This paper describes class templates for portable data-parallel (e.g. SIMD) programming via vector types.

ABSTRACT

CONTENTS
0 REMARKS

• This document talks about “vector” types/objects. In general this will not refer to the std::vector class template. References to the container type will explicitly call out the std prefix to avoid confusion.

• In the following, \( W_T \) denotes the number of scalar values (width) in a vector of type \( T \) (sometimes also called the number of SIMD lanes).

• [N4184], [N4185], and [N4395] provide more information on the rationale and design decisions. [N4454] discusses a matrix multiplication example. My PhD thesis [1] contains a very thorough discussion of the topic.

• This paper is not supposed to specify a complete API for data-parallel types and operations. It is meant as a useful starting point. Once the foundation is settled on, higher level APIs will be proposed.

1 CHANGEOLOGY

1.1 CHANGES FROM R0

Previous revision: [P0214R0].

• Extended the simd_abi tag types with a fixed_size<N> tag to handle arbitrarily sized vectors (5.1.1.1).

• Converted memory_alignment into a non-member trait (5.1.1.2).

• Extended implicit conversions to handle simd_abi::fixed_size<N> (5.1.2.2).

• Extended binary operators to convert correctly with simd_abi::fixed_size<N> (5.1.3.1).

• Dropped the section on “simd logical operators”. Added a note that the omission is deliberate (??).

• Added logical and bitwise operators to simd_mask (5.1.5.1).

• Modified simd_mask compares to work better with implicit conversions (5.1.5.3).

• Modified where to support different Abi tags on the simd_mask and simd arguments (5.1.5.5).
• Converted the load functions to non-member functions. SG1 asked for guidance from LEWG whether a load-expression or a template parameter to load is more appropriate.

• Converted the store functions to non-member functions to be consistent with the load functions.

• Added a note about masked stores not invoking out-of-bounds accesses for masked-off elements of the vector.

• Converted the return type of `simd::operator[]` to return a smart reference instead of an lvalue reference.

• Modified the wording of `simd_mask::operator[]` to match the reference type returned from `simd::operator[]`.

• Added non-trig/pow/exp/log math functions on `simd`.

• Added discussion on defaulting load/store flags.

• Added sum, product, min, and max reductions for `simd`.

• Added load constructor.

• Modified the wording of `native_handle()` to make the existence of the functions implementation-defined, instead of only the return type. Added a section in the discussion (cf. Section 6.8).

• Fixed missing flag objects.

1.2 Changes from R1

Previous revision: [P0214R1].

• Fixed converting constructor synopsis of `simd` and `simd_mask` to also allow varying Abi types.

• Modified the wording of `simd_mask::native_handle()` to make the existence of the functions implementation-defined.

• Updated the discussion of member types to reflect the changes in R1.

• Added all previous SG1 straw poll results.

• Fixed `commonabi` to not invent native Abi that makes the operator ill-formed.
• Dropped table of math functions.
• Be more explicit about the implementation-defined Abi types.
• Discussed resolution of the `simd_abi::fixed_size<N>` design (6.7.4).
• Made the compatible and native ABI aliases depend on `T` (5.1.1.1).
• Added `max_fixed_size` constant (5.1.1.1 p.4).
• Added masked loads.
• Added rationale for return type of `simd::operator~()` (6.10).

— SG1 guidance:
• Dropped the default load / store flags.
• Renamed the (un)aligned flags to `element_aligned` and `vector_aligned`.
• Added an `overaligned<N>` load / store flag.
• Dropped the ampersand on `native_handle` (no strong preference).
• Completed the set of math functions (i.e. add trig, log, and exp).

— LEWG (small group) guidance:
• Dropped `native_handle` and add non-normative wording for supporting `static_cast` to implementation-defined SIMD extensions.

• Dropped non-member load and store functions. Instead have `copy_from` and `copy_to` member functions for loads and stores. (5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4) (Did not use the load and store names because of the unfortunate inconsistency with `std::atomic`.)

• Added algorithm overloads for `simd` reductions. Integrate with `where` to enable masked reductions. (5.1.3.4) This made it necessary to spell out the class `where_expression`. 
1.3 Changes from r2

Previous revision: [P0214R2].

- Fixed return type of masked reduce (5.1.3.4).

- Added binary \texttt{min}, \texttt{max}, \texttt{minmax}, and \texttt{clamp} (5.1.3.6).

- Moved member \texttt{min} and \texttt{max} to non-member \texttt{hmin} and \texttt{hmax}, which cannot easily be optimized from \texttt{reduce}, since no function object such as \texttt{std::plus} exists (5.1.3.4).

- Fixed neutral element of masked \texttt{hmin}/\texttt{hmax} and drop UB (5.1.3.4).

- Removed remaining reduction member functions in favor of non-member reduce (as requested by LEWG).

- Replaced \texttt{init} parameter of masked reduce with \texttt{neutral_element} (5.1.3.4).

- Extend \texttt{where_expression} to support const \texttt{simd} objects (5.1.5.5).

- Fixed missing \texttt{explicit} keyword on \texttt{simd_mask} (bool) constructor (5.1.4.2).

- Made binary operators for \texttt{simd} and \texttt{simd_mask} friend functions of \texttt{simd} and \texttt{simd_mask}, simplifying the SFINAE requirements considerably (5.1.3.1, 5.1.5.1).

- Restricted broadcasts to only allow non-narrowing conversions (5.1.2.2).

- Restricted simd to simd conversions to only allow non-narrowing conversions with \texttt{fixed_size} ABI (5.1.2.2).

- Added generator constructor (as discussed in LEWG in Issaquah) (5.1.2.2).

- Renamed \texttt{copy_from} to \texttt{memload} and \texttt{copy_to} to \texttt{memstore} (5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4)

- Documented effect of \texttt{overaligned_tag} as \texttt{Flags} parameter to load/store. (5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4)

- Clarified \texttt{cv} requirements on \texttt{T} parameter of \texttt{simd} and \texttt{simd_mask}.

- Allowed all implicit \texttt{simd_mask} conversions with \texttt{fixed_size} ABI and equal size (5.1.4.2).

- Made increment and decrement of \texttt{where_expression} return void.
• Added static_simd_cast for simple casts (5.1.3.5).
• Clarified default constructor (5.1.2.1, 5.1.2.1).
• Clarified simd and simd_mask with invalid template parameters to be complete types with deleted constructors, destructor, and assignment (5.1.2.1, 5.1.2.1).
• Wrote a new subsection for a detailed description of where_expression (5.1.1.3).
• Moved masked loads and stores from simd and simd_mask to where_expression (5.1.1.3). This required two more overloads of where to support value objects of type simd_mask (5.1.5.5).
• Removed where_expression::operator! (5.1.1.3).
• Added aliases native_simd, native_mask, fixed_size_simd, fixed_size_mask (5.1.1).
• Removed bool overloads of mask reductions awaiting a better solution (5.1.5.4).
• Removed special math functions with f and l suffix and l and ll prefix (5.1.3.7).
• Modified special math functions with mixed types to use fixed_size instead of abi_for_size (5.1.3.7).
• Added simple ABI cast functions to_fixed_size, to_native, and to_compatible (5.1.3.5).

1.4

Previous revision: [P0214R3].

changes before Kona

• Add special math overloads for signed char and short. They are important to avoid widening to multiple SIMD registers and since no integer promotion is applied for simd types.
• Editorial: Prefer using over typedef.
• Overload shift operators with int argument for the right hand side. This enables more efficient implementations. This signature is present in the Vc library, and was forgotten in the wording.
• Remove empty section about the omission of logical operators.
• **Modify** `simd_mask` **compares to return a** `simd_mask` **instead of** `bool` (5.1.5.3). This resolves an inconsistency with all the other binary operators.

• **Editorial:** Improve reference member specification (5.1.2.1).

• **Require** `swap(v[0], v[1])` **to be valid** (5.1.2.1).

• **Fixed inconsistency of masked load constructor after move of** `memload` **to** `where_`-expression (5.1.1.3).

• **Editorial:** Use Requires clause instead of Remarks to require the memory argument to loads and stores to be large enough (5.1.1.3, 5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4).

• Add a note to special math functions that precondition violation is UB (5.1.3.7).

• **Bugfix:** Binary operators involving two `simd::reference` objects must work (5.1.2.1).

• **Editorial:** Replace Note clauses in favor of `[ Note: — end note ]`.

• **Editorial:** Replace UB Remarks on load/store alignment requirements with Requires clauses.

• **Add an example section** (4).

---

**Changes after Kona**

| design related:

• **Readd** `bool` **overloads of mask reductions and ensure that implicit conversions to** `bool` **are ill-formed.**

• Clarify effects of using an ABI parameter that is not available on the target (5.1.2.1 p.2, 5.1.4.1 p.2, 5.1.1.2 p.6).

