.1 Changes from revision 0 ................. 1
  1.2 Changes from revision 1 ................ 2
  1.3 Changes from revision 2 ................. 2
2 Straw Polls ....................... 3
  2.1 SG1 at Kona 2022 .................. 3
3 Introduction ................... 3
  3.1 Related papers ........................ 4
4 Changes after TS feedback

4.1 Improve ABI tags ......................................................... 5
4.2 Simplify/generalize casts ........................................... 7
4.3 Add simd_mask generator constructor ......................... 11
4.4 Default load/store flags to element_aligned ............... 11
4.5 Contiguous iterators for loads and stores ................. 12
4.6 constexpr everything ................................................. 12
4.7 Specify simd::size as integral_constant ..................... 13
4.8 Replace where facilities ........................................... 13
4.9 Make use of int and size_t consistent ....................... 16
4.10 Clean up math function overloads ......................... 18

5 Open questions

5.1 Alternatives to hmin and hmax ................................. 18
5.2 Make simd_mask reductions consistent with <bit> .......... 19
5.3 Argument order and naming of masked overloads ........... 22
5.4 Converting loads & stores consistency ...................... 23
5.5 Integration with ranges ............................................. 25
5.6 Correct place for simd in the IS? ............................ 25
5.7 element_reference is overspecified ......................... 25

6 Wording

6.1 Add Section 9 of N4808 with modifications .......... 26
6.2 Diff against Parallelism TS 2 (N4808) ....................... 56

A Bibliography 89
1.1 changes from revision 0

Previous revision: P1928R0

- Target C++26, addressing SG1 and LEWG.
- Call for a merge of the (improved & adjusted) TS specification to the IS.
- Discuss changes to the ABI tags as consequence of TS experience; calls for polls to change the status quo.
- Add template parameter \( T \) to `simd_abi::fixed_size`.
- Remove `simd_abi::compatible`.
- Add (but ask for removal) `simd_abi::abi_stable`.
- Mention TS implementation in GCC releases.
- Add more references to related papers.
- Adjust the clause number for [numbers] to latest draft.
- Add open question: what is the correct clause for [simd]?
- Add open question: integration with ranges.
- Add `simd_mask` generator constructor.
- Consistently add simd and simd_mask to headings.
- Remove experimental and parallelism_v2 namespaces.
- Present the wording twice: with and without diff against N4808 (Parallelism TS 2).
- Default load/store flags to `element_aligned`.
- Generalize casts: conditionally `explicit` converting constructors.
- Remove named cast functions.
1.2 changes from revision 1

Previous revision: P1928R1

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

1.3 changes from revision 2

Previous revision: P1928R2

- Propose alternative to `hmin` and `hmax`.
- Discuss `simd_mask` reductions wrt. consistency with `<bit>`. Propose better names to avoid ambiguity.
- Remove `some_of`.
- Add unary `~` to `simd_mask`.
- Discuss and ask for confirmation of masked “overloads” names and argument order.
- Resolve inconsistencies wrt. `int` and `size_t`: Change `fixed_size` and `resize_simd` NTTPs from `int` to `size_t` (for consistency).
- Discuss conversions on loads and stores. (Section 5.4)
2 Straw Polls

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

Replace wording.

Apply the new library specification style from P0788R3.

Add numeric_limits / numeric traits specializations since behavior of e.g. simd<float> and float may differ for reasonable implementations.

Consider adding a note that recommends implementations to let simd primary operations behave like operations of arithmetic types to never be function calls. (cf. GCC PR108030)

2 Straw Polls

2.1 sg1 at kona 2022

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

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

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

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

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

3 Introduction

[P0214R9] introduced simd<T> and related types and functions into the Parallelism TS 2 Section 9. The TS was published in 2018. An incomplete and non-conforming (because P0214 evolved) implementation existed for the whole time P0214 progressed through the committee. Shortly after
the GCC 9 release, a complete implementation of Section 9 of the TS was made available. Since GCC 11 a complete `simd` implementation of the TS is part of its standard library.

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

### 3.1 RELATED PAPERS

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

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

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

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

**DRAFT ON NON-MEMBER OPERATOR[]** TODO

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

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

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

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

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

**P2664** Permutations for `simd`.
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". This paper captures the major change requests but should still be considered a work-in-progress.

4.1 Improve ABI tags

I received consistent feedback that simd_abi::compatible<T> is the wrong default and it should rather be simd_abi::native<T> instead. All my tutorial material instructed users to use std::experimental::native_simd<T>. There really is little use for simd_abi::compatible<T>. The preferred approach should be the use of simd_abi::native<T> together with compiler flags that limit the available registers and instructions to whatever the user deems “compatible”. Consequently, there is no reason to keep simd_abi::compatible<T> in its current form.

Another common question was about a “fixed size” ABI tag, similar to std::experimental::simd_abi::fixed_size<N> but without the ABI compatibility cost.¹ Basically, the ABI footgun should be as dangerous as std::experimental::simd_abi::native<T>. The answer to that FAQ is to use std::experimental::simd_abi::deduce_t<T, N> as ABI tag. This will provide you with a high-performance footgun, if supported, but might also fall back to std::experimental::simd_abi::fixed_size<N>. With std::experimental::simd_abi::deduce_t<T, N> turning out to be used potentially more often than std::experimental::simd_abi::fixed_size<N> the aliases and names should be revisited. My proposal:

A0 = simd_abi::native<T> (no change from the TS semantics)

simd<T, A0> abstracts a SIMD register (or similar) with highest performance on the target system (typically widest available register, but that’s a QoI choice). Consequently, the number of elements is chosen by the implementation and may differ for different T and different compiler flags. simd_abi::native<T> is an alias for an unspecified type. simd_abi::native<T> can be an alias for simd_abi::scalar. If sizeof(simd<T, A0>) or

¹ Implementations of the TS are encouraged to make passing fixed_size objects ABI compatible between different hardware generations and/or even different architectures.
4 Changes after TS feedback

alignof(simd<T, A0>) in TU1 differ from the same expressions in TU2, then the types A0 in TU1 and TU2 have a different name.

A1 = simd_abi::fixed_size<T, N> (different to the TS semantics)
simd<T, A1> abstracts one or more registers storing N values. The actual hardware resources might store more values; but instructions are generated to make it appear as if there are exactly N values stored and manipulated.

Parameter passing may be ABI incompatible between different TUs when compiled with different compiler flags. Therefore, if sizeof(simd<T, A0>)\(^2\) or alignof(simd<T, A0>) in TU1 differ from the same expressions in TU2, then the types A1 in TU1 and TU2 have a different name. This new requirement (wrt. the TS) is the reason for the additional T parameter. This allows an implementation to define the fixed_size alias as e.g. `template <typename T, int N> using fixed_size = _Fixed<N, native<T>>;`. A1 and A0 are always different types, i.e. even if `simd_size_v<T, A0> == N`.

A major difference to std::experimental::simd_abi::fixed_size<N> in the TS is about simd_mask. In order to support ABI stability the simd_mask implementation must choose one form of storage for all possible targets:

- full SIMD registers with all bits set to 1 or 0 per corresponding element
- bitmasks
- an array of bool or similar

The new intent for the fixed_size<T, N> ABI tag would be to allow simd_mask<T, A1> to use either choice depending only on compiler flags.

A2 = simd_abi::scalar
No change.

At this point the simd_abi::deduce facility seems to be obsolete. However, it is still a useful tool for implementing the rebind_simd and resize_simd traits. Without more compelling reason for removal, it should be merged as is.

NAMING DISCUSSION

For context on naming, consider the use-cases that the ABI tags serve:

**simd_abi::native<T>** The equivalent to T: a direct abstraction of available hardware resources in terms of registers and instructions.

\(^2\) A0 is not a typo; this depends on simd_abi::native<T>
4 Changes after TS feedback

\texttt{simd}_{\text{abi}}::\text{fixed\_size}\langle T, N \rangle$ Higher abstraction level than $\text{native}\langle T \rangle$: the user/algorithm dictates the number of elements to be processed in parallel. The objects might not be direct mappings to hardware resources, but they use the best that is available on the given target system.

\texttt{simd}_{\text{abi}}::\text{scalar}$ The actual type of $\text{native}\langle T \rangle$ if the target hardware has no support for parallel processing of elements of $T$.\footnote{A typical example is $\text{simd}\_{\text{abi}}::\text{native}\langle \text{long double} \rangle$.} In addition, $\text{simd}\langle T, \text{simd}\_{\text{abi}}::\text{scalar} \rangle$ can be a useful debugging tool.

For reference, the name $\text{fixed\_size}$ is my preference over the following alternatives:

- $\text{simd}_{\text{abi}}::\text{fixed\_native}\langle N \rangle$ with $\text{simd}$ alias $\text{fixed\_native\_simd}\langle T, N \rangle$
- $\text{simd}_{\text{abi}}::\text{fixed}\langle N \rangle$ with $\text{simd}$ alias $\text{fixed\_simd}\langle T, N \rangle$
- $\text{simd}_{\text{abi}}::\text{sized}\langle N \rangle$ with $\text{simd}$ alias $\text{sized\_simd}\langle T, N \rangle$

### 4.2 Simplify/generalize casts

The change to the ABI tags requires a reconsideration of cast functions and implicit and explicit casts between data-parallel types of different ABI tags. This is in addition to TS feedback on casts being too strict or cumbersome to use.

#### 4.2.1 More (explicit) converting constructors

The TS allows implicit casts between $\text{fixed\_size}\langle N \rangle$ types that only differ in element type and where the values are preserved ("every possible value of $U$ can be represented with type $\text{value\_type}$").

However, from experience with the TS, it is better to also enable implicit conversions between any $\text{simd}$ specializations with equal element count, even if such a conversion might be non-portable between targets with different native SIMD widths. The expectation is, that users set up their types according to a pattern similar to Listing 1. Thus, users will work with a set of types that have equal

```cpp
using floatv = std::simd<float>;
using doublev = std::rebind_simd_t<\text{double}, floatv>;
using int32v = std::rebind_simd_t<\text{int32_t}, floatv>;
using uint32v = std::rebind_simd_t<\text{uint32_t}, floatv>;
using int16v = std::rebind_simd_t<\text{int16_t}, floatv>;
using uint16v = std::rebind_simd_t<\text{uint16_t}, floatv>;
// ...
```

Listing 1: Recommended setup of $\text{simd}$ types
number of elements by construction. Some of the types may use the fixed_size ABI tag and some may use an extended ABI tag. This detail should not stop the user from being able to cast between a compiler-flag dependent subset of these types.

Besides a constraint on the number of elements being equal, the converting constructor should be conditionally explicit: Implicit casts are only allowed if the element type conversion is value-preserving (same wording as in the TS).

This resolves major inconveniences when working with mixed-precision operations (cf. Tony Table 1). Type conversions for simd are still less error-prone than builtin types, because conversions that might lose information require an explicit cast. Also, unintended widening of the SIMD register size can happen, but typically leads to the need for an explicit cast in the complete statement (cf. Listing 2).

Tony Table 1: Improved generic code after adding converting constructors

<table>
<thead>
<tr>
<th>Parallelism TS 2</th>
<th>with P1928R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace stdx = std::experimental;</td>
<td>// assuming simd in namespace std</td>
</tr>
<tr>
<td>template &lt;class T&gt; void f(T a, int b)</td>
<td>template &lt;class T&gt; void f(T a, int b)</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>using I = std::conditional_t&lt;</td>
<td></td>
</tr>
<tr>
<td>std::is_simd_v&lt;T&gt;,</td>
<td></td>
</tr>
<tr>
<td>std::rebind_simd_t&lt;int, T&gt;, int&gt;;</td>
<td></td>
</tr>
</tbody>
</table>
| I c; | I c = static_cast<>
| if constexpr (std::is_simd_v<T>) | std::is_simd_v<T>,
| c = std::static_simd_cast<int>(a) + b; | std::rebind_simd_t<int, T>, int>;
| } | |
| else { | I c = static_cast<int>(a) + b; |
| c = static_cast<int>(a) + b; | } |
| g(c); | } |

Listing 2: Mixed precision code using the types from Listing 1, ensuring equal element count

```c++
void f(int32v a, doublev b, floatv c)
{
  doublev x = a * b + c; // OK: implicit (value-preserving) conversion from int and float
  // to double. Requires twice the register space, but there's no way around it and the
  // result type requires it anyway.
  int32v y = a * b; // ERROR: implicit conversion from double to int not value-preserving
  int32v z1 = static_cast<int32v>(a * b); // OK: cast hints at implicit register widening
  int32v z2 = a * static_cast<int32v>(b); // OK
}
```
4.2.2 REMOVE NAMED CAST FUNCTIONS

From the cast functions `std::experimental::to_fixed_size`, `std::experimental::to_native`, and `std::experimental::to_compatible` only the conversions from `simd_abi::fixed_size<T, N>` to `simd_abi::native<T>` and back may still benefit from a named cast function. Most importantly, the conversion from `native` to its `fixed_size` counterpart benefits from a cast expression that does not require spelling out the destination type. However, since converting constructors are provided by the standard library, it is simple for users to define their own `to_fixed_size` function if they want one (e.g. Listing 3). The reverse cast can trivially be spelled out as `static_cast<simd<T>>(y)` in program code. The only motivation for adding a `to_native` function would be the provision of a counterpart for the `to_fixed_size` cast function.

Besides the functions only implementing trivial implicit casts, there is little to no need for these functions. The named cast functions are therefore removed altogether.