• **Split** `where_expression` **into** `const` **and** `non-const` **class templates.**

• **Add section on naming.**

• **Discuss the questions/issues raised on** `max_fixed_size` **in Kona (Section 6.11).**

• **Make** `max_fixed_size` **dependent on** `T`.

• Clarify that converting loads and stores only work with arrays of non-bool arithmetic type (5.1.2.3, 5.1.2.4).
• Discuss `simd_mask` and `bitset` reduction interface differences (??).

• Relax requirements on return type of generator function for the generator constructor (5.1.2.2).

• Remove overly generic `simd_cast` function.

• Add proposal for a widening cast function (??).

• Add proposal for `split` and `concat` cast functions (??).

• Add `noexcept` or “Throws: Nothing.” to most functions.

— wording fixes & improvements:

• Remove non-normative noise about ABI tag types (5.1.1.1).

• Remove most of the text about vendor-extensions for ABI tag types, since it’s QoI (5.1.1.1).

• Clarify the differences and intent of `compatible<T>` vs. `native<T>` (5.1.1.1).

• Move definition of `where_expression` out of the synopsis (5.1.1.3).

• Editorial: Improve `is_simd` and `is_mask` wording (5.1.1.2).

• Make `ABI tag` a consistent term and add `is_abi_tag` trait (5.1.1.2, 5.1.1.1).

• Clarify that `simd_abi::fixed_size<N>` must support all `N` matching all possible implementation-defined ABI tags (5.1.1.1).

• Clarify `abi_for_size` wording (5.1.1.2).

• Turn `memory_alignment` into a trait with a corresponding `memory_alignment_v` variable template.

• Clarify `memory_alignment` wording; when it has no value member; and imply its value through a reference to the load and store functions (5.1.1.2).

• Remove exposition-only `where_expression` constructor and make exposition-only data members private (5.1.1.3).

• Editorial: use “shall not participate in overload resolution unless” consistently.

• Add a note about variability of `max_fixed_size` (5.1.1.1).
• Editorial: use “target architecture” and “currently targeted system” consistently.

• Add margin notes presenting a wording alternative that avoids “target system” and “target architecture” in normative text.

• Specify result of masked reduce with empty mask (5.1.3.4).

• Editorial: clean up the use of “supported” and resolve contradictions resulting from incorrect use of conventions in the rest of the standard text (5.1.2.1 p.2, 5.1.4.1 p.2, 5.1.1.2).

• Add Section 7 Feature Detection Macros.

1.5 changes from r4

Previous revision: [P0214R4].

• Changed align_val_t argument of overaligned<N> and overaligned_tag<N> to size_t. (Usage is otherwise too cumbersome.)

changes after LEWG Review

• Remove section on naming (after the topic was discussed and decided in LEWG).

• Rename datapar to simd and mask to simd_mask.

• Remove incorrect template parameters to scalar boolean reductions.

• Merge proposed simd_cast into the wording (using Option 3) and extend static_simd_cast accordingly.

• Merge proposed split and concat functions into the wording.

• Since the target is a TS, place headers into experimental/ directory.

1.6 changes from r5

Previous revision: [P0214R5].

• Renamed memload to copy_from and memstore to copy_to. (5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4)

• Fixed split to never convert the value_type. (5.1.3.5)

• Added missing long double overload of ceil. (5.1.3.7)
2

STRAW POLLS

2.1 sg1 at chicago 2013

Poll: Pursue SIMD/data parallel programming via types?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.2 sg1 at urbana 2014

- Poll: SF = ABI via namespace, SA = ABI as template parameter
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

- Poll: Apply size promotion to vector operations? SF = shortv + shortv = intv
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

- Poll: Apply “sign promotion” to vector operations? SF = ushortv + shortv = ushortv; SA = no mixed signed/unsigned arithmetic
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

2.3 sg1 at lenexa 2015

Poll: Make vector types ready for LEWG with arithmetic, compares, write-masking, and math?

<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>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.4 sg1 at jax 2016

- Poll: Should subscript operator return an lvalue reference?
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- Poll: Should subscript operator return a “smart reference”?
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
2 Straw Polls

- **Poll:** Specify simd width using ABI tag, with a special template tag for fixed size.
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Poll:** Specify simd width using <T, N, abi>, where abi is not specified by the user.
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

---

2.5 **SG1 at Oulu 2016**

- **Poll:** Keep `native_handle` in the wording (dropping the ampersand in the return type)?
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Poll:** Should the interface provide a way to specify a number for over-alignment?
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Poll:** Should loads and stores have a default load/store flag?
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

---

2.6 **LEWG at Issaquah 2016**

- **Poll:** Unary minus on unsigned simd should be ill formed
  
<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Poll:** Reductions only as free functions
  → unanimous consent

- **Poll:** Jens should work with the author and return with an updated paper
  → unanimous consent

---

2.7 **LEWG at Kona 2017**

- **Poll:** Want `operator<<(signed)` to work except where it’s undefined for the underlying integer?
  → unanimous consent
- Poll: Should there be overloads for \texttt{where} and \texttt{(simd\_mask and simd)} reductions with \texttt{bool/builtin} types in place of \texttt{simd\_mask/simd}?
  
  \[
  \begin{array}{c|c|c|c|c|c}
  \text{SF} & \text{F} & \text{N} & \text{A} & \text{SA} \\
  \hline
  1 & 5 & 6 & 1 & 0 \\
  \end{array}
  \]

  → unanimous consent

- Poll: Should there be a named “widen” function that widens the element type \( T \) from (e.g.) \texttt{int} to \texttt{long}, but rejects \texttt{int} to \texttt{short}? The number of elements is not changed.
  
  → unanimous consent
  
  cf. ??

- Poll: Should there be a named “concat” functions that concatenates several \texttt{simd}s with the same element type, but potentially different length?
  The latter two operations are currently lumped together in \texttt{simd\_cast}.
  The widen / concat questions, I think, comes down to whether we want the current \texttt{simd\_cast} or not.
  
  → unanimous consent
  
  cf. ??

- Poll: We still have a few open ends on implicit conversions (https://github.com/mattkretz/wg21-papers/issues/26, https://github.com/mattkretz/wg21-papers/issues/3). The current paper is rather strict, there may be room for more implicit conversions without introducing safety problems. Should we postpone any changes in this area to after TS feedback?
  
  → unanimous consent

- Poll: Keep or drop \texttt{simd\_abi::max\_fixed\_size}? (cf. https://github.com/mattkretz/wg21-papers/issues/38)
  
  Authors will come back with a proposal
  
  cf. Section 6.11

- Poll: Add a list of possible names for “simd” and “where” to the paper.
  
  → unanimous consent
  
  cf. ??

- Poll: Use Option 3 (support both element type and \texttt{simd} type as cast template argument, instead of just either one of them) for \texttt{static\_simd\_cast} and \texttt{simd\_}
cast.
→ unanimous consent

• Poll: Use abi_for_size_t instead of unconditionally using fixed_size for split and concat. Revisit the question after TS feedback.
→ unanimous consent

• Poll: Name it datapar (in favor) or simd (against)?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

We want TS feedback on the name, i.e. whether anyone is confused about the applicability of the feature because of the name.

• Poll: Name mask (in favor) or simdmask (against)?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

• Poll: Name simd_mask (in favor) or simdmask (against)?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• Poll: where (in favor) vs. masked (against)?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

No consensus for change, revisit after TS feedback.

• Vote for your preference for a name for the load/store functions

<table>
<thead>
<tr>
<th>load</th>
<th>memload</th>
<th>load_from</th>
<th>copy_from</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

• Poll: load_from (in favor) vs. copy_from (against)?

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

No consensus to override the author’s preference (copy_from).

• Poll: Keep simd_mask reductions as free functions with the current names. Consider follow-up papers if any names need to be changed (e.g. for consistency).
→ unanimous consent
3 INTRODUCTION

3.1 SIMD registers and operations

Since many years the number of SIMD instructions and the size of SIMD registers have been growing. Newer microarchitectures introduce new operations for optimizing certain (common or specialized) operations. Additionally, the size of SIMD registers has increased and may increase further in the future.

The typical minimal set of SIMD instructions for a given scalar data type comes down to the following:

- Load instructions: load $\mathcal{W}_T$ successive scalar values starting from a given address into a SIMD register.
- Store instructions: store from a SIMD register to $\mathcal{W}_T$ successive scalar values at a given address.
- Arithmetic instructions: apply the arithmetic operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD register.
- Compare instructions: apply the compare operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD mask register.
- Bitwise instructions: bitwise operations on SIMD registers.
- Shuffle instructions: permutation and/or blending of scalars in (a) SIMD register(s).

The set of available instructions may differ considerably between different microarchitectures of the same CPU family. Furthermore there are different SIMD register sizes. Future extensions will certainly add more instructions and larger SIMD registers.

3.2 Motivation for data-parallel types

SIMD registers and operations are the low-level ingredients to efficient programming for SIMD CPUs. At a more abstract level this is not only about SIMD CPUs, but efficient data-parallel execution (CPUs, GPUs, possibly FPGAs and classical vector...
supercomputers). Operations on fundamental types in C++ form the abstraction for CPU registers and instructions. Thus, a data-parallel type (SIMD type) can provide the necessary interface for writing software that can utilize data-parallel hardware efficiently. Higher-level abstractions can be built on top of these types. Note that if a low-level access to SIMD is not provided, users of C++ are either constrained to work within the limits of the provided abstraction or resort to non-portable extensions, such as SIMD intrinsics.

In some cases the compiler might generate better code if only the intent is stated instead of an exact sequence of operations. Therefore, higher-level abstractions might seem preferable to low-level SIMD types. In my experience this is a non-issue because programming with SIMD types makes intent very clear and compilers can optimize sequences of SIMD operations just like they can for scalar operations. SIMD types do not lead to an easy and obvious answer for efficient and easily usable data structures, though. But, in contrast to vector loops, SIMD types make unsuitable data structures glaringly obvious and can significantly support the developer in creating more suitable data layouts.

One major benefit from SIMD types is that the programmer can gain an intuition for SIMD. This subsequently influences further design of data structures and algorithms to better suit SIMD architectures.

There are already many users of SIMD intrinsics (and thus a primitive form of SIMD types). Providing a cleaner and portable SIMD API would provide many of them with a better alternative. Thus, SIMD types in C++ would capture and improve on widespread existing practice.

The challenge remains in providing portable SIMD types and operations.

3.3 problem

C++ has no means to use SIMD operations directly. There are indirect uses through automatic loop vectorization or optimized algorithms (that use extensions to C/C++ or assembly for their implementation).

All compiler vendors (that I worked with) add intrinsics support to their compiler products to make SIMD operations accessible from C. These intrinsics are inherently not portable and most of the time very directly bound to a specific instruction. (Compilers are able to statically evaluate and optimize SIMD code written via intrinsics, though.)
4.1 Loop Vectorization

This shows a low-level approach of manual loop chunking + epilogue for vectorization (“Leave no room for a lower-level language below C++ (except assembler).” [2]). It also shows SIMD loads, operations, write-masking (blending), and stores.

```cpp
using floatv = native_simd<float>;
void f() {
    alignas(memory_alignment_v<floatv>) float data[N];
    fill_data(data);
    size_t i = 0;
    for (; i + floatv::size() <= N; i += floatv::size()) {
        floatv v(&data[i], flags::vector_aligned);
        where(v > 100.f, v) = 100.f + (v - 100.f) * 0.1f;
        v.copy_to(&data[i], flags::vector_aligned);
    }
    for (; i < N; ++i) {
        float x = data[i];
        if (x > 100.f) {
            x = 100.f + (x - 100.f) * 0.1f;
        }
        data[i] = x;
    }
}
```

5 Wording

The following is a draft targeting inclusion into the Parallelism TS 2. It defines a basic set of data-parallel types and operations.

5.1 Data-Parallel Types

5.1.1 Header <experimental/simd> synopsis
namespace flags {
    struct element_aligned_tag {};
    struct vector_aligned_tag {};
    template <size_t N> struct overaligned_tag {};
    constexpr element_aligned_tag element_aligned {};
    constexpr vector_aligned_tag vector_aligned {};
    template <size_t N> constexpr overaligned_tag<N> overaligned {};
}

// traits [simd.traits]
template <class T> struct is_abi_tag {};
template <class T> constexpr bool is_abi_tag_v = is_abi_tag<T>::value {};
template <class T> struct is_simd {};
template <class T> constexpr bool is_simd_v = is_simd<T>::value {};
template <class T> struct is_simd_mask {};
template <class T> constexpr bool is_simd_mask_v = is_simd_mask<T>::value {};
template <class T, size_t N> struct abi_for_size { using type = implementation-defined; };
template <class T, size_t N> using abi_for_size_t = typename abi_for_size<T, N>::type {};
template <class T, class Abi = simd_abi::compatible<T>> class simd {};
template <class T> using native_simd = simd<T, simd_abi::native<T>>;
template <class T, int N> using fixed_size_simd = simd<T, simd_abi::fixed_size<N>>;

// class template simd [simd]
template <class T, class Abi = simd_abi::compatible<T>> class simd {};
template <class T> using native_simd = simd<T, simd_abi::native<T>>;

// class template simd_mask [simd_mask]
template <class T, class Abi = simd_abi::compatible<T>> class simd_mask {};
template <class T> using native_simd_mask = simd_mask<T, simd_abi::native<T>>;

// casts [simd.casts]
template <class T, class U, class A> /* see below*/ simd_cast (const simd<U, A> &);
template <class T, class U, class A> /* see below*/ static_simd_cast (const simd<U, A> &);
template <class T, class A> simd<T, simd_abi::fixed_size<simd_size_v<T>, A>> to_fixed_size (const simd<T, A> &) noexcept {};
template <class T, class A> simd_mask<T, simd_abi::fixed_size<simd_size_v<T>, A>> to_fixed_size (const simd_mask<T, A> &) noexcept {};
template <class T, size_t N> simd<T, simd_abi::native<T>> to_native (const simd<T, simd_abi::fixed_size<N>, A> &) noexcept {};
template <class T, size_t N> simd_mask<T, simd_abi::native<T>> to_native (const simd_mask<T, simd_abi::fixed_size<N>, A> &) noexcept {};
template <class T, size_t N> simd<T, simd_abi::compatible<T>> to_compatible (const simd<T, simd_abi::fixed_size<N>, A> &) noexcept {};
template <class T, size_t N> simd_mask<T, simd_abi::compatible<T>> to_compatible (const simd_mask<T, simd_abi::fixed_size<N>, A> &) noexcept {};

template <size_t... Sizes, class T, class A>
tuple<simd<T, abi_for_size_t<Sizes>>...> split(const simd<T, A>&);

template<typename V, class A>
array<V, simd_size_v<typename V::value_type, A>> where(const typename V::mask_type &,
const simd<T, A>&) noexcept;

template<typename V, class A>
const array<V, simd_size_v<typename V::value_type, A>> where(const remove_const_t<
simd<T, A>>&,
const simd<T, A>&) noexcept;

template<class T, class M>
const typename where_expression<T, M>::value_type reduce(const typename
where_expression<T, M>::value_type neutral_element = default_neutral_element,
const where_expression<T, M>& x, T& d) noexcept;

// reductions [simd_mask.reductions]
template<class T, class Abi>
bool all_of(simd_mask<T, Abi>) noexcept;

bool any_of(simd_mask<T, Abi>) noexcept;

bool none_of(simd_mask<T, Abi>) noexcept;

int popcount(simd_mask<T, Abi>) noexcept;

int find_first_set(simd_mask<T, Abi>) noexcept;

int find_last_set(simd_mask<T, Abi>) noexcept;

// masked assignment [simd_mask.where]
template<class M, class T>
class const_where_expression;

template<class M, class T>
class where_expression;

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

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

template<class T, class A>
where_expression<simd_mask<T, A>, simd<T, A>> where(const remove_const_t<
simd_mask<T, A>>&,
simd_mask<T, A>&) noexcept;

template<class T, class A>
const where_expression<simd_mask<T, A>, const simd_mask<T, A>> where(
const remove_const_t<
simd_mask<T, A>>&,
const simd_mask<T, A>&) noexcept;

template<class T> where_expression<bool, T> where(implementation-defined k, T d) noexcept;
5.1.1.1 simd ABI tags

An ABI tag type indicates a choice of target architecture dependent size and binary representation for simd and simd_mask objects. 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 5.1.2.1 p.2). The ABI tags enable users to safely pass simd and simd_mask objects between translation unit boundaries (e.g. function calls or I/O). — end note]

Use of the scalar tag type forces simd and simd_mask to store a single component (i.e. simd<T, simd_abi::scalar>::size() returns 1). [Note: scalar shall not be an alias for fixed_size<1>. — end note]

Use of the simd_abi::fixed_size<N> tag type forces simd and simd_mask to store and manipulate N components (i.e. simd<T, simd_abi::fixed_size<N>::size() returns N). An implementation must support at least any N ∈ [1 ... 32]. Additionally, for every supported simd<T, A> (see 5.1.2.1 p.2), where A is an implementation-defined ABI tag, N = simd<T, A>::size() must be supported. [Note: An implementation may choose to forego ABI compatibility between differently compiled translation units for simd and simd_mask instantiations using the same simd_abi::fixed_size<N> tag. Otherwise, the efficiency of simd<T, Abi> is likely to be better than for simd<T, fixed_size<simd_size_v<T, Abi>>> (with Abi not a instance of simd_abi::fixed_size). — end note]

The value of max_fixed_size<T> declares that an instance of simd<T, fixed_size<N>> with N <= max_fixed_size<T> is supported by the implementation. [Note: It is unspecified whether an implementation supports simd<T, fixed_size<N>> with N > max_fixed_size<T>. The value of max_—
fixed_size<T> may depend on compiler flags and may change between different compiler versions.