4.2.3 REMOVE SIMD_CAST AND STATIC_SIMD_CAST

There are two cast function templates in the TS: `simd_cast` and `static_simd_cast`. The former is equivalent to the latter except that only value-preserving conversions are allowed. The template parameter can either be a `simd` specialization or a vectorizable type `T`. In the latter case, the cast function determines the return type as `fixed_size_simd<T, input.size()>`.

Since we allow all conversions covered by `std::experimental::simd_cast` and `std::experimental::static_simd_cast` via `std::simd` constructors, the cast functions can be removed altogether. The lost feature (cast via element type) can be replaced using `rebind_simd` as shown in Tony Table 2.

4.2.4 MASK CASTS

`simd_mask` casts should work when `simd` casts work. I.e. if `simd<T0, A0>` is implicitly convertible to `simd<T1, A1>` then `simd_mask<T0, A0>` is implicitly convertible to `simd_mask<T1, A1>`. The reverse (if `simd_mask` is convertible then `simd` is convertible) does not have to be true. Specifically, the TS allows all `fixed_size<N>` mask to be interconvertible, irrespective of the element type. For the IS merge, the proposal is to make this more consistent with `simd` while also
4 Changes after TS feedback

<table>
<thead>
<tr>
<th>Parallelism TS 2</th>
<th>with P1928R3</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>template</em> &lt;typename V&gt;</td>
<td></td>
</tr>
<tr>
<td><em>void</em> f(V x)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td><em>const auto</em> y = std::<em>static_simp_cast</em>&lt;double&gt;(x);</td>
<td></td>
</tr>
<tr>
<td>// ...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td><em>template</em> &lt;typename V&gt;</td>
<td></td>
</tr>
<tr>
<td><em>void</em> f(V x)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td><em>const auto</em> y = std::<em>rebind_simd</em>T&lt;double, V&gt;(x);</td>
<td></td>
</tr>
<tr>
<td>// ...</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Tony Table 2: Casting without specifying the target ABI tag

preserving most of the convenience: Allow implicit conversions if the `sizeof` the element types are equal, otherwise the conversion must be explicit.

Conversions with different element count are not possible via a constructor (consistent with `simd`). This would require a different function, such as the `resize<N>(simd)` function proposed by Towner et al. [P2638R0].

4.2.5 COMPLETE CASTS FOR `simd_mask`

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

The proposed changes for casts solve this issue.

4.2.6 SUMMARY OF CASTS

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.

3. `simd_mask<T0, A0>` is convertible to `simd_mask<T1, A1>` if `simd_size_v<T0, A0> == simd_size_v<T1, A1>`.

4. `simd_mask<T0, A0>` is implicitly convertible to `simd_mask<T1, A1>` if, additionally, `sizeof(T0) == sizeof(T1)`.

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

6. `simd_mask<T0, A0>` can be bit_casted to `simd_mask<T1, A1>` if `sizeof(simd_mask<T0, A0>) == sizeof(simd_mask<T1, A1>)`.
4 Changes after TS feedback

4.3 ADD simd_mask GENERATOR CONSTRUCTOR

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

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

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

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

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

Therefore add:

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

4.4 DEFAULT LOAD/STORE FLAGS TO element_aligned

Consider:

```cpp
1 std::simd<float> v(addr, std::vector_aligned);
2 v.copy_from(addr + 1, std::element_aligned);
3 v.copy_to(dest, std::element_aligned);
```

Line 1 supplies an optimization hint to the load operation. Line 2 says what really? “Please don’t crash. I know this is not a vector aligned access.” Line 3 says: “I don’t know whether it’s vector aligned or not. Compiler, if you know more, please optimize, otherwise just don’t make it crash.” (To clarify, the difference between lines 2 and 3 is what line 1 says about the alignment of addr.)

4 Of course, vector aligned is equivalent to element aligned if simd<float>::size() == 1
4 Changes after TS feedback

both cases of \texttt{element\_aligned} access, the developer requested a behavior we take as given in all other situations. Why does the TS force to spell it out in this case?

Since C++20, we also have another option:

\begin{verbatim}
1 std::simd<float> v(std::assume_aligned<std::memory_alignment_v<std::simd<float>>>(addr));
2 v.copy_from(addr + 1);
3 v.copy_to(dest);
\end{verbatim}

This seems to compose well, except that line 1 is rather long for a common pattern in this interface. Also, this removes implementation freedom because the library cannot statically determine the alignment properties of the pointer.

Consequently, as a minimal improvement to the TS keep the load/store flags as is, but default them to \texttt{element\_aligned}. I.e.:

\begin{verbatim}
1 std::simd<float> v(addr, std::vector_aligned);
2 v.copy_from(addr + 1);
3 v.copy_to(dest);
\end{verbatim}

Section 5.4 discusses an option for additional flags.

4.5 CONTIGUOUS ITERATORS FOR LOADS AND STORES

After Ranges and Concepts introduced \texttt{std::contiguous\_iterator}, the load/store interface for \texttt{simd} can easily be generalized from \texttt{U*} to \texttt{std::contiguous\_iterator} with additional constraints for \texttt{input\_iterator/output\_iterator} and \texttt{iter\_value\_t}. This was not a possible design choice for the TS but does make a lot of sense to modernize with the merge. Therefore, the merge generalizes the load/store interfaces to look like Listing 4.

\begin{verbatim}
1 template <contiguous_iterator It, typename Flags = element\_aligned\_tag>
2 requires detail::vectorizable<Iter\_value\_t<It>>
3 constexpr void copy_from(const It& first, Flags f = {});
4
5 template <contiguous_iterator It, typename Flags = element\_aligned\_tag>
6 requires output\_iterator<It, Tp> && detail::vectorizable<Iter\_value\_t<It>>
7 constexpr void copy_to(const It& first, Flags f = {}) const;
\end{verbatim}

Listing 4: \texttt{copy\_from} and \texttt{copy\_to} declarations using \texttt{contiguous\_iterator}

4.6 constexpr EVERYTHING

The libstdc++ implementation implements the complete TS API as \texttt{constexpr} as an optional extension. This is useful (e.g. for computing constants) and not a significant implementation burden. Users (as well as Towner et al. [P2638R0]) have called for \texttt{constexpr}. The merge consequently adds \texttt{constexpr} to all functions.
4.7 Specify \texttt{simd::size} as \texttt{integral\_constant}

The TS specifies \texttt{simd::size} as a \texttt{static const expr} function returning the number of elements of the \texttt{simd} specialization. Instead of a function, this paper uses a static data member of type \texttt{std::integral\_constant< std::size\_t, N >}, which is both convertible to \texttt{std::size\_t} and callable. The upside of using a static data member is that it can be used as function parameter without conversion to integer and thus easily pass the size into a function as constant expression. See Listing 5 for an example.

1. \texttt{template < std::ranges::contiguous\_range R, std::size\_t Size >}
2. \texttt{std::span< const std::ranges::range\_value\_t<R >, Size >}
3. \texttt{auto subscript(\texttt{const R} \& r, std::size\_t first, \texttt{std::integral\_constant< std::size\_t, Size >}) \{}
4. \texttt{\quad return \texttt{std::span< const std::ranges::range\_value\_t<R >, Size >}\
5. \quad \quad \quad \texttt{(std::ranges::data(r)} + \texttt{first}, \texttt{Size());}}
6. \texttt{\}}
7. \texttt{void g(\texttt{std::vector< float >} data) \{}
8. \texttt{\quad std::\texttt{simd< float >} v;}
9. \texttt{\quad for (std::size\_t i = 0; i + v.size < data.size(); i += v.size) \{}}
10. \texttt{\quad \quad v = subscript(data, i, v.size); // simd::simd(span) to be proposed}
11. \texttt{\quad // ...}
12. \texttt{\}}
13. \texttt{\}

Listing 5: Example: Pass \texttt{simd::size} as "constant expression function argument"

4.8 Replace \texttt{where} facilities

The \texttt{where} functions and corresponding \texttt{where\_expression} have been the most controversial part going into the TS. My interpretation of the feedback I received is that users can work with it but do not find it intuitive. Instead, many have asked for a blend / select / conditional operator instead. Whenever I asked users whether they would like to use the \texttt{?:} operator I got positive and often enthusiastic responses. An overloaded \texttt{operator?} would open the door to generic and intuitive SIMD code.

A major motivation for the \texttt{where} function in the TS was its ability to express masked operations in addition to masked assignments. This enables library implementations to explicitly use masked operation intrinsics instead of resorting to an unmasked operation with subsequent masked assignment. The latter can be contracted to a masked operation by compilers, but obviously there’s no guarantee. In any case, the topic is a QoI issue that doesn’t have to dictate the API.

If \texttt{operator?} had been overloadable when I designed \texttt{std::experimental::simd} then I would have proposed \texttt{?:} overloads for \texttt{simd\_mask} and \texttt{simd}. Consequently, \texttt{where} would likely not have existed. Sadly we still cannot overload \texttt{operator?}: even though there has been positive feedback in EWG-I. That work is currently blocked on \texttt{[P2600R0]}. 

13
4 Changes after TS feedback

4.8.1 Proposed replacements for `where`

This paper proposes the following replacements for `std::experimental::where`:

- Overloads for `simd::copy_from`, `simd::copy_to`, `simd_mask::copy_from`, `simd_mask::copy_to`, `reduce`, `hmin`, and `hmax` with additional `simd_mask` parameter. There are still open questions on these functions, discussed in Section 5.3.

- hidden friend `operator?:` / `conditional_operator` functions in `simd` and `simd_mask`:
  - `simd simd::operator?:(mask_type, simd, simd)`
  - `template <class U1, class U2> requires convertible_to<simd_mask, rebind_simd_t<common_type_t<U1, U2>, simd_mask> friend constexpr rebind_simd_t<common_type_t<U1, U2>, simd_type>`
    `simd_mask::operator?:(simd_mask, U1, U2)`
  - `simd_mask simd_mask::operator?:(simd_mask, simd_mask, simd_mask)`
  - `simd_mask simd_mask::operator?:(simd_mask<K, KABI>, simd_mask, simd_mask<U, UABI>)` (for disambiguation of the above because `simd_masks` can be interconvertible)
  - `simd_mask simd_mask::operator?:(simd_mask, bool, bool)` (for consistency; it's not very useful)

- facilities for converting `simd_mask<T>` to `simd<T>` with values 0 or 1:
  - `simd_mask::operator simd_type(not explicit, preferably with 4.1 of [P2600R0] adopted)`
  - unary `simd_mask::operator+; equivalent to +(operator simd_type())`
  - unary `simd_mask::operator-; equivalent to -(operator simd_type())`
  - unary `simd_mask::operator ~; equivalent to ~(operator simd_type())`

(Wording for the above is still TBD.)

4.8.2 Examples

Listing 6 presents a few simple examples of working with a ` simd_mask` result in the absence of `where`. Note that the compiler I used implements the ADL fix proposed in [P2600R0] and implements `operator?:` overloading as explored in [D0917].

---

5 as soon as EWG lifts the restriction...
```cpp
auto f0(std::simd<int> x) { return x > 0 ? 2 * x : x; }

auto f1(std::simd<int> x) { return x > 0 ? 1 : 0; }

auto f2(std::simd<int> x) { return std::simd(x > 0); }

auto f3(std::simd<int> x) { return -(x > 0); }

auto f4(std::simd<int> x) { return x > 0 ? -1 : 0; }

auto f5(std::simd<int> x) { return x > 0 ? true : false; }
```

Listing 6: `std::simd` conditionals without `where` and with [P2600R0] and [D0917], showing the corresponding assembly output (`gcc -O2 -std=c++23 -march=skylake-avx512; personal GCC 12.1 branch with patches implementing [P2600R0] and [D0917])
4 Changes after TS feedback

- The function \( f_0 \) scales all positive values in \( x \) by 2. The compiler contracts the blending of \( 2 \times x \) and \( x \) with the multiply operation (a left shift by 1) to a masked left shift instruction.

- The functions \( f_1 \) and \( f_2 \) both return a \( \text{simd}<\text{int}> \) where all positive entries of \( x \) are replaced by 1 and the remaining entries are 0. I.e. converting the comparison result to \( \text{simd} \) works analogue to promotion of \( \text{bool} \) to \( \text{int} \).

- The functions \( f_3 \) and \( f_4 \) both return a \( \text{simd}<\text{int}> \) where all positive entries of \( x \) are replaced by -1 and the remaining entries are 0. The ISA allows a more efficient translation and the compiler recognizes the pattern in both variants.

- Finally, to complete the set, \( f_5 \) shows how one could even blend \( \text{bool} \) arguments into a \( \text{simd}_{\text{mask}} \). The compiler recognizes that the conditional operator is a no-op and simply returns the result of the comparison itself.

Tony Table 3 presents an algorithm for counting all positive \( \text{float} \) values in a \( \text{std}::\text{vector} \). For simplicity, the code uses \( \text{vector}<\text{simd}<\text{float}>> \) and assumes the \( \text{vector} \) is not empty. If a \( \text{simd}_{\text{mask}} \) implicitly converts to a \( \text{simd} \) (as proposed and analogue to \( \text{bool} \)), the code is simplified significantly. However, at this point, the TS implementation compiles to a masked add instruction while the implementation for this paper does not. The difference is that the former executes an unmasked addition followed up by a masked assignment while the latter converts the mask into a \( \text{simd} \) of 1s and 0s followed up by an unmasked addition. The compiler needs to recognize this pattern in order to reach the same performance (QoI).

4.9 Make use of \text{int} and \text{size_t} consistent

The TS uses \text{int} as NTTP for

- \text{std}::\text{experimental}::\text{simd}_{\text{abi}}::\text{fixed}::\text{size},

- \text{std}::\text{experimental}::\text{fixed}::\text{size}::\text{simd},

- \text{std}::\text{experimental}::\text{fixed}::\text{size}::\text{simd}_{\text{mask}}, and

- \text{std}::\text{experimental}::\text{resize}::\text{simd}.