--- end note ---

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

simd_abi::compatible<T> is an alias for the ABI tag with the most efficient data parallel execution for the element type T that ensures ABI compatibility on the target architecture.

Example: Consider a target architecture supporting the implementation-defined ABI tags simd128 and simd256, where the simd256 type requires an optional ISA extension on said target architecture. Also, the target architecture does not support long double with either ABI tag. The implementation therefore defines

- compatible<T> as an alias for simd128 for all arithmetic T, except long double,
- and compatible<long double> as an alias for scalar.

--- end example ---

simd_abi::native<T> is an alias for the ABI tag with the most efficient data parallel execution for the element type T that is supported on the currently targeted system.

Note: For target architectures without ISA extensions, the native<T> and compatible<T> aliases will likely be the same. For target architectures with ISA extensions, compiler flags may influence the native<T> alias while compatible<T> will be the same independent of such flags. --- end note ---

Example: Consider a currently targeted system supporting the implementation-defined 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 ---

5.1.1.2 simd type traits [simd.traits]

```cpp
template <class T> struct is_abi_tag;

The type is_abi_tag<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is the type of a standard or implementation-defined ABI tag, and false_type otherwise.

template <class T> struct is_simd;

The type is_simd<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is an instance of the simd class template, and false_type otherwise.

template <class T> struct is_mask;

The type is_mask<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is an instance of the simd_mask class template, and false_type otherwise.

template <class T, size_t N> struct abi_for_size { using type = implementation-defined; };
```
The member type shall be omitted if

- $T$ is not a cv-unqualified floating-point or integral type except $\text{bool}$.
- or if $\text{simd}\_\text{abi}::\text{fixed\_size}<N>$ is not supported (see 5.1.1.1 p.3).

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

- $\text{simd}\_\text{size}\_v<T, \text{type}>> == N$.
- $\text{simd}\_T, \text{type}$ is default constructible (see 5.1.2.1 p.2).

$\text{simd}\_\text{abi}::\text{scalar}$ takes precedence over $\text{fixed\_size}<N>$. The precedence of implementation-defined ABI tags over $\text{simd}\_\text{abi}::\text{fixed\_size}<N>$ is implementation-defined. [ Note: It is expected that implementation-defined ABI tags can produce better optimizations and thus take precedence over $\text{simd}\_\text{abi}::\text{fixed\_size}<N>$. — end note ]

```
template <class T, class Abi = $\text{simd}\_\text{abi}::\text{compatible}<T>>$ struct simd_size;
```

$simd\_\text{size}\_T, \text{Abi}$ shall have no member value if either

- $T$ is not a cv-unqualified floating-point or integral type except $\text{bool}$,
- or $\text{is\_abi\_tag\_v}<\text{Abi}>$ is false.

[ Note: The rules are different from 5.1.2.1 p.2 — end note ]

Otherwise, the type $\text{simd\_size}<T, \text{Abi}>$ is a BinaryTypeTrait with a BaseCharacteristic of integral_constant $<\text{size\_t}, N>$ with $N$ equal to the number of elements in a $\text{simd}\_T, \text{Abi}$ object. [ Note: If $\text{simd}\_T, \text{Abi}$ is not supported for the currently targeted system, $\text{simd\_size}\_T, \text{Abi}::\text{value}$ produces the value $\text{simd}\_T, \text{Abi}::\text{size}()$ would return if it were supported. — end note ]

```
template <class T, class U = typename T::value\_type> struct memory\_alignment;
```

$\text{memory\_alignment}\_T, U$ shall have no member value if either

- $T$ is cv-qualified,
- or $U$ is cv-qualified,
- or $\text{is\_simd}\_v<T> \&\& \text{is\_mask}\_v<T>$,
- or $\text{is\_simd}\_v<T>$ and $U$ is not an arithmetic type or $U$ is $\text{bool}$,
- or $\text{is\_mask}\_v<T>$ and $U$ is not $\text{bool}$.

Otherwise, the type $\text{memory\_alignment}\_T, U$ is a BinaryTypeTrait with a BaseCharacteristic of integral_constant $<\text{size\_t}, N>$ for some implementation-defined $N$. [ Note: value identifies the alignment restrictions on pointers used for (converting) loads and stores for the given type $T$ on arrays of type $U$ (see 5.1.2.3, 5.1.2.4, 5.1.4.3, 5.1.4.4). — end note ]

5.1.1.3 Class templates const\_where\_expression and where\_expression

```cpp
namespace std {
    namespace experimental {
        template <class M, class T> class const\_where\_expression {
            const M& simd\_mask; // exposition only
            T& data; // exposition only
        public:
            const\_where\_expression(const const\_where\_expressions& = delete);
        }
    }
}
```
The class templates `const_where_expression` and `where_expression<M, T>` combine a predicate and a value object to implement an interface that restricts assignments and/or operations on the value object to the elements selected via the predicate.

The first template argument `M` must be cv-unqualified `bool` or a cv-unqualified `simd_mask` instantiation.

The second template argument `T` must be a cv-unqualified or const qualified type `U`. If `M` is `bool`, `U` must be an arithmetic type. Otherwise, `U` must either be `M` or `M::simd_type`.

- **Note:** The implementation initializes a `where_expression<M, T>` object with a predicate of type `M` and a reference to a value object of type `T`. The predicate object and a const qualified value object may be copied by the constructor implementation. — *end note*

- **Note:** The following declarations refer to the predicate as data member `mask` and to the value reference as data member `data`. — *end note*
6 Returns: If \( M \) is bool, \(-\text{data} \) if mask is true, \text{data} otherwise. If \( M \) is not bool, returns an object with the \( i \)-th element initialized to \(-\text{data}[i] \) if mask\([i]\) is true and \text{data}[i] otherwise for all \( i \in [0, M::\text{size}()) \).

7 template <class U, class Flags> 
    [[nodiscard]] remove_const_t<T> copy_from(const U *mem, Flags) const &;

8 Remarks: If remove_const_t<T> is bool or is_mask_v<remove_const_t<T>>, the function shall not participate in overload resolution unless \( U \) is bool. Otherwise, the function shall not participate in overload resolution unless \( U \) is an arithmetic type except bool.

9 Returns: If \( M \) is bool, return \( \text{mem}[0] \) if mask equals true and return \text{data} otherwise. If \( M \) is not bool, return an object with the \( i \)-th element initialized to the \( i \)-th element of \text{data} if mask\([i]\) is false and static_cast<T::value_type>(\( \text{mem}[i] \)) if mask\([i]\) is true for all \( i \in [0, M::\text{size}()) \).

10 Requires: If \( M \) is not bool, the largest \( i \) where mask\([i]\) is true is less than the number of values pointed to by \( \text{mem} \).

11 Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<remove_const_t<T>, \( U \)>. If the Flags template parameter is of type flags::overaligned_tag\(<N>\), the pointer value represents an address aligned to \( N \).

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

13 Remarks: If remove_const_t<T> is bool or is_mask_v<remove_const_t<T>>, the function shall not participate in overload resolution unless \( U \) is bool. Otherwise, the function shall not participate in overload resolution unless \( U \) is an arithmetic type except bool.

14 Effects: If \( M \) is bool, assigns \text{data} to \( \text{mem}[0] \) unless mask is false. If \( M \) is not bool, copies the elements \text{data}[i] where mask\([i]\) is true as if \( \text{mem}[i] = \text{static_cast<}\U*(\text{data}[i]) \) for all \( i \in [0, M::\text{size}()) \).

15 Requires: If \( M \) is not bool, the largest \( i \) where mask\([i]\) is true is less than the number of values pointed to by \( \text{mem} \).

16 Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<remove_const_t<T>, \( U \)>. If the Flags template parameter is of type flags::overaligned_tag\(<N>\), the pointer value represents an address aligned to \( N \).

17 template <class U> void operator=(U&& x);
18 template <class U> void operator+=(U&& x);
19 template <class U> void operator-=(U&& x);
20 template <class U> void operator*=(U&& x);
21 template <class U> void operator/=(U&& x);
22 template <class U> void operator%=(U&& x);
23 template <class U> void operator&=(U&& x);
24 template <class U> void operator|=(U&& x);
25 template <class U> void operator^=(U&& x);
26 template <class U> void operator<<=( U&& x);
27 template <class U> void operator>>=( U&& x);

18 Remarks: Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type \( T \).
Effects: If \( M \) is bool, applies the indicated operator on \( \text{data} \) and \( \text{forward<U>(x)} \) unless \( \text{mask} \) is false. If \( M \) is not bool, applies the indicated operator on \( \text{data} \) and \( \text{forward<U>(x)} \) without modifying the elements \( \text{data}[i] \) where \( \text{mask}[i] \) is false for all \( i \in [0, M::\text{size()}) \).

Remarks: It is unspecified whether the arithmetic/bitwise operation, which is implied by a compound assignment operator, is executed on all elements or only on the ones written back.

```cpp
void operator++();
void operator++(\text{int});
void operator--();
void operator--(\text{int});
```

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

Effects: If \( M \) is bool, applies the indicated operator on \( \text{data} \) unless \( \text{mask} \) is false. If \( M \) is not bool, applies the indicated operator on \( \text{data} \) without modifying the elements \( \text{data}[i] \) where \( \text{mask}[i] \) is false for all \( i \in [0, M::\text{size()}) \). [Note: It is unspecified whether the inc-/decrement operation is executed on all elements or only on the ones written back.—end note]

```cpp
template <\text{class} U, \text{class} Flags> \text{void} \text{copy_from(const U *mem, Flags);};
```

Remarks: If \( T \) is bool or \( \text{is\_mask\_v<T>} \), the function shall not participate in overload resolution unless \( U \) is bool.

Effects: If \( M \) is bool, assign \( \text{mem}[0] \) to \( \text{data} \) unless \( \text{mask} \) is false. If \( M \) is not bool, replace the elements of \( \text{data} \) where \( \text{mask}[i] \) is true such that the \( i \)-th element is assigned with \( \text{static\_cast<T::value\_type>(mem[i])} \) for all \( i \in [0, M::\text{size()}) \).

Requires: If \( M \) is not bool, the largest \( i \) where \( \text{mask}[i] \) is true is less than the number of values pointed to by \( \text{mem} \).

Requires: If the \( \text{Flags} \) template parameter is of type \( \text{flags::vector\_aligned\_tag} \), the pointer value represents an address aligned to \( \text{memory\_alignment\_v<T, U>} \). If the \( \text{Flags} \) template parameter is of type \( \text{flags::overaligned\_tag<N>} \), the pointer value represents an address aligned to \( N \).
simd(const simd&) = default;
simd(simd&) = default;
simd& operator=(const simd&) = default;
simd& operator=(simd&) = default;

// implicit broadcast constructor
template <class U> simd(U&);

// implicit type conversion constructor
template <class U> simd(const simd<U, simd_abi::fixed_size<size()>>&);

// generator constructor
template <class G> simd(G& gen);

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

// loads [simd.load]
template <class U, class Flags> void copy_from(const U* mem, Flags);

// stores [simd.store]
template <class U, class Flags> void copy_to(U* mem, Flags) const;

// scalar access [simd.subscr]
reference operator[](size_type);
value_type operator[](size_type) const;

// unary operators [simd.unary]
simd& operator++();
simd& operator++(int);
simd& operator--();
simd& operator--(int);
mask_type operator!() const;
simd operator-() const;
simd operator+() const;
simd operator-() const;

// binary operators [simd.binary]
friend simd& operator+(const simd&, const simd&);
friend simd& operator-(const simd&, const simd&);
friend simd& operator*(const simd&, const simd&);
friend simd& operator/(const simd&, const simd&);
friend simd& operator%(const simd&, const simd&);
friend simd& operator<<(const simd&, int);
friend simd& operator>>(const simd&, int);

// compound assignment [simd.cassign]
friend simd& operator+=(const simd&, const simd&);
friend simd& operator-=(const simd&, const simd&);
friend simd& operator*=(const simd&, const simd&);
friend simd& operator/=(const simd&, const simd&);
friend simd& operator%=(const simd&, const simd&);
friend simd& operator<<(const simd&, int);
friend simd& operator>>(const simd&, int);
The class template `simd<T, Abi>` is a one-dimensional smart array. The number of elements in the array is a constant expression, according to the `Abi` template parameter.

The resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment unless all of the following hold:

1. The first template argument `T` is a cv-unqualified integral or floating-point type except `bool` (3.9.1 [basic.fundamental]).
2. The second template argument `Abi` is an ABI tag so that `is_abi_tag_v<Abi>` is true.
3. The `Abi` type is a supported ABI tag. It is supported if
   - `Abi` is `simd_abi::scalar`, or
   - `Abi` is `simd_abi::fixed_size<N>` with `N ≤ 32` or implementation-defined additional valid values for `N` (see 5.1.1.1 p.3).

It is implementation-defined whether a given combination of `T` and an implementation-defined ABI tag is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. — end note]

[Example: Consider an implementation that defines the implementation-defined 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 except `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 results in a type with deleted constructor
- `simd<long double, simd_abi::simd_x>` is not supported and results in a type with 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]

The member type `reference` is an implementation-defined type acting as a reference to an element of type `value_type` with the following properties:
• The type has a deleted default constructor, copy constructor, and copy assignment operator.

• Assignment, compound assignment, increment, and decrement operators shall not participate in overload resolution unless the reference object is an rvalue and the corresponding operator of type value_type is usable.

• Objects of type reference are implicitly convertible to value_type.

• If a binary operator is applied to an object of type reference, the operator is only applied after converting the reference object to value_type.

• Calls to swap(reference &&, value_type &) and swap(value_type &, reference &&) exchange the values referred to by the reference object and the value_type reference. Calls to swap(reference &&, reference &&) exchange the values referred to by the reference objects.

static constexpr size_type size () noexcept;

5 Returns: the number of elements stored in objects of the given simd<T, Abi> type.

6 [ Note: Implementations are encouraged to enable static_casting from/to (an) implementation-defined SIMD type(s). This would add one or more of the following declarations to class simd:

```cpp
explicit operator implementation-defined() const;
explicit simd<const implementation-defined& init>;
```

— end note ]

5.1.2.2 simd constructors

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

1 Remarks: This constructor shall not participate in overload resolution unless either:

• U is an arithmetic type except bool and every possible value of type U can be represented with type value_type,

• or U is not an arithmetic type and is implicitly convertible to value_type,

• or U is int,

• or U is unsigned int and value_type is an unsigned integral type.

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

3 Throws: Any exception thrown while converting the argument to value_type.

template <class U> simd(const simd<U, simd_abi::fixed_size<size()>>& x);

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

• abi_type equals simd_abi::fixed_size<size>(),

• and every possible value of U can be represented with type value_type,

• and, if both U and value_type are integral, the integer conversion rank [N4618, (4.15)] of value_type is greater than the integer conversion rank of U.

5 Effects: Constructs an object where the i-th element equals static_cast<T>(x[i]) for all i ∈ [0, size()).
template <class G> simd(G&& gen);

Remarks: This constructor shall not participate in overload resolution unless simd{gen(integral_constant<size_t, 0>())} is well-formed.

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

Remarks: The order of calls to \(\text{gen}\) is unspecified.

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

Requires: \(\text{size()}\) is less than or equal to the number of values pointed to by \(\text{mem}\).

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to \(\text{memory_alignment_v}<\text{simd}, U>\). If the Flags template parameter is of type flags::overaligned_tag<\(N\), the pointer value represents an address aligned to \(N\).

5.1.2.3 simd load function

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

Requires: \(\text{size()}\) is less than or equal to the number of values pointed to by \(\text{mem}\).

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to \(\text{memory_alignment_v}<\text{simd}, U>\). If the Flags template parameter is of type flags::overaligned_tag<\(N\), the pointer value represents an address aligned to \(N\).

5.1.2.4 simd store function

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

Requires: \(\text{size()}\) is less than or equal to the number of values pointed to by \(\text{mem}\).