The constant \text{std}::\text{experimental}::\text{simd}_{\text{abi}}::\text{max}::\text{fixed}::\text{size} is of type \text{int}. The TS uses \text{size_t} as NTTP for

- \text{split}, and

- \text{split}::\text{by}.

This paper uses \text{integral}::\text{constant}<\text{size_t}, \mathcal{W}> for
Parallelism TS 2 with P1928R3

```cpp
namespace stdx = std::experimental;

int count_positive(
    const std::vector<std::native_simd<float>>& x)
{
    // simplify generated assembly:
    if (x.size() == 0) std::unreachable();
    using floatv = std::native_simd<float>;
    using intv = std::rebind_simd_t<int, floatv>;
    intv counter = {};
    for (std::simd v : x) {
        auto k = std::static_simd_cast<intv::mask_type>(v > 0);
        ++where(k, counter);
    }
    return reduce(counter);
}
```

Tony Table 3: Counting positive values in a `std::vector`
Finally, `simd_size_v<T, Abi>` is of type `size_t`.

All of these integers denote a SIMD width. They should be consistent. Since the `size` member will never get consensus to use type `int`, the decision falls on `size_t` for all.

The merge proposal therefore uses `size_t` for `std::experimental::simd_abi::fixed_size`, `std::experimental::fixed_size_simd`, `std::experimental::fixed_size_simd_mask`, `std::experimental::resize_simd`, and `std::experimental::simd_abi::max_fixed_size`.

### 5.1 Alternatives to `hmin` and `hmax`

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

With C++17, there was nothing equivalent to `std::plus<>` for minimum and maximum. Since the merge of Ranges (C++20), we have `std::ranges::min` and `std::ranges::max`. The `reduce(simd)` specification requires the `binary_op` to be callable with two `simd` arguments, though (split initial argument in half, call `binary_op`, split again, call `binary_op`, ...until only a scalar is left). This doesn't work with `std::ranges::min` (and `max`) because it requires an lvalue reference as return type. If we added another `operator()` to `std::ranges::min`, then their use with `reduce(simd)` would be slightly inconsistent:
However, if `simd` will be an `input_range` (see Section 5.5) then the `std::ranges::min(ranges::input_range auto& ...) overload matches and `std::ranges::min(simd)` works out of the box. We could then leave it up to QoI to recognize the opportunity for a SIMD implementation of the reduction.

Alternatively (or in addition) we could rename the TS functions to `reduce_min(simd)` and `reduce_max(simd)`.

### 5.1.1 Suggested polls

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

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

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

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

**Poll:** Extend `std::ranges::min` and `max` to allow prvalue return types.

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

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

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

### 5.2 Make simd_mask reductions consistent with <bit>

The functions in `<bit>` were added in C++20 [P0553R4] and now `simd_mask` needs to adjust for consistency. Overloads for `simd<unsiged integer type>` will also have to be added. But not with this `merge` paper. I promise to write a follow-up paper.

The three relevant functions in the TS are

- `popcount(simd_mask)` and `popcount(bool)`: The names match. However, currently `popcount(bool)` is ill-formed. This would be very
unfortunate for generic code that calls \texttt{popcount} on the result of a comparison. For \texttt{simd\_mask<T>} this operation provides value, for \texttt{T} it simply casts the \texttt{bool} to an \texttt{int}. The TS merge would therefore add \texttt{bool} to the list of valid types for \texttt{popcount}.

- \texttt{find\_first\_set(simd\_mask)} and \texttt{find\_first\_set(\textit{bool})}:
  This matches either \texttt{countl\_zero} or \texttt{countr\_zero}, except that the \texttt{simd\_mask} version has a precondition. Same issue for \texttt{\textit{bool}} as discussed for \texttt{popcount}.

- \texttt{find\_last\_set(simd\_mask)} and \texttt{find\_last\_set(\textit{bool})}:
  This matches either \texttt{countl\_zero} or \texttt{countr\_zero}, except that the \texttt{simd\_mask} version has a precondition. Same issue for \texttt{\textit{bool}} as discussed for \texttt{popcount}.

I believe we should remove the precondition on the \texttt{simd\_mask} reductions and allow masks where \texttt{none\_of} returns \texttt{true}. Then behavior is identical to \texttt{countl\_zero} and \texttt{countr\_zero}. The reason for the precondition was efficiency on older ISAs. At this point, especially with the precedent set by the \texttt{count[lr]} functions, I don’t believe the performance issue is worth the UB.

Renaming the \texttt{find\_first/last\_set} functions is not obvious though. Consider that a \texttt{simd\_mask \textit{k}} can be indexed via subscript operator. Then \texttt{\textit{k}[0]} identifies the first element in \texttt{k}. For all of us who think left-to-right, it would therefore be intuitive to have \texttt{countl\_zero} be the replacement for \texttt{find\_first\_set}. But if you were to convert \texttt{\textit{k}} into a \texttt{std::bitset} and convert the \texttt{bitset} into an unsigned integer, then \texttt{countl\_zero} would have the opposite meaning (starting from the most significant bit). Indexing a \texttt{simd\_mask} and a \texttt{bitset} is consistent as is the confusion about left and right with regard to the integer representation of a \texttt{bitset}.

I believe it was wrong to call a bit position left or right. That is always ambiguous. Assuming that we will not change the existing names, the consistent name to use for \texttt{std::experimental::find\_first\_set} is \texttt{std::countr\_zero}. To put the new name into perspective take a look at Listing 7. My mental model is iterating the \texttt{string} from left to right, but on every SIMD chunk I need a twist and search right to left. I believe \texttt{countr\_zero} is an indefensible name for returning

\begin{verbatim}
int find(char needle, const std::string& hay) {
    using V = std::simd<char>;
    for (std::size_t i = 0; i + V::size <= hay.size(); i += V::size) {
        const V chunk(hay.data() + i);
        if (any_of(chunk == needle)) {
            return i + countr_zero(chunk == needle);
            // was: i + find\_first\_set(chunk == needle) in the TS
        }
    }
}
\end{verbatim}

Listing 7: SIMD algorithm to find a \texttt{char} in a \texttt{string}

6 cf. https://godbolt.org/z/ov78q1751
the lowest index where the mask is true. On the other hand, using countl_zero would be too inconsistent.

My recommendation is to overload count[lr]_zero/one for simd but not for simd_mask. Unless ...

5.2.1 the story of element-wise operations

Everything you can do with T you can do with simd<T>, only that the operations apply element-wise.

Since simd_mask<T> is the return type of simd<T> comparisons, the goal for simd_mask is:

Everything you can do with bool you can do with simd_mask<T>, only that the operations apply element-wise.

By that reasoning, popcount(simd_mask) should be element-wise popcount(bool), and countr_zero(simd_mask) should be element-wise countr_zero(bool). The <bit> functions would therefore not be mask reductions. If <bit> were extended to include bool overloads, there should also be simd_mask overloads, applying element-wise.

As a consequence, the mask reductions need a different name than the <bit> functions. See Table 1 for a suggestion. Note that if simd_mask becomes a std::ranges::input_range, then

<table>
<thead>
<tr>
<th>TS names</th>
<th>alternative names</th>
</tr>
</thead>
<tbody>
<tr>
<td>void f(std::native_simd_mask&lt;int&gt; k) {</td>
<td>void f(std::simd_mask&lt;int&gt; k) {</td>
</tr>
<tr>
<td>bool all = all_of(k);</td>
<td>bool all = reduce_and(k);</td>
</tr>
<tr>
<td>bool any = any_of(k);</td>
<td>bool any = reduce_or(k);</td>
</tr>
<tr>
<td>bool none = none_of(k);</td>
<td>bool none = reduce_nand(k);</td>
</tr>
<tr>
<td>int count = popcount(k);</td>
<td>int count = reduce_count(k);</td>
</tr>
<tr>
<td>int first = find_first_set(k);</td>
<td>int first = reduce_min_index(k);</td>
</tr>
<tr>
<td>int last = find_last_set(k);</td>
<td>int last = reduce_max_index(k);</td>
</tr>
</tbody>
</table>

Table 1: Mask reductions

std::ranges::all_of(simd_mask) will be well-formed and agree with the result of std::experimental::all_of. (Same for any_of and none_of.) The naming precedent suggests that all_of, any_of, and none_of should not be renamed.

My recommendation is to merge all_of, any_of, and none_of without change and rename popcount, find_first_set, and find_last_set according to Table 2.

5.2.2 remove some_of

While we’re at the topic. Let’s drop std::experimental::some_of. Nobody understands or wants the reduction, apparently. It can be added back later if really needed (unlikely).
5 Open questions

<table>
<thead>
<tr>
<th>TS name</th>
<th>IS name</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_of</td>
<td>all_of</td>
</tr>
<tr>
<td>any_of</td>
<td>any_of</td>
</tr>
<tr>
<td>none_of</td>
<td>none_of</td>
</tr>
<tr>
<td>popcount</td>
<td>reduce_count</td>
</tr>
<tr>
<td>find_first_set</td>
<td>reduce_min_index</td>
</tr>
<tr>
<td>find_last_set</td>
<td>reduce_max_index</td>
</tr>
</tbody>
</table>

Table 2: Recommended renaming of mask reductions

### 5.2.3 Suggested polls

Poll: Mask reductions should use the `<bit>` functions.

Poll: Rename mask reductions as proposed in P1928R3 Table 2.

Poll: Remove UB of `find_first_set` / `find_last_set`.

Poll: Do not merge / remove `some_of`.

### 5.3 Argument order and naming of masked overloads

In the TS, where-expressions made it possible to reuse existing function names and argument orders for masked operations. With the removal of where-expressions the mask must become a function argument. See Table 3 for a possible pattern to replace where-expressions.

There are more options, of course. E.g. possible replacements for `std::experimental::where_expression::copy_from`:

- `v.copy_from_if(v > 0, ptr)`

Here the condition directly follows the word "if", which seems helpful. However:
5.3.1 Suggested polls

Poll: The names of masked "overloads" should include an if and follow the argument order proposed in P1928R3 Section 5.3

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

5.4 Converting loads & stores consistency

For the TS, we allowed pointers to any vectorizable type as valid arguments to copy_from and copy_to. I.e. loads and stores can be converting operations without a clue in the code other than the type of the pointer. It can therefore happen that a conversion that is not value-preserving goes
unnoticed. The broadcast and `simd` conversion constructors guard against accidental use of such conversions.

I have not received feedback that users wrote buggy because of this liberal interface. However, in the TS process this question was never really considered. Therefore, I just wanted to show a suggestion for a stricter but just as powerful interface. Listing 8 presents converting broadcast and cast expressions, which are ill-formed because the type conversion is not value-preserving.

```cpp
float fmem[std::simd_size_v<float>] = {};
double dmem[std::simd_size_v<float>] = {};
short smem[std::simd_size_v<float>] = {};
std::simd<float> a = 1.; // ERROR: double -> float conversion is not value-preserving
std::simd<float> b = std::rebind_simd_t<int, std::simd<float>>(1); // ERROR:
    // int -> float is not value-preserving

// TS:
std::simd<float> ts;
ts.copy_from(fmem, std::element_aligned); // OK
ts.copy_from(dmem, std::element_aligned); // OK
ts.copy_from(smem, std::element_aligned); // OK

// idea, not status quo of this paper:
std::simd<float> v;
v.copy_from(fmem); // OK
v.copy_from(dmem); // ERROR: converting load
v.copy_from(smem); // ERROR: converting load

// Option (a) - one flag only:
v.copy_from(dmem, std::simd_converting); // OK
v.copy_from(smem, std::simd_converting); // OK

// Option (b) - two flags:
v.copy_from(dmem, std::simd_safe_cvt); // ERROR: double -> float is not value-preserving
v.copy_from(dmem, std::simd_any_cvt); // OK
v.copy_from(smem, std::simd_safe_cvt); // OK
v.copy_from(smem, std::simd_any_cvt); // OK
```

Listing 8: Load-store flags as opt-in to converting loads and stores

The equivalent conversions on `copy_from` are well-formed, though. I believe it would be better for users to opt-in to conversions on load and store. There are value-preserving and non-value-preserving conversions, which could be combined into the same opt-in or we provide a separate spelling for value-preserving conversions (the safe kind of conversion).

If LEWG is interested, I would be thankful for naming suggestions. I do not believe that using "safe" is a good term here.
5.5 **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 only a random-access range. * SIMD* can trivially model *input_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 *input_range*.

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.

5.6 **Correct Place for simd in the IS?**

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

I am seeking feedback before making a recommendation.

5.7 **element_reference is Overspecified**

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

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

6 **Wording**

The following section presents the wording to be applied against the C++ working draft. The subsequent section, Section 6.2, reproduces the same wording as a diff against the Parallelism TS 2.

The wording still needs work:

- Replace *where* & *where_expression* wording with *conditional_operator* and masked overloads.

- Apply the new library specification style from P0788R3.
6.1 Add section 9 of N4808 with modifications

---

Add a new subclause after §28.8 [numbers]

(6.1.1) 28.9 Data-Parallel Types

(6.1.1.1) 28.9.1 General

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

2 Throughout this Clause, the term data-parallel type refers to all supported (28.9.6.1) specializations of the simd and simd_mask class templates. A data-parallel object is an object of data-parallel type.