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to \(\text{memory_alignment_v}<\text{simd}, U>\). If the Flags template parameter is of type flags::overaligned_tag<\(N\), the pointer value represents an address aligned to \(N\).
5.1.2.5 simd subscript operators

```cpp
reference operator[](size_type i);
```

1. Requires: The value of i is less than size().
2. Returns: A temporary object of type reference (see 5.1.2.1 p.4) with the following effects:
3. Effects: The assignment, compound assignment, increment, and decrement operators of reference execute the indicated operation on the i-th element of the simd object.
4. Effects: Conversion to value_type returns a copy of the i-th element.
5. Throws: Nothing.

```cpp
value_type operator[](size_type i) const;
```

6. Requires: The value of i is less than size().
7. Returns: A copy of the i-th element.
8. Throws: Nothing.

5.1.2.6 simd unary operators

```cpp
simd operator++();
```

1. Effects: Increments every element of *this by one.
2. Returns: An lvalue reference to *this after incrementing.
3. Remarks: Overflow semantics follow the same semantics as for T.

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

5. Effects: Increments every element of *this by one.
6. Returns: A copy of *this before incrementing.
7. Remarks: Overflow semantics follow the same semantics as for T.
8. Throws: Nothing.

```cpp
simd operator--();
```

9. Effects: Decrements every element of *this by one.
10. Returns: An lvalue reference to *this after decrementing.
11. Remarks: Underflow semantics follow the same semantics as for T.

```cpp
simd operator--(int);
```
Effects: Decrements every element of *this by one.

Returns: A copy of *this before decrementing.

Remarks: Underflow semantics follow the same semantics as for T.

Throws: Nothing.

mask_type operator!() const;

Returns: A simd_mask object with the i-th element set to !operator[](i) for all i ∈ [0, size()).

Throws: Nothing.

simd operator~() const;

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

Remarks: simd::operator~() shall not participate in overload resolution unless T is an integral type.

Throws: Nothing.

simd operator+() const;

Returns: A copy of *this

Throws: Nothing.

simd operator-() const;

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

Throws: Nothing.

5.1.3 simd non-member operations

5.1.3.1 simd binary operators

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

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

Returns: A simd object initialized with the results of the component-wise application of the indicated operator.

Throws: Nothing.
friend simd operator<(const simd & v, int n);
friend simd operator>(const simd & v, int n);

Remarks: Both operators shall not participate in overload resolution unless value_type is an unsigned integral type.

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 \( i \in [0, \text{size()} \).

Throws: Nothing.

5.1.3.2 simd compound assignment [simd.cassign]

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

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

Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

Throws: Nothing.

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

Remarks: Both operators shall not participate in overload resolution unless value_type is an unsigned integral type.

Effects: Performs the indicated shift by \( n \) operation on the \( u\)-th element of \( v \) for all \( i \in [0, \text{size()} \).\)

Returns: A reference to the first argument.

Throws: Nothing.

5.1.3.3 simd compare operators [simd.comparison]

friend mask_type operator==(const simd &, const simd &);
friend mask_type operator!=(const simd &, const simd &);
friend mask_type operator>=((const simd &, const simd &);
friend mask_type operator<=(const simd &, const simd &);
friend mask_type operator>=(const simd &, const simd &);
friend mask_type operator<(const simd &, const simd &);
friend mask_type operator>(const simd &, const simd &);
Returns: A simd_mask object initialized with the results of the component-wise application of the indicated operator.

Throws: Nothing.

5.1.3.4 simd reductions

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

Returns: GENERALIZED_SUM(binary_op, x.data[i], ...) for all \(i \in [0, \text{size}())\).

Requires: binary_op shall be callable with two arguments of type T or two arguments of type simd<T, A1>, where A1 may be different to Abi.

[Note: This overload of reduce does not require an initial value because x is guaranteed to be non-empty. — end note]

```
template <class BinaryOperation = std::plus<>, class M, class V>
typename V::value_type reduce(const where_expression<M, V>& x, typename V::value_type neutral_element = default_neutral_element, BinaryOperation binary_op = BinaryOperation());
```

Returns: If \(\text{none_of}(x.\text{mask})\), returns neutral_element. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all \(i \in \{j \in \mathbb{N}_0 | j < \text{size()} \land x.\text{mask}[j]\}\).

Requires: binary_op shall be callable with two arguments of type T or two arguments of type simd<T, A1>, where A1 may be different to Abi.

[Note: This overload of reduce requires a neutral value to enable a parallelized implementation: A temporary simd object initialized with neutral_element is conditionally assigned from x.data using x.mask. Subsequently, the parallelized reduction (without mask) is applied to the temporary object. — end note]

```
template <class T, class A> T hmin(const simd<T, A>& x);
```

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

Throws: Nothing.

```
template <class M, class V> T hmin(const where_expression<M, V>& x);
```

Returns: If \(\text{none_of}(x.\text{mask})\), 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 \(i \in [0, \text{size}())\).

Throws: Nothing.

```
template <class T, class A> T hmax(const simd<T, A>& x);
```

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

Throws: Nothing.

```
template <class M, class V> T hmax(const where_expression<M, V>& x);
```

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

Throws: Nothing.
Returns: If none_of(x.mask), the return value is numeric_limits<V::value_type>::min(). 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 i ∈ [0, size()).

Throws: Nothing.

5.1.3.5 simd casts

template <class T, class U, class A> /* see below*/ simd_cast(const simd<U, A>& x);

Remarks: Let To identify T::value_type if is_simd_v<T> or T otherwise.
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> or T::size() is equal to simd<U, A>::size().
Remarks: If is_simd_v<T>, the return type is T. Otherwise, if either U and T are equal or U and T are
integral types that only differ in signedness, the return type is simd<T, A>. Otherwise, the return type is
simd<T, simd_abi::fixed_size<simd <U, A>::size>>.
Returns: A simd object with the i-th element initialized to static_cast<To>{x[i]}.
Throws: Nothing.

template <class T, class U, class A> /* see below*/ static_simd_cast(const simd<U, A>& x);

Remarks: Let To identify T::value_type if is_simd_v<T> or T otherwise.
Remarks: The function shall not participate in overload resolution unless either !is_simd_v<T> or T::size()
is equal to simd<U, A>::size().
Remarks: If is_simd_v<T>, the return type is T. Otherwise, if either U and T are equal or U and T are
integral types that only differ in signedness, the return type is simd<T, A>. Otherwise, the return type is
simd<T, simd_abi::fixed_size<simd <U, A>::size>>.
Returns: A simd object with the i-th element initialized to static_cast<To>{x[i]}.
Throws: Nothing.

template <class T, class A>
simd<T, simd_abi::fixed_size<simd_size_v<T>, A>> to_fixed_size(const simd<T, A>& x) noexcept;

template <class T, class A>
mask<T, simd_abi::fixed_size<simd_size_v<T>, A>> to_fixed_size(const simd_mask<T, A>& x) noexcept;

Returns: An object with the i-th element initialized to x[i].

template <class T, size_t N>
simd<T, simd_abi::native<T>> to_native(const simd<T, simd_abi::fixed_size<N>>& x) noexcept;

template <class T, size_t N>
mask<T, simd_abi::native<T>> to_native(const simd_mask<T, simd_abi::fixed_size<N>>& x) noexcept;

Remarks: These functions shall not participate in overload resolution unless simd_size_v<T, simd_abi::native<T>> is equal to N.
Returns: An object with the i-th element initialized to x[i].
template <class T, size_t N>
simd<T, simd::compatible<T>> to_compatible(const simd<T, simd::fixed_size<N>>& x) noexcept;

template <class T, size_t N>
mask<T, simd::compatible<T>> to_compatible(const simd_mask<T, simd::fixed_size<N>>& x) noexcept;

Remarks: These functions shall not participate in overload resolution unless simd_size_v<T, simd::compatible<T>> is equal to N.

Returns: An object with the i-th element initialized to x[i].

template <size_t... Sizes, class T, class A>
tuple<simd<T, abi::for_size_t<Sizes>>...> split(const simd<T, A>& x);

template <size_t... Sizes, class T, class A>
tuple<simd_mask<T, abi::for_size_t<Sizes>>...> split(const simd_mask<T, A>& x);

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, A>.

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

template <class V, class A>
array<V, simd::size_v<typename V::value_type, A>> split(const simd<typename V::value_type, A>& x);

template <class V, class A>
array<V, simd::size_v<typename V::value_type, A>> split(const simd_mask<typename V::value_type, A>& x);

Remarks: These functions shall not participate in overload resolution unless
• is_simd_v<V> for the first signature / is_mask_v<V> for the second signature,
• and simd_size_v<typename V::value_type, A> is an integral multiple of V::size().