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

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

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

(6.1.1.2) 28.9.2 Header <simd> synopsis

```cpp
namespace std {
    namespace simd_abi {
        using scalar = see below;
        template<class T, size_t N> using fixed_size = see below;
        template<class T> inline constexpr size_t max_fixed_size = implementation-defined;
        template<class T> using native = implementation-defined;
        template<class T, size_t N, class... Abis> struct deduce { using type = see below; };
        template<class T, size_t N, class... Abis> using deduce_t = typename deduce<T, N, Abis...>::type;

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

        // 28.9.4, simd type traits
        template<class T> struct is_abi_tag;
    }
}
```

26
template<class T> inline constexpr bool is_abi_tag_v = is_abi_tag<T>::value;

template<class T> struct is_simd;

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

template<class T> struct is_simd_mask;

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

template<class T> struct is_simd_flag_type;

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

template<class T, class Abi = simd_abi::native<T>> struct simd_size;

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

template<class T, class U = typename T::value_type> struct memory_alignment;

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

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

template<class T, class V> using rebind_simd_t = typename rebind_simd<T, V>::type;

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

template<size_t N, class V> using resize_simd_t = typename resize_simd<N, V>::type;

// 28.9.6, Class template simd

template<class T, class Abi = simd_abi::native<T>> class simd;

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

// 28.9.8, Class template simd_mask

template<class T, class Abi = simd_abi::native<T>> class simd_mask;

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

constexpr tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
split(const simd<T, Abi>&) noexcept;

constexpr tuple<simd_mask<T, simd_mask_abi::deduce_t<T, Sizes>>...>
split(const simd_mask<T, Abi>&) noexcept;

constexpr array<V, simd_size_v<typename V::value_type, Abi> / V::size()>
split(const simd<typename V::value_type, Abi>&) noexcept;

constexpr array<V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>
split(const simd_mask<typename V::simd_type::value_type, Abi>&) noexcept;
template<size_t N, class T, class A>
constexpr array<resize_simd_t<simd_size_v<T, A> / N, simd<T, A>>, N>
split_by(const simd<T, A>& x) noexcept;

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

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

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

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

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

// 28.9.9.4, simd_mask reductions

// 28.9.5, Where expression class templates

// 28.9.9.5, Where functions
where(const typename simd<T, Abi>::mask_type&, simd<T, Abi>&) noexcept;

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

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

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

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

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

// 28.9.7.4, simd reductions

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

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

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

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

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

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

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

(6.1.1.3) 28.9.3 simd ABI tags

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

1 An ABI tag is a type in the std::simd_abi namespace that indicates a choice of size and binary representation for objects of data-parallel type. [Note: The intent is for the size and binary representation to depend on the target architecture. — end note ] The ABI tag, together with a given element type implies a number of elements. ABI tag types are used as the second template argument to simd and simd_mask.

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

3 scalar is an alias for an unspecified ABI tag that is different from fixed_size<1>. Use of the scalar tag type requires data-parallel types to store a single element (i.e., simd_size_v<T, simd_abi::scalar> equals 1).

4 The value of max_fixed_size<T> is at least 32.
fixed_size<N> is an alias for an unspecified ABI tag. fixed_size does not introduce a non-deduced context. Use of the simd_abi::fixed_size<N> tag type requires data-parallel types to store N elements (i.e. simd_size_v<T, simd_abi::fixed_size<N>> equals N). simd<T, fixed_size<N>> and simd_mask<T, fixed_size<N>> with N > 0 and N <= max_fixed_size<T> shall be supported. Additionally, for every supported simd<T, Abi> (see 28.9.6.1), where Abi is an ABI tag that is not a specialization of simd_abi::fixed_size, N == simd<T, Abi>::size() shall be supported. [Note: It is unspecified whether simd<T, fixed_size<N>> with N > max_fixed_size<T> is supported. The value of max_fixed_size<T> can depend on compiler flags and can change between different compiler versions.—end note]

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

An implementation may define additional extended ABI tag types in the std::simd_abi namespace, to support other forms of data-parallel computation.
	native<T> is an implementation-defined alias for an ABI tag. [Note: The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type T that is supported on the currently targeted system. For target architectures with ISA extensions, compiler flags may change the type of the native<T> alias.—end note]

[Example: Consider a target architecture supporting the extended ABI tags __simd128 and __simd256, where hardware support for __simd256 only exists for floating-point types. The implementation therefore defines native<T> as an alias for

- __simd256 if T is a floating-point type, and
- __simd128 otherwise.
—end example]

The member type shall be present if and only if

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

Where present, the member typedef type shall name an ABI tag type that satisfies

- simd_size<T, type> == N, and
- simd<T, type> is default constructible (see 28.9.6.1).

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

The behavior of a program that adds specializations for deduce is undefined.
template<class T> struct is_simd { see below };

3 The type is_simd<T> is a UnaryTypeTrait with a base characteristic of true_type if T is a specialization of the simd class template, and false_type otherwise.

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

template<class T> struct is_simd_mask { see below };

5 The type is_simd_mask<T> is a UnaryTypeTrait with a base characteristic of true_type if T is a specialization of the simd_mask class template, and false_type otherwise.

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

template<class T> struct is_simd_flag_type { see below };

7 The type is_simd_flag_type<class T> is a UnaryTypeTrait with a base characteristic of true_type if T is one of
   • element_aligned_tag, or
   • vector_aligned_tag, or
   • overaligned_tag<N> with N > 0 and N an integral power of two,
   and false_type otherwise.

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

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

9 simd_size<T, Abi> shall have a member value if and only if
   • T is a vectorizable type, and
   • is_abi_tag_v<Abi> is true.
   [ Note: The rules are different from those in (28.9.6.1): The member value is present even if simd<T, Abi>
     is not supported for the currently targeted system. — end note ]

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

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

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

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

13 If value is present, the type memory_alignment<T, U> is a BinaryTypeTrait with a base characteristic of integral_constant<size_t, N> for some implementation-defined N (see 28.9.6.5 and 28.9.8.4). [ Note: value identifies the alignment restrictions on pointers used for (converting) loads and stores for the give type T
     on arrays of type U. — end note ]

14 The behavior of a program that adds specializations for memory_alignment is undefined.
template<class T, class V> struct rebind_simd { using type = see below; 

15
The member type is present if and only if
• V is either simd<U, Abi0> or simd_mask<U, Abi0>, where U and Abi0 are deduced from V, and
• T is a vectorizable type, and
• simd_abi::deduce<T, simd_size_v<U, Abi0>, Abi0> has a member type.

16
Let Abi1 denote the type deduce_t<T, simd_size_v<U, Abi0>, Abi0>. Where present, the member typedef type names simd<T, Abi1> if V is simd<U, Abi0> or simd_mask<T, Abi1> if V is simd_mask<U, Abi0>.

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

17
The member type is present if and only if
• V is either simd<T, Abi0> or simd_mask<T, Abi0>, where T and Abi0 are deduced from V, and
• simd_abi::deduce<T, N, Abi0> has a member type.

18
Let Abi1 denote the type deduce_t<T, N, Abi0>. Where present, the member typedef type names simd<T, Abi1> if V is simd<T, Abi0> or simd_mask<T, Abi1> if V is simd_mask<T, Abi0>.

(6.1.1.5)
28.9.5 Where expression class templates

[6.1.1.5]
template<class M, class T> class const_where_expression {
    const M mask;  // exposition only
    T& data;       // exposition only

    public:
    const_where_expression(const const_where_expression&) = delete;
    const_where_expression& operator=(const const_where_expression&) = delete;

    T operator-() const & noexcept;
    T operator+(const T& x) const & noexcept;
    T operator~() const & noexcept;
    template<class U, class Flags = element_aligned_tag> void copy_to(U* mem, Flags f = {}) const & noexcept;
};

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

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

1 The class templates const_where_expression and where_expression abstract the notion of selecting elements of a
given object of arithmetic or data-parallel type.
2 The first templates argument \( M \) shall be cv-unqualified bool or a cv-unqualified simd_mask specialization.
3 If \( M \) is bool, \( T \) shall be a cv-unqualified arithmetic type. Otherwise, \( T \) shall either be \( M \) or typename \( M::\text{simd_type} \).
4 In this subclause, if \( M \) is bool, data[0] is used interchangeably for data, mask[0] is used interchangeably for mask, and
   \( M::\text{size()} \) is used interchangeably for 1.
5 The selected indices signify the integers \( i \in \{ j \in \mathbb{N} | j < M::\text{size()} \land \text{mask}[j] \} \). The selected elements signify
   the elements data[i] for all selected indices \( i \).
6 In this subclause, the type value_type is an alias for \( T \) if \( M \) is bool, or an alias for typename \( T::\text{value_type} \) if is_simd_mask_v<\( M \)>
   is true.
7 [Note: The where functions 28.9.9.5 initialize mask with the first argument to where and data with the second
   argument to where. — end note]

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

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

template<class U, class Flags = element_aligned_tag> void copy_to(U* mem, Flags = {}) const &&;

9 Requires:
   - If \( M \) is not bool, the largest selected index is less than the number of values pointed to by \( mem \).
   - If the template parameter Flags is vector_aligned_tag, \( mem \) shall point to storage aligned by memory_alignment_v<\( T, U >.
   - If the template parameter Flags is overaligned_tag<N>, \( mem \) shall point to storage aligned by \( N \).
   - If the template parameter Flags is element_aligned_tag, \( mem \) shall point to storage aligned by alignof(\( U \)).

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

Throws: Nothing.

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

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

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

**Remarks:** This operator shall not participate in overload resolution unless U is convertible to T.

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

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

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

```
void operator++() && noexcept;
void operator++(int) && noexcept;
void operator--() && noexcept;
void operator--(int) && noexcept;
```

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

**Remarks:** Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type T.

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

**Requires:**
- If is_simd_flag_type_v<U> is true, for all selected indices i, i shall be less than the number of values pointed to by mem.
- If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<T, U>.
- If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.
- If the template parameter Flags is element_aligned_tag, mem shall point to storage aligned by alignments(U).
Effects: Replaces the selected elements as if \( \text{data}[i] = \text{static_cast<value_type>(mem[i])} \) for all selected indices \( i \).

Throws: Nothing.

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

(6.1.1.6) 28.9.6 Class template \text{simd}  

(6.1.1.6.1) 28.9.6.1 Class template \text{simd} overview

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

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

  constexpr simd() noexcept = default;
  // 28.9.6.4, simd constructors
  template<class U> constexpr simd(U&& value) noexcept;
  template<class U, class UAbi> constexpr explicit(simd<U, UAbi>&) noexcept;
  template<class G> constexpr explicit simd(G&& gen) noexcept;
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  constexpr simd(const It& first, Flags = {});

  // 28.9.6.5, simd copy functions
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  constexpr void copy_from(const It& first, Flags f = {});
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  requires output_iterator<It, value_type>
  constexpr void copy_to(const It& first, Flags f = {}) const;

  // 28.9.6.6, simd subscript operators
  constexpr reference operator[](size_t);
  constexpr value_type operator[](size_t) const;

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

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

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

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

1 The class template `simd` is a data-parallel type. The width of a given `simd` specialization is a constant expression, determined by the template parameters.
Every specialization of `simd` is a complete type. The specialization `simd<T, Abi>` is supported if `T` is a vectorizable type and

- `Abi` is `simd_abi::scalar`, or
- `Abi` is `simd_abi::fixed_size<N>`, with `N` constrained as defined in 28.9.3.

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

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

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

[Example: Consider an implementation that defines the extended ABI tags `__simd_x` and `__gpu_y`. When the compiler is invoked to translate to a machine that has support for the `__simd_x` ABI tag for all arithmetic types other than `long double` and no support for the `__gpu_y` ABI tag, then:

- `simd<T, simd_abi::__gpu_y>` is not supported for any `T` and has a deleted constructor.
- `simd<long double, simd_abi::__simd_x>` is not supported and has a deleted constructor.
- `simd<double, simd_abi::__simd_x>` is supported.
- `simd<long double, simd_abi::scalar>` is supported.
—end example]

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

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

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

[Example: Consider an implementation that supports the type `__vec4f` and the function `__vec4f __vec4f_addsub(__vec4f, __vec4f)` for the currently targeted system. A user may require the use of `__vec4f_addsub` for maximum performance and thus writes:

```cpp
using V = simd<float, simd_abi::__simd128>;
V addsub(V a, V b) {
    return static_cast<V>(_vec4f_addsub(static_cast<__vec4f>(a), static_cast<__vec4f>(b))));
}
```

—end example]

### 6.1.1.6.2 `simd` width

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

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

---

(6.1.1.6.2) 28.9.6.2 `simd` width

[ simd.width]
28.9.6.3 Element references

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

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

```cpp
class reference // exposition only
{
public:
    reference() = delete;
    reference(const reference&) = delete;

    constexpr operator value_type() const noexcept;

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

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

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

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

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

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

5. Returns: A copy of *this.

6. Remarks: This function shall not participate in overload resolution unless declval<value_type&>() = std::forward<U>(x) is well-formed.
template<class U> constexpr reference operator+=(U&& x) && noexcept;
template<class U> constexpr reference operator-=(U&& x) && noexcept;
template<class U> constexpr reference operator*=(U&& x) && noexcept;
template<class U> constexpr reference operator/=(U&& x) && noexcept;
template<class U> constexpr reference operator%=(U&& x) && noexcept;
template<class U> constexpr reference operator|=(U&& x) && noexcept;
template<class U> constexpr reference operator&=(U&& x) && noexcept;
template<class U> constexpr reference operator^=(U&& x) && noexcept;
template<class U> constexpr reference operator<<=(U&& x) && noexcept;
template<class U> constexpr reference operator>>=(U&& x) && noexcept;

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

Returns: A copy of *this.

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

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

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

Returns: A copy of *this.

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

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

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

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

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

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

Effects: Exchanges the values a and b refer to.

(6.1.1.6.4) 28.9.6.4 simd constructors

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

[simd ctor]
Effects: Constructs an object with each element initialized to the value of the argument after conversion to value_type.

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

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

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

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

Remarks: This constructor shall not participate in overload resolution unless simd_size_v<U, UAbi> == size().

The constructor is explicit iff

- at least one possible value of U cannot be represented with type value_type, or
- if both U and value_type are integral types, the integer conversion rank (??) of U is greater than the integer conversion rank of value_type, or
- if both U and value_type are floating-point types, the floating-point conversion rank (??) of U is greater than the floating-point conversion rank of value_type.

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

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

Remarks: This constructor shall not participate in overload resolution unless simd(gen(integral_constant<size_t, i>())) is well-formed for all $i$ in the range of $[0, \text{size()}]$.

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

```cpp
template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr simd(const It& first, Flags = {});
```

Constraints: iter_value_t<tIt> is a vectorizable type.

Requires:

- [first, first + size()) is a valid range.
- If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<simd, U>.
- If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.
If the template parameter Flags is `element_aligned_tag`, mem shall point to storage aligned by `alignment(U)`.

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

**Remarks:** This constructor shall not participate in overload resolution unless
- `is_simd_flag_type_v<Flags>` is true.

---

### 28.9.6.5 simd copy functions

#### template<contiguous_iterator It, class Flags = element_aligned_tag>
```cpp
constexpr void copy_from(const It& first, Flags f = {});
```

**Constraints:** `iter_value_t<It>` is a vectorizable type.

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

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

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

---

#### template<contiguous_iterator It, class Flags = element_aligned_tag>
```cpp
requires output_iterator<It, value_type>
constexpr void copy_to(const It& first, Flags f = {}) const;
```

**Constraints:** `iter_value_t<It>` is a vectorizable type.

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

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

**Remarks:** This function shall not participate in overload resolution unless
- `is_simd_flag_type_v<Flags>` is true.
(6.1.6.6) 28.9.6.6 simd subscript operators

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

1 Requires: \( i < \text{size}() \).
2 Returns: A reference (see 28.9.6.3) referring to the \( i \)th element.
3 Throws: Nothing.

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

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

(6.1.6.7) 28.9.6.7 simd unary operators

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

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

2 Effects: Increments every element by one.
3 Returns: *this.

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

4 Effects: Increments every element by one.
5 Returns: A copy of *this before incrementing.

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

6 Effects: Decrements every element by one.
7 Returns: *this.

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

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

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

10 Returns: A simd_mask object with the \( i \)th element set to \( \lnot \operator[](i) \) for all \( i \) in the range of \([0, \text{size}())\).
constexpr simd operator~() const noexcept;

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

Remarks: This operator shall not participate in overload resolution unless T is an integral type.

cconstexpr simd operator+() const noexcept;

Returns: *this.

cconstexpr simd operator-() const noexcept;

Returns: A simd object where the ith element is initialized to -operator[]() for all i in the range of [0, size()).

(6.1.1.7) 28.9.7 simd non-member operations [simd.nonmembers]

(6.1.1.7.1) 28.9.7.1 simd binary operators [simd.binary]

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

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

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

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

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

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

(6.1.1.7.2) 28.9.7.2 simd compound assignment [simd.cassign]
friend constexpr simd& operator+=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator-=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator*=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator/=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator%=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator&=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator|=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator^=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator<<=(simd& lhs, const simd& rhs) noexcept;
friend constexpr simd& operator>>=(simd& lhs, const simd& rhs) noexcept;

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

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

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

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

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

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

(6.1.1.7.3) 28.9.7.3 simd compare operators

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

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

(6.1.1.7.4) 28.9.7.4 simd reductions

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

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

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

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

Throws: Any exception thrown from binary_op.
template<class M, class V, class BinaryOperation>
constexpr typename V::value_type reduce(const const_where_expression<M, V>& x, typename V::value_type identity_element, BinaryOperation binary_op = {});

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

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

7 Throws: Any exception thrown from binary_op.

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

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

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

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

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

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

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

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

15 Returns: The value of an element x[j] for which x[j] <= x[i] for all i in the range of [0, size()).
template<class M, class V> constexpr typename V::value_type hmin(const const_where_expression<M, V>& x) noexcept;

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

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

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

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

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

(6.1.1.7.5) 28.9.7.5 Casts

[simd.casts]

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

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

Returns: A tuple of data-parallel objects with the $i$th simd/simd_mask element of the $j$th tuple element initialized to the value of the element $x$ with index $i + $ sum of the first $j$ values in the Sizes pack.

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

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

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

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

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

- $is_simd_v<V>$ is true and $simd_size_v<
type_name V::value_type, Abi> is an integral multiple of $V::size()$, or
- $is_simd_mask_v<V>$ is true and $simd_size_v<
type_name V::simd_type::value_type, Abi> is an integral multiple of $V::size()$. 
template<
    size_t N,
    class T,
    class A>
constexpr
array<
    resize_simd<
        simd_size_v<T, A> / N,
        simd<T, A>>
    N>
split_by(const simd<T, A>& x) noexcept;

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

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

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

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

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

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

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

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

Returns: A data-parallel object, the ith element of which is initialized by arr[i / simd_size_v<T, Abi>][i % simd_size_v<T, Abi>].

(6.1.1.7.6) 28.9.7.6 Algorithms

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

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

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

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

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

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

\[
\text{template}<\text{class } T, \text{ class } Abi> \text{ simd}\langle T, Abi\rangle
\]

\[
\text{constexpr } \text{ clamp(} \text{ const simd}\langle T, Abi\rangle & v, \text{ const simd}\langle T, Abi\rangle & lo, \text{ const simd}\langle T, Abi\rangle & hi); \]

\(4\) \textit{Requires:} No element in \( lo \) shall be greater than the corresponding element in \( hi \).

\(5\) \textit{Returns:} The result of element-wise application of \( \text{std::clamp}(v[i], lo[i], hi[i]) \) for all \( i \) in the range of \([0, \text{size}())\).

\(6.1.1.7.7\) \textbf{28.9.7.7} simd math library \[\text{simd.math}\]

1 For each set of overloaded functions within \(<\text{cmath}>\), there shall be additional overloads sufficient to ensure that if any argument corresponding to a \texttt{double} parameter has type \texttt{simd}\langle T, Abi\rangle, where \texttt{is\textunderscore floating\textunderscore point\_v<T>} is true, then:

• All arguments corresponding to \texttt{double} parameters shall be convertible to \texttt{simd}\langle T, Abi\rangle.

• All arguments corresponding to \texttt{double\_} parameters shall be of type \texttt{simd}\langle T, Abi\rangle.\_\star.

• All arguments corresponding to parameters of integral type \( U \) shall be convertible to \texttt{fixed\_size\_simd}\langle U, \text{simd\_size\_v<T, Abi}>\rangle.

• All arguments corresponding to \( U\_\star \), where \( U \) is integral, shall be of type \texttt{fixed\_size\_simd}\langle U, \text{simd\_size\_v<T, Abi}>\rangle.\_\star.

• If the corresponding return type is \texttt{double}, the return type of the additional overloads is \texttt{simd}\langle T, Abi\rangle. Otherwise, if the corresponding return type is \texttt{bool}, the return type of the additional overload is \texttt{simd\_mask}\langle T, Abi\rangle. Otherwise, the return type is \texttt{fixed\_size\_simd}\langle R, \text{simd\_size\_v<T, Abi}>\rangle, with \( R \) denoting the corresponding return type.

It is unspecified whether a call to these overloads with arguments that are all convertible to \texttt{simd}\langle T, Abi\rangle but are not of type \texttt{simd}\langle T, Abi\rangle is well-formed.

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

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

4 TODO: Allow abs\langle\texttt{simd\langle signed\_integral\rangle}\rangle.

5 If abs is called with an argument of type \texttt{simd\langle X, Abi\rangle} for which \texttt{is\textunderscore unsigned\_v<X>} is true, the program is ill-formed.

\(6.1.1.8\) \textbf{28.9.8} Class template \texttt{simd\_mask} \[\text{simd.mask.class}\]

\(6.1.1.8.1\) \textbf{28.9.8.1} Class template \texttt{simd\_mask} overview \[\text{simd.mask.overview}\]
template<class T, class Abi> class simd_mask {
public:
  using value_type = bool;
  using reference = see below;
  using simd_type = simd<T, Abi>;
  using abi_type = Abi;

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

  constexpr simd_mask() noexcept = default;

  // 28.9.8.3, simd_mask constructors
  constexpr explicit simd_mask(value_type) noexcept;
  template<class U, class UAbi>
  constexpr explicit(sizeof(U) != sizeof(T)) simd_mask(const simd_mask<U, UAbi>&) noexcept;
  template<class G> constexpr explicit simd_mask(G&& gen) noexcept;
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  constexpr simd_mask(const It& first, Flags = {});

  // 28.9.8.4, simd_mask copy functions
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  constexpr void copy_from(const It& first, Flags = {});
  template<contiguous_iterator It, class Flags = element_aligned_tag>
  requires output_iterator<It, value_type>
  constexpr void copy_to(const It& first, Flags = {}) const;

  // 28.9.8.5, simd_mask subscript operators
  constexpr reference operator[](size_t);
  constexpr value_type operator[](size_t) const;

  // 28.9.8.6, simd_mask unary operators
  constexpr simd_mask operator!() const noexcept;

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

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

  // 28.9.9.3, simd_mask comparisons
}
friend constexpr simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
friend constexpr simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;
};

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

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

- Abi is simd_abi::scalar, or
- Abi is simd_abi::fixed_size<N>, with N constrained as defined in (28.9.3).

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

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

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

Default initialization performs no initialization of the elements; value-initialization initializes each element with false. [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 simd_mask:

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

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

(28.9.6.3)

(6.1.8.2) 28.9.8.2 simd_mask width

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

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

(6.1.8.3) 28.9.8.3 simd_mask constructors

constexpr explicit simd_mask(value_type x) noexcept;

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

template<class U, class UAbi>
constexpr explicit(sizeof(U) != sizeof(T)) simd_mask(const simd_mask<U, UAbi>& x) noexcept;
Effects: Constructs an object of type `simd_mask` where the \(i\)th element equals \(x[i]\) for all \(i\) in the range of \([0, \text{size()}]\).

Remarks: This constructor shall not participate in overload resolution unless \(\text{simd_size_v}<\text{U}, \text{UAbi}> == \text{size()}\).

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

Effects: Constructs an object where the \(i\)th element is initialized to `gen(integral_constant<size_t, i>())`.

Remarks: This constructor shall not participate in overload resolution unless `static_cast<bool>(gen(integral_constant<size_t, i>()))` is well-formed for all \(i\) in the range of \([0, \text{size()}]\).

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

```cpp
template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr simd_mask(const It& first, Flags = {});
```

Constraints: `iter_value_t<It>` is of type `bool`.

Requires:

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

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

Throws: Nothing.

Remarks: This constructor shall not participate in overload resolution unless `is_simd_flag_type_v<Flags>` is true.
Effects: Replaces the elements of the simd_mask object such that the $i^{th}$ element is replaced with $\text{mem}[i]$ for all $i$ in the range of $[0, \text{size}())$.

Throws: Nothing.

Remarks: This function shall not participate in overload resolution unless $\text{is_simd_flag_type}_v<\text{Flags}>$ is true.

template<\text{contiguous_iterator} \text{It}, \text{class Flags} = \text{element_aligned_tag}>
\text{requires output_iterator<It, value_type>}
\text{constexpr void copy_to(const It& first, Flags = {}) const;}

Constraints: iter_value_t<It> is of type bool.

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

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

Throws: Nothing.

Remarks: This function shall not participate in overload resolution unless $\text{is_simd_flag_type}_v<\text{Flags}>$ is true.

(6.1.1.8.5) 28.9.8.5 simd_mask subscript operators

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

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

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

Throws: Nothing.

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

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

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

Throws: Nothing.

(6.1.1.8.6) 28.9.8.6 simd_mask unary operators

\text{constexpr simd_mask operator!() const noexcept;}

Returns: The result of the element-wise application of operator!.
(6.1.1.9) 28.9.9 Non-member operations

28.9.9.1 simd_mask binary operators

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

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

(6.1.1.9.2) 28.9.9.2 simd_mask compound assignment

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

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

(6.1.1.9.3) 28.9.9.3 simd_mask comparisons

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

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

(6.1.1.9.4) 28.9.9.4 simd_mask reductions

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

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

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

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

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

3 Returns: true if none of the one boolean elements in k is true, false otherwise.
template<class T, class Abi> constexpr bool some_of(const simd_mask<T, Abi>& k) noexcept;

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

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

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

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

Requires: any_of(k) returns true.

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

Throws: Nothing.

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

Requires: any_of(k) returns true.

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

Throws: Nothing.

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

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

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

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

Requires: The value of the argument is true.

Returns: 0.

Throws: Nothing.

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

(6.1.1.9.5) 28.9.9.5 where functions [simd.mask.where]
template<class T, class Abi>
    where_expression<simd_mask<T, Abi>, simd<T, Abi>>
    where(const typename simd<T, Abi>::mask_type& k, simd<T, Abi>& v) noexcept;

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

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

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

1 Returns: An object (28.9.5) with mask and data initialized with k and v respectively.

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

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

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

3 Returns: An object (28.9.5) with mask and data initialized with k and v respectively.

6.2 DIFF AGAINST PARALLELISM TS 2 (N4808)
In the following, the wording from Section 6.1 is repeated with additional indications of differ-
ences with regard to N4808. Changes relative to N4808, which contains editorial changes after
the publication of the TS, are marked using color for additions and removals.

(6.2.1) 28.9 Data-Parallel Types

(6.2.1.1) 28.9.1 General

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

2 Throughout this Clause, the term data-parallel type refers to all supported (28.9.6.1) specializations of the simd
and simd_mask class templates. A data-parallel object is an object of data-parallel type.
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.

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

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

(6.2.1.2) 28.9.2 Header <experimental/simd> synopsis