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

template <class T, class... As>
simd<T, abi::for_size_t<T, (simd_size_v<T, As> + ...)>> concat(const simd<T, As>&... xs);

template <class T, class... As>
simd_mask<T, abi::for_size_t<T, (simd_size_v<T, As> + ...)>> concat(const simd_mask<T, As>&... xs);

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

5.1.3.6 simd algorithms

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

Returns: An object with the i-th element initialized with the smaller value of a[i] and b[i] for all i ∈ [0, size()].
template <class T, class A>
simd<T, A> max(const simd<T, A>&, const simd<T, A>&) noexcept;

Returns: An object with the i-th element initialized with the larger value of a[i] and b[i] for all i ∈ [0, size()).

template <class T, class A>
std::pair<simd<T, A>, simd<T, A>> minmax(const simd<T, A>&, const simd<T, A>&) noexcept;

Returns: An object with the i-th element in the first pair member initialized with the smaller value of a[i] and b[i] for all i ∈ [0, size()). The i-th element in the second pair member is initialized with the larger value of a[i] and b[i] for all i ∈ [0, size()).

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

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

Returns: An object with the i-th element initialized with lo[i] if v[i] is smaller than lo[i], hi[i] if v[i] is larger than hi[i], otherwise v[i] for all i ∈ [0, size()).

5.1.3.7 simd math library [simd.math]

namespace std {
namespace experimental {

// exposition only
template <class Abi> using scharv = simd<signed char, Abi>;

// exposition only
template <class Abi> using shortv = simd<short, Abi>;

// exposition only
template <class Abi> using intv = simd<int, Abi>;

// exposition only
template <class Abi> using longv = simd<long int, Abi>;

// exposition only
template <class Abi> using llongv = simd<long long int, Abi>;

// exposition only
template <class Abi> using floatv = simd<float, Abi>;

// exposition only
template <class Abi> using doublev = simd<double, Abi>;

// exposition only
template <class Abi> using ldoublev = simd<long double, Abi>;

// exposition only
template <class V> using samesize = fixed_size_simd<T, V::size()>; // exposition only

template <class Abi> floatv<Abi> acos(floatv<Abi> x);

template <class Abi> doublev<Abi> acos(doublev<Abi> x);

template <class Abi> ldoublev<Abi> acos(ldoublev<Abi> x);

template <class Abi> floatv<Abi> asin(floatv<Abi> x);

template <class Abi> doublev<Abi> asin(doublev<Abi> x);

template <class Abi> ldoublev<Abi> asin(ldoublev<Abi> x);

template <class Abi> floatv<Abi> atan(floatv<Abi> x);

template <class Abi> doublev<Abi> atan(doublev<Abi> x);

template <class Abi> ldoublev<Abi> atan(ldoublev<Abi> x);

template <class Abi> floatv<Abi> atan2(floatv<Abi> y, floatv<Abi> x);

template <class Abi> doublev<Abi> atan2(doublev<Abi> y, doublev<Abi> x);

template <class Abi> ldoublev<Abi> atan2(ldoublev<Abi> y, ldoublev<Abi> x);

template <class Abi> floatv<Abi> cos(floatv<Abi> x);

template <class Abi> doublev<Abi> cos(doublev<Abi> x);

template <class Abi> ldoublev<Abi> cos(ldoublev<Abi> x);

34
template <class Abi> ldoublev<Abi> cos(ldoublev<Abi> x);
template <class Abi> floatv<Abi> sin(floatv<Abi> x);
template <class Abi> doublev<Abi> sin(doublev<Abi> x);
template <class Abi> ldoublev<Abi> sin(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tan(floatv<Abi> x);
template <class Abi> doublev<Abi> tan(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tan(ldoublev<Abi> x);

template <class Abi> floatv<Abi> acosh(floatv<Abi> x);
template <class Abi> doublev<Abi> acosh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> acosh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> asinh(floatv<Abi> x);
template <class Abi> doublev<Abi> asinh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> asinh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> atanh(floatv<Abi> x);
template <class Abi> doublev<Abi> atanh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> atanh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> cosh(floatv<Abi> x);
template <class Abi> doublev<Abi> cosh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> cosh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> sinh(floatv<Abi> x);
template <class Abi> doublev<Abi> sinh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> sinh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tanh(floatv<Abi> x);
template <class Abi> doublev<Abi> tanh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tanh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp(floatv<Abi> x);
template <class Abi> doublev<Abi> exp(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp2(floatv<Abi> x);
template <class Abi> doublev<Abi> exp2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> expm1(floatv<Abi> x);
template <class Abi> doublev<Abi> expm1(doublev<Abi> x);
template <class Abi> ldoublev<Abi> expm1(ldoublev<Abi> x);

template <class Abi> floatv<Abi> frexp(floatv<Abi> value, samesize<int, floatv<Abi>>* exp);
template <class Abi> doublev<Abi> frexp(doublev<Abi> value, samesize<int, doublev<Abi>>* exp);
template <class Abi> ldoublev<Abi> frexp(ldoublev<Abi> value, samesize<int, ldoublev<Abi>>* exp);

template <class Abi> samesize<int, floatv<Abi>> ilogb(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> ilogb(doublev<Abi> x);
template <class Abi> samesize<int, ldoublev<Abi>> ilogb(ldoublev<Abi> x);

template <class Abi> samesize<int, floatv<Abi>> ldexp(floatv<Abi> x, samesize<int, floatv<Abi>> exp);
template <class Abi> samesize<int, doublev<Abi>> ldexp(doublev<Abi> x, samesize<int, doublev<Abi>> exp);
template <class Abi> samesize<int, ldoublev<Abi>> ldexp(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> exp);
template <class Abi> floatv<Abi> log(floatv<Abi> x);
template <class Abi> doublev<Abi> log(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log10(floatv<Abi> x);
template <class Abi> doublev<Abi> log10(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log10(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log1p(floatv<Abi> x);
template <class Abi> doublev<Abi> log1p(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log1p(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log2(floatv<Abi> x);
template <class Abi> doublev<Abi> log2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> logb(floatv<Abi> x);
template <class Abi> doublev<Abi> logb(doublev<Abi> x);
template <class Abi> ldoublev<Abi> logb(ldoublev<Abi> x);

template <class Abi> floatv<Abi> modf(floatv<Abi> value, floatv<Abi>* iptr);
template <class Abi> doublev<Abi> modf(doublev<Abi> value, doublev<Abi>* iptr);
template <class Abi> ldoublev<Abi> modf(ldoublev<Abi> value, ldoublev<Abi>* iptr);

template <class Abi> floatv<Abi> scalbn(floatv<Abi> x, samesize<int, floatv<Abi>> n);
template <class Abi> doublev<Abi> scalbn(doublev<Abi> x, samesize<int, doublev<Abi>> n);
template <class Abi> ldoublev<Abi> scalbn(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> n);

template <class Abi> floatv<Abi> cbrt(floatv<Abi> x);
template <class Abi> doublev<Abi> cbrt(doublev<Abi> x);
template <class Abi> ldoublev<Abi> cbrt(ldoublev<Abi> x);

template <class Abi> scharv<Abi> abs(scharv<Abi> j);
template <class Abi> shortv<Abi> abs(shortv<Abi> j);
template <class Abi> intv<Abi> abs(intv<Abi> j);
template <class Abi> longv<Abi> abs(longv<Abi> j);
template <class Abi> llongv<Abi> abs(llongv<Abi> j);
template <class Abi> floatv<Abi> abs(floatv<Abi> j);
template <class Abi> doublev<Abi> abs(doublev<Abi> j);
template <class Abi> ldoublev<Abi> abs(ldoublev<Abi> j);

template <class Abi> floatv<Abi> hypot(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> hypot(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> hypot(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> pow(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> pow(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> pow(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> sqrt(floatv<Abi> x);
template <class Abi> doublev<Abi> sqrt(doublev<Abi> x);
template <class Abi> doublev<Abi> sqrt(ldoublev<Abi> x);

template <class Abi> floatv<Abi> erf(floatv<Abi> x);
template <class Abi> doublev<Abi> erf(doublev<Abi> x);
template <class Abi> ldoublev<Abi> erf(ldoublev<Abi> x);

template <class Abi> floatv<Abi> erfc(floatv<Abi> x);
template <class Abi> doublev<Abi> erfc(doublev<Abi> x);
template <class Abi> ldoublev<Abi> erfc(ldoublev<Abi> x);

template <class Abi> floatv<Abi> lgamma(floatv<Abi> x);
template <class Abi> doublev<Abi> lgamma(doublev<Abi> x);
template <class Abi> ldoublev<Abi> lgamma(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tgamma(floatv<Abi> x);
template <class Abi> doublev<Abi> tgamma(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tgamma(ldoublev<Abi> x);

template <class Abi> floatv<Abi> ceil(floatv<Abi> x);
template <class Abi> doublev<Abi> ceil(doublev<Abi> x);
template <class Abi> ldoublev<Abi> ceil(ldoublev<Abi> x);