```cpp
namespace std::experimental { inline namespace parallelism_v2 {
    namespace simd_abi {
        using scalar = see below;
        template<class T, int size_t N> using fixed_size = see below;
        template<class T> inline constexpr int size_t max_fixed_size = implementation-defined;
        template<class T> using compatible = implementation-defined;
        template<class T> using native = implementation-defined;

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

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

    // 28.9.4, simd type traits
    template<class T> struct is_abi_tag;
    template<class T> inline constexpr bool is_abi_tag_v = is_abi_tag<T>::value;

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

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

    template<class T> struct is_simd_flag_type;
    template<class T> inline constexpr bool is_simd_flag_type_v = is_simd_flag_type<T>::value;
}
```
template<class T, class Abi = simd_abi::compatible_abi<T>> struct simd_size;
inline constexpr size_t simd_size_v = simd_size<T,Abi>::value;

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

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

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

// 28.9.6, Class template simd
template<class T, class Abi = simd_abi::compatible_abi<T>> class simd;
template<class T> using native_simd = simd<T, simd_abi::native<T>>;
template<class T, int size_t N> using fixed_size_simd = simd<T, simd_abi::fixed_size<T, N>>;

// 28.9.8, Class template simd_simd
template<class T, class Abi = simd_abi::compatible_abi<T>> class simd;
template<class T> using native_simd_simd = simd_simd<T, simd_abi::native<T>>;
template<class T, int size_t N> using fixed_size_simd_simd = simd_simd<T, simd_abi::fixed_size<T, N>>;

// 28.9.7.5, Casts
template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below static simd_cast(const simd<U, Abi>&) noexcept;

template<class T, class Abi>
fixed_size_simd<T, simd_size_v<T, Abi>> to_fixed_size(const simd<T, Abi>&) noexcept;
template<class T, class Abi>
fixed_size_simd_mask<T, simd_size_v<T, Abi>> to_fixed_size(const simd_mask<T, Abi>&) noexcept;

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

template<class T, int N>
simd<T> to_compatible(const fixed_size_simd<T, N>&) noexcept;
template<class T, int N>
simd_mask<T> to_compatible(const fixed_size_simd_mask<T, N>&) noexcept;

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

constexpr tuple<simd_mask<T>, simd_mask_abi::deduce_t<T, Sizes>>...
    split(const simd_mask<T, Abi>&) noexcept;

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

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

template<size_t N, class T, class A>
constexpr array<resize_simd_t<
    simd_size_v<T, A>> * N, simd<T, A>> split_by(const array<simd<T, A>, N>& arr) noexcept;

template<size_t N, class T, class A>
constexpr array<resize_simd_t<
    simd_size_v<T, A>> * N, simd_mask<T, A>> split_by(const array<simd_mask<T, A>, N>& arr) noexcept;

// 28.9.9.4, Rsimd_mask reductions
template<class T, class Abi> constexpr bool all_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr bool any_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr bool none_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr bool some_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr int popcount(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> constexpr int find_first_set(const simd_mask<T, Abi>&);
template<class T, class Abi> constexpr int find_last_set(const simd_mask<T, Abi>&);

constexpr bool all_of(T) noexcept;
constexpr bool any_of(T) noexcept;
constexpr bool none_of(T) noexcept;
constexpr bool some_of(T) noexcept;
constexpr int popcount(T) noexcept;
constexpr int find_first_set(T);
constexpr int find_last_set(T);
// 28.9.5. Where expression class templates
template<class M, class T> class const_where_expression;
template<class M, class T> class where_expression;

// 28.9.9.5. Where functions
template<class T, class Abi>
where_expression<simd_mask<T, Abi>, simd<T, Abi>>
where(const typename simd<T, Abi>::mask_type&,
      simd<T, Abi>&) noexcept;

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

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

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

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

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

// 28.9.7.4. R SIMD reductions
template<class T, class Abi, class BinaryOperation = plus>
constexpr T reduce(const simd<T, Abi>&,
                    BinaryOperation = {});

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

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

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

template<class M, class V>

```cpp
constexpr typename V::value_type reduce(const const_where_expression<M, V>& x, 
  bit_and<> binary_op) noexcept;

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

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

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

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

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

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

// 28.9.7.6, Algorithms

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

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

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

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

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

(6.2.1.3) 28.9.3 simd ABI tags

namespace simd_abi {
  using scalar = see below;
  template<class T, innate_t N> using fixed_size = see below;
  template<class T> inline constexpr innate_t max_fixed_size = implementation-defined;
  template<class T> using compatible = implementation-defined;
  template<class T> using native = implementation-defined;
}
An ABI tag is a type in the std::experimental::parallelism_v2::simd_abi namespace that indicates a choice of size and binary representation for objects of data-parallel type. [Note: The intent is for the size and binary representation to depend on the target architecture. — end note] The ABI tag, together with a given element type implies a number of elements. ABI tag types are used as the second template argument to simd and simd_mask.

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

scalar is an alias for an unspecified ABI tag that is different from fixed_size<1>. Use of the scalar tag type requires data-parallel types to store a single element (i.e., simd<T, simd_abi::scalar>::size() returns simd_size_v<T, simd_abi::scalar> equals 1).

The value of max_fixed_size<T> is at least 32.

fixed_size<N> is an alias for an unspecified ABI tag. fixed_size does not introduce a non-deduced context. Use of the simd_abi::fixed_size<N> tag type requires data-parallel types to store N elements (i.e. simd<T, simd_abi::fixed_size<N> >::size(N). simd<T, fixed_size<N>> and simd_mask<T, fixed_size<N>> with N > 0 and N <= max_fixed_size<T> shall be supported. Additionally, for every supported simd<T, Abi> (see 28.9.6.1), where Abi is an ABI tag that is not a specialization of simd_abi::fixed_size, N == simd<T, Abi>::size() shall be supported.

[Note: It is unspecified whether simd<T, fixed_size<N>> with N > max_fixed_size<T> is supported. The value of max_fixed_size<T> can depend on compiler flags and can change between different compiler versions. — end note]

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

[Note: An implementation can forego ABI compatibility between differently compiled translation units for simd and simd_mask specializations using the same simd_abi::fixed_size<T> tag. Otherwise, the efficiency of simd<T, Abi> is likely to be better than for simd<T, fixed_size<simd_size_v<T, simd_abi::fixed_size<T>> equals N> with Abi not a specialization of simd_abi::fixed_size). — end note]

An implementation may define additional extended ABI tag types in the std::experimental::parallelism_v2::simd_abi namespace, to support other forms of data-parallel computation.

compatible<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 ensures ABI compatibility between translation units on the target architecture. — end note] [Example: Consider a target architecture supporting the extended ABI tags __simd128 and __simd256, where the __simd256 type requires an optional ISA extension on said architecture. Also, the target architecture does not support long double with either ABI tag. The implementation therefore defines compatible<T> as an alias for:

- scalar if T is the same type as long double, and
- __simd128 otherwise.

— end example]

native<T> is an implementation-defined alias for an ABI tag. [Note: The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type T that is supported on the currently targeted system. For target architectures without ISA extensions, the native<T> and compatible<T> aliases will likely be the same. For target architectures with ISA extensions, compiler flags may influence change the type of the native<T> alias, while compatible<T> will be the same independent of such flags. — end note] [Example: Consider a target architecture supporting the extended ABI tags __simd128 and __simd256, where hardware support for __simd256 only exists for floating-point types. The implementation therefore defines native<T> as an alias for:

- __simd256 if T is a floating-point type, and

62
template<T, size_t N, class... Abis> struct deduce { using type = __simd128 otherwise. — end example }

The member type shall be present if and only if
- T is a vectorizable type, and
- simd_abi::fixed_size<N> is supported (see 28.9.3), and
- every type in the Abis pack is an ABI tag.

Where present, the member typedef type shall name an ABI tag type that satisfies
- simd_size<T, type> == N, and
- simd<T, type> is default constructible (see 28.9.6.1).

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

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

6.2.1.4 28.9.4 simd type traits

template<class T> struct is_abi_tag { see below };  

The type is_abi_tag<T> is a UnaryTypeTrait with a base characteristic of true_type if T is a standard or extended ABI tag, and false_type otherwise.

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

template<class T> struct is_simd { see below };  

The type is_simd<T> is a UnaryTypeTrait with a base characteristic of true_type if T is a specialization of the simd class template, and false_type otherwise.

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

template<class T> struct is_simd_mask { see below };  

The type is_simd_mask<T> is a UnaryTypeTrait with a base characteristic of true_type if T is a specialization of the simd_mask class template, and false_type otherwise.

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

template<class T> struct is_simd_flag_type { see below };  

The type is_simd_flag_type<class T> is a UnaryTypeTrait with a base characteristic of true_type if T is one of
- element_aligned_tag, or
• \texttt{vector\_aligned\_tag}, or
• \texttt{overaligned\_tag<\textbackslash N>} with \( \texttt{N} > \texttt{B} \) and \( \texttt{N} \) an integral power of two, and \texttt{false\_type} otherwise.

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

\begin{verbatim}
template<class T, class Abi = simd\_abi:: compatibility\_native<T>> struct simd\_size { see below }

9  simd\_size<T, Abi> shall have a member \texttt{value} if and only if
• \texttt{T} is a vectorizable type, and
• \texttt{is\_abi\_tag\_v<Abi>} is true.

\end{verbatim}

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

If \texttt{value} is present, the type \texttt{simd\_size<T, Abi>} is a \texttt{BinaryTypeTrait} with a base characteristic of \texttt{integral\_constant<\texttt{size\_t, N}>} with \( \texttt{N} \) equal to the number of elements in a \texttt{simd<T, Abi>} object. \[\textit{Note: If simd<T, Abi> is not supported for the currently targeted system, simd\_size<T, Abi>::value produces the value simd<T, Abi>::\texttt{size}() would return if it were supported. — end note}\]

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

\begin{verbatim}
template<class T, class U = typename T::value\_type> struct memory\_alignment { see below }

12  memory\_alignment<T, U> shall have a member \texttt{value} if and only if
• \texttt{is\_simd\_mask\_v<T>} is true and \texttt{U} is \texttt{bool}, or
• \texttt{is\_simd\_v<T>} is true and \texttt{U} is a vectorizable type.

\end{verbatim}

If \texttt{value} is present, the type \texttt{memory\_alignment<T, U>} is a \texttt{BinaryTypeTrait} with a base characteristic of \texttt{integral\_constant<\texttt{size\_t, N}>} for some implementation-defined \( \texttt{N} \) (see 28.9.6.5 and 28.9.8.4). \[\textit{Note: value identifies the alignment restrictions on pointers used for (converting) loads and stores for the give type \texttt{T} on arrays of type \texttt{U}. — end note}\]

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

\begin{verbatim}
template<class T, class V> struct rebind\_simd { using type = see below ; }

15  The member \texttt{type} is present if and only if
• \texttt{V} is either \texttt{simd<U, Abi>} or \texttt{simd\_mask<U, Abi>}, where \texttt{U} and \texttt{Abi} are deduced from \texttt{V}, and
• \texttt{T} is a vectorizable type, and
• \texttt{simd\_abi::\texttt{deduce<T, simd\_size\_v<U, Abi>}, Abi>} has a member type \texttt{type}.

\end{verbatim}

Let \texttt{Abi1} denote the type \texttt{deduce\_t<T, simd\_size\_v<U, Abi>}, Abi>. Where present, the member typedef \texttt{name}s \texttt{simd<T, Abi1>} if \texttt{V} is \texttt{simd<U, Abi> or simd\_mask<T, Abi>}, \texttt{Abi1>} if \texttt{V} is \texttt{simd\_mask<U, Abi>}.
• simd_abi::deduce<T, N, Abi0> has a member type type.

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

(6.2.1.5) 28.9.5 Where expression class templates

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

    public:
        const_where_expression(const const_where_expression&) = delete;
        const_where_expression& operator=(const const_where_expression&) = delete;

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

        template<class U, class Flags = element_aligned_tag> void copy_to(U* mem, Flags f = {}) const & & noexcept;

    };

template<class M, class T>
class where_expression : public const_where_expression<M, T> {

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

        template<class U, class Flags = element_aligned_tag> void copy_from(const U* mem, Flags f = {}) & & noexcept;

};//
```

1 The class templates const_where_expression and where_expression abstract the notion of selecting elements of a given object of arithmetic or data-parallel type.

2 The first templates argument M shall be cv-unqualified bool or a cv-unqualified simd_mask specialization.
If \( M \) is \( \text{bool} \), \( T \) shall be a cv-unqualified arithmetic type. Otherwise, \( T \) shall either be \( M \) or \text{typename } \( M::\text{simd_type} \).

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

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

In this subclause, the type \text{value_type} \ is an alias for \( T \) if \( M \) is \( \text{bool} \), or an alias for \text{typename } \( T::\text{value_type} \) if \text{is_simd_mask_v<M>} \ is \text{true}.

[ Note: The \text{where} functions 28.9.9.5 initialize \text{mask} \ with the first argument to \text{where} \ and \text{data} \ with the second argument to \text{where}. — end note ]

\[
\begin{align*}
\text{T} & \text{ operator-()} \text{ const & noexcept; } \\
\text{T} & \text{ operator+()} \text{ const & noexcept; } \\
\text{T} & \text{ operator-()} \text{ const & noexcept; }
\end{align*}
\]

Returns: A copy of \text{data} with the indicated unary operator applied to all selected elements.

\[
\begin{align*}
\text{template<class } U, \text{ class Flags = element_aligned_tag> void copy_to(U* mem, Flags = {})} & \text{ const & noexcept; } \\
\text{Requires: } & \\
& \bullet \text{ If } M \text{ is not } \text{bool}, \text{ the largest selected index is less than the number of values pointed to by } \text{mem}. \\
& \bullet \text{ If the template parameter } \text{Flags} \text{ is vector_aligned_tag, } \text{mem} \text{ shall point to storage aligned by } \text{memory_alignment_v<T, U>}. \\
& \bullet \text{ If the template parameter } \text{Flags} \text{ is overaligned_tag<N>, } \text{mem} \text{ shall point to storage aligned by } N. \\
& \bullet \text{ If the template parameter } \text{Flags} \text{ is element_aligned_tag, } \text{mem} \text{ shall point to storage aligned by } \text{alignof(U)}. \\
\text{Effects: Copies the selected elements as if } \text{mem}[i] = \text{static_cast<U>(data[i])} \text{ for all selected indices } i. \\
\text{Thros:} & \\
\text{Nothing.} \\
\text{Remarks: This function shall not participate in overload resolution unless } & \\
& \bullet \text{ is_simd_flag_type_v<Flags> is } \text{true}, \text{ and } \\
& \bullet \text{ either } \\
& \quad - \text{ U is } \text{bool} \text{ and } \text{value_type} \text{ is } \text{bool}, \text{ or } \\
& \quad - \text{ U is a vectorizable type and } \text{value_type} \text{ is not } \text{bool}. \\
\text{template<class } U> \text{ void operator=(U&& x) & noexcept; } \\
\text{Effects: Replaces } \text{data}[i] \text{ with } \text{static_cast<T>(std::forward<U>(x))[i]} \text{ for all selected indices } i. \\
\text{Remarks: This operator shall not participate in overload resolution unless } u \text{ is convertible to } T.
\]

\[
\begin{align*}
\text{template<class } U> \text{ void operator==(U&& x) & noexcept; } \\
\text{template<class } U> \text{ void operator!=(U&& x) & noexcept; } \\
\text{template<class } U> \text{ void operator<=(U&& x) & noexcept; } \\
\text{template<class } U> \text{ void operator>=(U&& x) & noexcept; } \\
\text{template<class } U> \text{ void operator<<=(U&& x) & noexcept; } \\
\text{template<class } U> \text{ void operator>=(U&& x) & noexcept; }
\end{align*}
\]
template<class U> void operator|=(U&& x) && noexcept;
template<class U> void operator^=(U&& x) && noexcept;
template<class U> void operator<<=(U&& x) && noexcept;
template<class U> void operator>>=(U&& x) && noexcept;

Effects: Replaces data[i] with static_cast<T>(data @ std::forward<U>(x))[i] (where @ denotes the indicated operator) for all selected indices i.
Remarks: Each of these operators shall not participate in overload resolution unless the return type of data @ std::forward<U>(x) is convertible to T. It is unspecified whether the binary operator, implied by the compound assignment operator, is executed on all elements or only on the selected elements.

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

Effects: Applies the indicated operator to the selected elements.
Remarks: Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type T.

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

Requires:
• If is_simd_flag_type_v<U> is true, for all selected indices i, i shall be less than the number of values pointed to by mem.
• If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<T, U>.
• If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.
• If the template parameter Flags is element_aligned_tag, mem shall point to storage aligned by alignof(U).

Effects: Replaces the selected elements as if data[i] = static_cast<value_type>(mem[i]) for all selected indices i.
Throws: Nothing.
Remarks: This function shall not participate in overload resolution unless
• is_simd_flag_type_v<Flags> is true, and
• either
  - U is bool and value_type is bool, or
  - U is a vectorizable type and value_type is not bool.
template<class T, class Abi> class simd {
public:
    using value_type = T;
    using reference = see below;
    using mask_type = simd_mask<T, Abi>;
    using abi_type = Abi;
    static constexpr size_t size() noexcept = default;

    // 28.9.6.4, simd constructors
    template<class U> constexpr simd(U&& value) noexcept;
    template<class U, class UAbi> constexpr simd(const simd<U, simd::fixed_size<size_t>>& UAbi) noexcept;
    template<class G> constexpr simd(G&& gen) noexcept;
    template<class G, class contiguous_iterator It, class Flags = element_aligned_tag> constexpr simd(const U* mem, It& first, Flags f = {});

    // 28.9.6.5, simd copy functions
    template<class G, class contiguous_iterator It, class Flags = element_aligned_tag> constexpr void copy_from(const U* mem, It& first, Flags f = {});
    template<class G, class contiguous_iterator It, class Flags = element_aligned_tag> requires output_iterator<It, value_type> constexpr void copy_to(U* mem, const It& first, Flags f = {})

    // 28.9.6.6, simd subscript operators
    constexpr reference operator[](size_t);
    constexpr value_type operator[](size_t) const;

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

    // 28.9.7.1, simd binary operators
    friend constexpr simd operator+(const simd&, const simd&) noexcept;
    friend constexpr simd operator-(const simd&, const simd&) noexcept;
    friend constexpr simd operator*(const simd&, const simd&) noexcept;
    friend constexpr simd operator/(const simd&, const simd&) noexcept;
    friend constexpr simd operator%(const simd&, const simd&) noexcept;
}
// 28.9.7.2, simd compound assignment
friend constexpr simd operator=(simd, const simd&) noexcept;
friend constexpr simd operator+=(simd, const simd&) noexcept;
friend constexpr simd operator-=(simd, const simd&) noexcept;
friend constexpr simd operator*=(simd, const simd&) noexcept;
friend constexpr simd operator/=(simd, const simd&) noexcept;
friend constexpr simd operator%=(simd, const simd&) noexcept;
friend constexpr simd operator&=(simd, const simd&) noexcept;
friend constexpr simd operator|=(simd, const simd&) noexcept;
friend constexpr simd operator^=(simd, const simd&) noexcept;
friend constexpr simd operator<<=(simd, const simd&) noexcept;
friend constexpr simd operator>>=(simd, const simd&) noexcept;
friend constexpr simd operator<<=(simd, int) noexcept;
friend constexpr simd operator>>=(simd, int) noexcept;
friend constexpr simd& operator+=(simd&, const simd&) noexcept;
friend constexpr simd& operator-=(simd&, const simd&) noexcept;
friend constexpr simd& operator*=(simd&, const simd&) noexcept;
friend constexpr simd& operator/=(simd&, const simd&) noexcept;
friend constexpr simd& operator%=(simd&, const simd&) noexcept;
friend constexpr simd& operator&=(simd&, const simd&) noexcept;
friend constexpr simd& operator|=(simd&, const simd&) noexcept;
friend constexpr simd& operator^=(simd&, const simd&) noexcept;
friend constexpr simd& operator<<=(simd&, const simd&) noexcept;
friend constexpr simd& operator>>=(simd&, const simd&) noexcept;
friend constexpr simd& operator<<=(simd&, int) noexcept;
friend constexpr simd& operator>>=(simd&, int) noexcept;

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

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

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

- Abi is simd_abi::scalar, or
- Abi is simd_abi::fixed_size<N>, with N constrained as defined in 28.9.3.

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

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

- is_nothrow_move_constructible_v<simd<T, Abi>>, and
- is_nothrow_move_assignable_v<simd<T, Abi>>, and
- is_nothrow_default_constructible_v<simd<T, Abi>>.
Example: Consider an implementation that defines the extended ABI tags __simd_x and __gpu_y. When the compiler is invoked to translate to a machine that has support for the __simd_x ABI tag for all arithmetic types other than long double and no support for the __gpu_y ABI tag, then:

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

— end example —

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

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

```cpp
constr.expr explicit operator implementation-defined() const;
constr.expr explicit simd(const implementation-defined & init);
```

Example: Consider an implementation that supports the type __vec4f and the function __vec4f __vec4f_addsub(__vec4f, __vec4f) for the currently targeted system. A user may require the use of __vec4f_addsub for maximum performance and thus writes:

```cpp
using V = simd<float, simd_abi::__simd128>
V addsub(V a, V b) {
    return static_cast<V>(_vec4f_addsub(static_cast<__vec4f>(a), static_cast<__vec4f>(b)));
}
```

— end example —

(6.2.1.6.2) 28.9.6.2 `simd` width

```cpp
static constexpr size_t size() noexcept

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

(6.2.1.6.3) 28.9.6.3 Element references

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

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

```cpp
class reference // exposition only
{
public:
    reference() = delete;
    reference(const reference&) = delete;

    constr.expr operator value_type() const noexcept;
```
template<class U> constexpr reference operator=(U&& x) && noexcept;

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

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

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

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

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

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

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

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

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

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

constexpr reference operator++() && noexcept;

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

constexpr reference operator--() && noexcept;

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

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

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

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

constexpr operator value_type() const noexcept;

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

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

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

5 Returns: A copy of *this.

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

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

Returns: A copy of *this.

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

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

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

Returns: A copy of *this.

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

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

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

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

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

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

Effects: Exchanges the values a and b refer to.

(6.2.1.6.4) 28.9.6.4 simd constructors

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

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

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

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

template<class U, class UAbi>
constexpr explicit(see below) simd(const simd<U, simd_abi::fixed_size<size_t(UAbi)>& x) noexcept;
3 Effects: Constructs an object where the \(i\)th element equals `static_cast<T>(x[i])` for all \(i\) in the range of \([0, \text{size}())\).

4 Remarks: This constructor shall not participate in overload resolution unless \(\text{simd}_\text{size}_\text{v}<U, \text{UAbi}>=\text{size}()\).
- `abi_type` is `simd_abi::fixed_size<size_t>()`, and
- every possible value of \(u\) can be represented with type `value_type`, and
- if both \(u\) and `value_type` are integral, the integer conversion rank (??) of `value_type` is greater than the integer conversion rank of \(u\).

5 The constructor is explicit if:
- at least one possible value of \(u\) cannot be represented with type `value_type`, or
- if both \(u\) and `value_type` are integral types, the integer conversion rank (??) of \(u\) is greater than the integer conversion rank of `value_type`, or
- if both \(u\) and `value_type` are floating-point types, the floating-point conversion rank (??) of \(u\) is greater than the floating-point conversion rank of `value_type`.

6 \texttt{template<class G> constexpr simd(G&& gen) noexcept;}

7 Effects: Constructs an object where the \(i\)th element is initialized to `gen(integral_constant<size_t, i>())`.

8 Remarks: This constructor shall not participate in overload resolution unless `simd(gen(integral_constant<size_t, i>()))` is well-formed for all \(i\) in the range of \([0, \text{size}())\).

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

10 \texttt{template<class U, contiguous_iterator It, class Flags = element_aligned_tag> constexpr simd(const U* memIt& first, Flags f = {});}

11 Constraints: `iter_value_t<It>` is a vectorizable type.

12 Requires:
- \([\text{first, first + size()}]\) is a valid range.
- If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd, U>`.
- If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by \(N\).
- If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignment_v<U>`.

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

14 Remarks: This constructor shall not participate in overload resolution unless
- `is_simd_flag_type_v<Flags>` is `true`, and
- \(u\) is a vectorizable type.
Constraints: `iter_value_t<It>` is a vectorizable type.

Requires:
- `[mem, mem + size())` is a valid range.
- If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd, U>`.
- If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.
- If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

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

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

```cpp
template<class U, class Flags = element_aligned_tag>
requires output_iterator<It, value_type>
constexpr void copy_to(U* mem, const It& first, Flags f = {}) const;
```

Constraints: `iter_value_t<It>` is a vectorizable type.

Requires:
- `[mem, mem + size())` is a valid range.
- If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd, U>`.
- If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.
- If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

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

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

(6.2.1.6.6) 28.9.6.6 simd subscript operators

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

Requires: `i < size()`.

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

Throws: Nothing.

```cpp
constexpr value_type operator[](size_t i) const;
```
4 Requires: \( i < \text{size}() \).
5 Returns: The value of the \( i \)th element.
6 Throws: Nothing.

(6.2.1.6.7) 28.9.6.7 simd unary operators  

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

constexpr simd\& operator++() noexcept;
2 Effects: Increments every element by one.
3 Returns: *this.

constexpr simd operator++(int) noexcept;
4 Effects: Increments every element by one.
5 Returns: A copy of *this before incrementing.

constexpr simd\& operator--() noexcept;
6 Effects: Decrements every element by one.
7 Returns: *this.

constexpr simd operator--(int) noexcept;
8 Effects: Decrements every element by one.
9 Returns: A copy of *this before decrementing.

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

constexpr simd operator~() const noexcept;
11 Returns: A simd object where each bit is the inverse of the corresponding bit in *this.
12 Remarks: This operator shall not participate in overload resolution unless \( \tau \) is an integral type.

constexpr simd operator*() const noexcept;
13 Returns: *this.

constexpr simd operator-() const noexcept;
14 Returns: A simd object where the \( i \)th element is initialized to \(-\text{operator}[i] \) for all \( i \) in the range of \([0, \text{size}())\).
(6.2.1.7) 28.9.7 simd non-member operations

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

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

2 Remarks: Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type \text{value_type}.

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

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

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

(6.2.1.7.2) 28.9.7.2 simd compound assignment

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

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

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

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

76
friend constexpr simd& operator<=(simd& lhs, int n) noexcept;
friend constexpr simd& operator>>=(simd& lhs, int n) noexcept;

4 Effects: Equivalent to: return operator@=(lhs, simd(n));
5 Remarks: These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type value_type.

(simd.comparison)

28.9.7.3 simd compare operators

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

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

(simd.reductions)

28.9.7.4 simd reductions

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

template<class T, class Abi, class BinaryOperation = plus<>>

1 constexpr T reduce(const simd<T, Abi>& x, BinaryOperation binary_op = {});

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

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

4 Throws: Any exception thrown from binary_op.

5 template<class M, class V, class BinaryOperation>

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

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

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

7 Throws: Any exception thrown from binary_op.

77
Returns: If none_of(x.mask), returns 0. Otherwise, returns \texttt{GENERALIZED\_SUM}(binary\_op, x.data[i], ...) for all selected indices \(i\).

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

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

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

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

Returns: If none_of(x.mask), returns \(-V::value_type(). Otherwise, returns \texttt{GENERALIZED\_SUM}(binary\_op, x.data[i], ...) for all selected indices \(i\).

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

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

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

Returns: If none_of(x.mask), the return value is \texttt{numeric\_limits\langle V::value\_type\rangle::max}. Otherwise, returns the value of an element \(x.data[j]\) for which \(x.mask[j] == \text{true}\) and \(x.data[j] \leq x.data[i]\) for all selected indices \(i\).

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

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

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

Returns: If none_of(x.mask), the return value is \texttt{numeric\_limits\langle V::value\_type\rangle::lowest}. Otherwise, returns the value of an element \(x.data[j]\) for which \(x.mask[j] == \text{true}\) and \(x.data[j] \geq x.data[i]\) for all selected indices \(i\).
template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>& x) noexcept;

Let T\_value\_type denote T\_value\_type if is\_simd\_v<T> is true, or \( T \) otherwise.

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

Remarks: The function shall not participate in overload resolution unless

- every possible value of type \( U \) can be represented with type \( To \), and

- either
  - is\_simd\_v<T> is false, or
  - T::size() == simd<U, Abi>::size() is true.

The return type is

- if is\_simd\_v<T> is true;

- otherwise, simd<T, Abi> if \( U \) is the same type as \( T \);

- otherwise, simd<T, simd\_abi::fixed\_size<simd<U, Abi>::size()>>.

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

Let T\_value\_type denote T\_value\_type if is\_simd\_v<T> is true, or \( T \) otherwise.

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

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

- is\_simd\_v<T> is false, or

- T::size() == simd<U, Abi>::size() is true.

The return type is

- if is\_simd\_v<T> is true;

- otherwise, simd<T, Abi> if \( U \) is the same type as \( T \) or make\_signed\_t<\( U \) is the same type as make\_signed\_t<T>;

- otherwise, simd<T, simd\_abi::fixed\_size<simd<U, Abi>::size()>>.

template<class T, class U, class Abi>
fixed\_size\_simd\_T, simd\_size\_v<T, Abi> to\_fixed\_size(const simd\_T, Abi\& x) noexcept;
template<class T, class U, class Abi>
fixed\_size\_simd\_mask\_T, simd\_size\_v<T, Abi> to\_fixed\_size(const simd\_mask\_T, Abi\& x) noexcept;

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

template<class T, int N, native\_simd\_T to\_native(const fixed\_size\_simd\_T, N\& x) noexcept;
template<class T, int N, native\_simd\_mask\_T to\_native(const fixed\_size\_simd\_mask\_T, N\& x) noexcept;

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

Remarks: These functions shall not participate in overload resolution unless simd\_size\_v<T, simd\_abi::native\_T> == N is true.
template<class T, int N> simd<T> to_compatible(const fixed_size_simd<T, N>& x) noexcept;

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

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

Remarks: These functions shall not participate in overload resolution unless \(\text{simd\_size\_v<T, \text{simd\_abi::compatible}\_T}==N\) is true.

template<
size_t... Sizes, class T, class Abi>
constexpr tuple<
const simd<T, Abi>& x

split(const simd<T, Abi>& x) noexcept;

template<
size_t... Sizes, class T, class Abi>
constexpr tuple<
const simd_mask<T, Abi>& x

split(const simd_mask<T, Abi>& x) noexcept;

Returns: A tuple of data-parallel objects with the \(i\)th \(\text{simd/simd\_mask}\) element of the \(j\)th tuple element initialized to the value of the element in \(x\) with index \(i + \text{sum of the first } j \text{ values in the } Sizes \text{ pack}\).

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

template<class V, class Abi>
constexpr array<
V, simd_size_v<typename V::value_type, Abi> / V::size()>

split(const simd<typename V::value_type, Abi>& x) noexcept;

template<class V, class Abi>
constexpr array<
V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>

split(const simd_mask<typename V::simd_type::value_type, Abi>& x) noexcept;

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

Remarks: These functions shall not participate in overload resolution unless either:
- \(\text{is\_simd\_v\_V}\) is true and \(\text{simd\_size\_v\_typename V::value\_type, Abi}>\) is an integral multiple of \(V::size()\),
- or
- \(\text{is\_simd\_mask\_v\_V}\) is true and \(\text{simd\_size\_v\_typename V::simd\_type::value\_type, Abi}>\) is an integral multiple of \(V::size()\).

template<
size_t N, class T, class A>
constexpr array<
resize_simd<
size_t N, size_t N, class A>

split_by(const simd<T, A>& x) noexcept;

template<
size_t N, class T, class A>
constexpr array<
resize_simd<
size_t N, size_t N, class A>

split_by(const simd_mask<T, A>& x) noexcept;

Returns: An array \(arr\), where \(arr[i][j]\) is initialized by \(x[i \times \text{simd\_size\_v\_T, A} / N + j]\).

Remarks: The functions shall not participate in overload resolution unless \(\text{simd\_size\_v\_T, A} >\) is an integral multiple of \(N\).
template<class T, class... Abis>

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

template<class T, class... Abis>

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

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

template<class T, class Abi, size_t N>

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

template<class T, class Abi, size_t N>

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

21 Returns: A data-parallel object, the \(i^{th}\) element of which is initialized by \(\text{arr}[i / \text{simd_size_v<T, Abi>}]\).
For each set of overloaded functions within `<cmath>`, there shall be additional overloads sufficient to ensure that if any argument corresponding to a `double` parameter has type `simd<T, Abi>`, where `is_floating_point_v<T>` is true, then:

- All arguments corresponding to `double` parameters shall be convertible to `simd<T, Abi>`.
- All arguments corresponding to `double*` parameters shall be of type `simd<T, Abi>.star`.
- All arguments corresponding to parameters of integral type `U` shall be convertible to `fixed_size_simd<U, simd_size_v<T, Abi>>`.
- All arguments corresponding to `U*`, where `U` is integral, shall be of type `fixed_size_simd<U, simd_size_v<T, Abi>>*`.
- If the corresponding return type is `double`, the return type of the additional overloads is `simd<T, Abi>`. Otherwise, if the corresponding return type is `bool`, the return type of the additional overload is `simd_mask<T, Abi>`. Otherwise, the return type is `fixed_size_simd<R, simd_size_v<T, Abi>>`, with `R` denoting the corresponding return type.

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

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

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

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

### 28.9.8 Class template `simd_mask`

#### 28.9.8.1 Class template `simd_mask` overview

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

        static constexpr size_t size() noexcept = default;

        constexpr simd_mask() noexcept = default;

    // 28.9.8.3, `simd_mask` constructors
```
constexpr explicit simd_mask(value_type) noexcept;

```cpp
template<class U, class UAbi>
  constexpr explicit(sizeof(U) != sizeof(T)) noexcept;
template<class G> constexpr explicit simd_mask(G&& gen) noexcept;
template<contiguous_iterator It, class Flags = element_aligned_tag>
  constexpr simd_mask(const simd_mask<U, simd_abi::fixed_size<size()>>&) noexcept;
template<class G> constexpr simd_mask(G&& gen) noexcept;
```
If Abi is an extended ABI tag, it is implementation-defined whether simd_mask<T, Abi> is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. — end note]

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

- is_nothrow_move_constructible_v<simd_mask<T, Abi>>
- is_nothrow_move_assignable_v<simd_mask<T, Abi>>
- is_nothrow_default_constructible_v<simd_mask<T, 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 simd_mask:

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

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

(28.9.6.3)

(6.2.18.2) 28.9.8.2 simd_mask width

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

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

(6.2.18.3) 28.9.8.3 C:simd_mask constructors

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

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

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

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

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

```
abi_type_t simd_abi::fixed_size<size()> simd_size_v<U, UAbi> == size();
```

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

4 Effects: Constructs an object where the i-th element is initialized to gen(integral_constant<type_size, i>()).

5 Remarks: This constructor shall not participate in overload resolution unless static_cast<bool>(gen(integral_constant<type_size, i>())) is well-formed for all i in the range of [0, size()).

6 The calls to gen are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by gen(??).
template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr simd_mask(const value_type* mem, It first, Flags = {});

Constraints: iter_value_t<It> is of type bool.

Requires:
• [mem, mem + size()) is a valid range.
• If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<simd_mask>.
• If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.
• If the template parameter Flags is element_aligned_tag, mem shall point to storage aligned by alig

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

Throws: Nothing.

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

(6.2.1.8.4) 28.9.8.4 Cs

template<contiguous_iterator It, class Flags = element_aligned_tag>
constexpr void copy_from(const value_type* mem, It first, Flags = {});

Constraints: iter_value_t<It> is of type bool.

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

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

Throws: Nothing.

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

template<contiguous_iterator It, class Flags = element_aligned_tag>
requires output_iterator<It, value_type>
constexpr void copy_to(value_type* mem, const It first, Flags = {} const);

Constraints: iter_value_t<It> is of type bool.

Requires:
• [mem, mem + size()) is a valid range.
• If the template parameter Flags is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd_mask>.
• If the template parameter Flags is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.
• If the template parameter Flags is `element_aligned_tag`, `mem` shall point to storage aligned by `alignment(value_type)`.

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

**Throws:** Nothing.

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

(6.2.1.8.5) **28.9.8.5 `simd_mask` subscript operators**

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

1 **Requires:** `i < size()`.
2 **Returns:** A `reference` (see 28.9.6.3) referring to the `i`th element.
3 **Throws:** Nothing.

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

4 **Requires:** `i < size()`.
5 **Returns:** The value of the `i`th element.
6 **Throws:** Nothing.

(6.2.1.8.6) **28.9.8.6 `simd_mask` unary operators**

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

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

(6.2.1.9) **28.9.9 Non-member operations**

(6.2.1.9.1) **28.9.9.1 `simd_mask` binary operators**

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

1 **Returns:** A `simd_mask` object initialized with the results of applying the indicated operator to `lhs` and `rhs` as a binary element-wise operation.
(6.2.1.9.2) 28.9.9.2 \texttt{Csimd\_mask} compound assignment [simd.mask.cassign]

friend \texttt{constexpr simd\_mask\& operator\&=(simd\_mask\& lhs, const simd\_mask\& rhs) noexcept;}
friend \texttt{constexpr simd\_mask\& operator\|=(simd\_mask\& lhs, const simd\_mask\& rhs) noexcept;}
friend \texttt{constexpr simd\_mask\& operator^=(simd\_mask\& lhs, const simd\_mask\& rhs) noexcept;}

1 \textbf{Effects}: These operators apply the indicated operator to \texttt{lhs} and \texttt{rhs} as a binary element-wise operation.
2 \textbf{Returns}: \texttt{lhs}.

(6.2.1.9.3) 28.9.9.3 \texttt{C simd\_mask} comparisons [simd.mask.comparison]

friend \texttt{constexpr simd\_mask operator==(const simd\_mask&, const simd\_mask&) noexcept;}
friend \texttt{constexpr simd\_mask operator!=(const simd\_mask&, const simd\_mask&) noexcept;}

1 \textbf{Returns}: A \texttt{simd\_mask} object initialized with the results of applying the indicated operator to \texttt{lhs} and \texttt{rhs} as a binary element-wise operation.

(6.2.1.9.4) 28.9.9.4 \texttt{R simd\_mask} reductions [simd.mask.reductions]

\texttt{template<class T, class Abi> constexpr bool all_of(const simd\_mask\& T, Abi\& k) noexcept;}
\texttt{template<class T, class Abi> constexpr bool any_of(const simd\_mask\& T, Abi\& k) noexcept;}
\texttt{template<class T, class Abi> constexpr bool none_of(const simd\_mask\& T, Abi\& k) noexcept;}
\texttt{template<class T, class Abi> constexpr bool some_of(const simd\_mask\& T, Abi\& k) noexcept;}
\texttt{template<class T, class Abi> constexpr int popcount(const simd\_mask\& T, Abi\& k) noexcept;}
\texttt{template<class T, class Abi> constexpr int find_first_set(const simd\_mask\& T, Abi\& k);}
template<class T, class Abi> constexpr int find_last_set(const simd_mask<T, Abi>& k);

Requires: any_of(k) returns true.

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

Throws: Nothing.

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

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

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

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

Returns: The value of the argument is true.

Returns: 0.

Throws: Nothing.

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

(6.2.1.9.5) 28.9.9.5 where functions

```cpp
28.9.9.5 where functions [simd.mask.where]

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

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

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

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

Returns: An object (28.9.5) with mask and data initialized with \( k \) and \( v \) respectively.
```
template<class T>
where_expression<bool T>
where(see below k, T& v) noexcept;

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

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

Returns: An object (28.9.5) with mask and data initialized with k and v respectively.