template <class Abi> floatv<Abi> floor(floatv<Abi> x);
template <class Abi> doublev<Abi> floor(doublev<Abi> x);
template <class Abi> ldoublev<Abi> floor(ldoublev<Abi> x);

template <class Abi> floatv<Abi> nearbyint(floatv<Abi> x);
template <class Abi> doublev<Abi> nearbyint(doublev<Abi> x);
template <class Abi> ldoublev<Abi> nearbyint(ldoublev<Abi> x);

template <class Abi> floatv<Abi> round(floatv<Abi> x);
template <class Abi> doublev<Abi> round(doublev<Abi> x);
template <class Abi> ldoublev<Abi> round(ldoublev<Abi> x);

template <class Abi> samesize<long int, floatv<Abi>> lrint(floatv<Abi> x);
template <class Abi> samesize<long int, doublev<Abi>> lrint(doublev<Abi> x);
template <class Abi> samesize<long int, ldoublev<Abi>> lrint(ldoublev<Abi> x);

template <class Abi> samesize<long long int, floatv<Abi>> llrint(floatv<Abi> x);
template <class Abi> samesize<long long int, doublev<Abi>> llrint(doublev<Abi> x);
template <class Abi> samesize<long long int, ldoublev<Abi>> llrint(ldoublev<Abi> x);

template <class Abi> floatv<Abi> trunc(floatv<Abi> x);
template <class Abi> doublev<Abi> trunc(doublev<Abi> x);
template <class Abi> ldoublev<Abi> trunc(ldoublev<Abi> x);
template <class Abi> floatv<Abi> fmod(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmod(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmod(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> remainder(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> remainder(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> remainder(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> remquo(floatv<Abi> x, floatv<Abi> y, samesize<int, floatv<Abi>>* quo);
template <class Abi> doublev<Abi> remquo(doublev<Abi> x, doublev<Abi> y, samesize<int, doublev<Abi>>* quo);
template <class Abi> ldoublev<Abi> remquo(ldoublev<Abi> x, ldoublev<Abi> y, samesize<int, ldoublev<Abi>>* quo);

template <class Abi> floatv<Abi> copysign(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> copysign(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> copysign(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> doublev<Abi> nan(const char* tagp);
template <class Abi> floatv<Abi> nanf(const char* tagp);
template <class Abi> ldoublev<Abi> nanl(const char* tagp);

template <class Abi> floatv<Abi> nextafter(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> nextafter(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> nextafter(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> nexttoward(floatv<Abi> x, ldoublev<Abi> y);
template <class Abi> doublev<Abi> nexttoward(doublev<Abi> x, ldoublev<Abi> y);
template <class Abi> ldoublev<Abi> nexttoward(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fdim(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fdim(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fdim(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmax(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmax(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmax(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmin(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmin(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmin(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fma(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);
template <class Abi> doublev<Abi> fma(doublev<Abi> x, doublev<Abi> y, doublev<Abi> z);
template <class Abi> ldoublev<Abi> fma(ldoublev<Abi> x, ldoublev<Abi> y, ldoublev<Abi> z);

template <class Abi> samesize<int, floatv<Abi>> fpclassify(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> fpclassify(doublev<Abi> x);

template <class Abi> simd_mask<float, Abi> isfinite(floatv<Abi> x);
template <class Abi> simd_mask<double, Abi> isfinite(doublev<Abi> x);

template <class Abi> samesize<int, floatv<Abi>> isnan(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> isnan(doublev<Abi> x);

template <class Abi> samesize<int, floatv<Abi>> isinf(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> isinf(doublev<Abi> x);

template <class Abi> samesize<int, ldoublev<Abi>> isnan(ldoublev<Abi> x);
template <class Abi> samesize<int, ldoublev<Abi>> isinf(ldoublev<Abi> x);
template <class Abi> simd_mask<float, Abi> isnan(floatv<Abi> x);
template <class Abi> simd_mask<double, Abi> isnan(doublev<Abi> x);
template <class Abi> simd_mask<long double, Abi> isnan(ldoublev<Abi> x);

template <class Abi> simd_mask<float, Abi> isnormal(floatv<Abi> x);
template <class Abi> simd_mask<double, Abi> isnormal(doublev<Abi> x);
template <class Abi> simd_mask<long double, Abi> isnormal(ldoublev<Abi> x);

template <class Abi> simd_mask<float, Abi> signbit(floatv<Abi> x);
template <class Abi> simd_mask<double, Abi> signbit(doublev<Abi> x);
template <class Abi> simd_mask<long double, Abi> signbit(ldoublev<Abi> x);

template <class Abi> simd_mask<float, Abi> isgreater(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> isgreater(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> isgreater(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> simd_mask<float, Abi> isgreaterequal(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> isgreaterequal(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> isgreaterequal(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> simd_mask<float, Abi> isless(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> isless(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> isless(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> simd_mask<float, Abi> islessequal(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> islessequal(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> islessequal(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> simd_mask<float, Abi> islessgreater(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> islessgreater(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> islessgreater(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> simd_mask<float, Abi> isunordered(floatv<Abi> x, floatv<Abi> y);
template <class Abi> simd_mask<double, Abi> isunordered(doublev<Abi> x, doublev<Abi> y);
template <class Abi> simd_mask<long double, Abi> isunordered(ldoublev<Abi> x, ldoublev<Abi> y);

template <class V> struct simd_div_t { V quot, rem; };

template <class Abi> simd_div_t<scharv<Abi>> div(scharv<Abi> numerator, scharv<Abi> denominator);
template <class Abi> simd_div_t<shortv<Abi>> div(shortv<Abi> numerator, shortv<Abi> denominator);
template <class Abi> simd_div_t<intv<Abi>> div(intv<Abi> numerator, intv<Abi> denominator);
template <class Abi> simd_div_t<longv<Abi>> div(longv<Abi> numerator, longv<Abi> denominator);

1 Each listed function concurrently applies the indicated mathematical function component-wise. The results per component are not required to be binary equal to the application of the function which is overloaded for the element type. [Note: If a precondition of the indicated mathematical function is violated, the behavior is undefined. —end note]

2 If abs() is called with an argument of type simd<X, Abi> for which is_unsigned<X>::value is true, the program is ill-formed.

5.1.4 Class template simd_mask
5.1.4.1 Class template simd_mask overview

namespace std {
namespace experimental {
    template <class T, class Abi> class simd_mask {
        using value_type = bool;
        using reference = implementation-defined; // see simd::reference
        using simd_type = simd<T, Abi>;
        using size_type = size_t;
        using abi_type = Abi;

        static constexpr size_type size () noexcept;
        simd_mask () = default;
        simd_mask (const simd_mask &) = default;
        simd_mask (simd_mask &&) = default;
        simd_mask & operator=(const simd_mask &) = default;
        simd_mask & operator=(simd_mask &&) = default;

        // broadcast constructor
        explicit simd_mask(value_type ) noexcept;

        // implicit type conversion constructor
        template <class U> simd_mask (const simd_mask <U, simd_abi::fixed_size<size ()>>&) noexcept;

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

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

        // stores
        template <class Flags> void copy_to (value_type* mem, Flags ) const;

        // scalar access
        reference operator[](size_type);
        value_type operator[](size_type) const;

        // unary operators
        simd_mask operator!() const noexcept;

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

        // simd_mask compound assignment
        friend simd_mask operator+=(const simd_mask&, const simd_mask&) noexcept;
        friend simd_mask operator-=(const simd_mask&, const simd_mask&) noexcept;
        friend simd_mask operator*=(const simd_mask&, const simd_mask&) noexcept;
        friend simd_mask operator/=(const simd_mask&, const simd_mask&) noexcept;

        // simd_mask compares
        friend simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
        friend simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;
    }
}
}
The class template `simd_mask<T, Abi>` is a one-dimensional smart array of booleans. The number of elements in the array is a constant expression, equal to the number of elements in `simd<T, Abi>`.

The resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment unless all of the following hold:

- The first template argument `T` is a cv-unqualified integral or floating-point type except `bool` (3.9.1 [basic.fundamental]).
- The second template argument `Abi` is an ABI tag so that `is_abi_tag_v<Abi>` is true.
- The `Abi` type is a supported ABI tag. It is supported if
  - `Abi` is `simd_abi::scalar`, or
  - `Abi` is `simd_abi::fixed_size<N>` with `N \leq 32` or implementation-defined additional valid values for `N` (see 5.1.1.1 p.3).

It is implementation-defined whether a given combination of `T` and an implementation-defined ABI tag is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. — end note]

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

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

Returns: the number of boolean elements stored in objects of the given `simd_mask<T, Abi>` type.

[Note: Implementations are encouraged to enable static_casting from/to (an) implementation-defined SIMD mask type(s). This would add one or more of the following declarations to class `simd_mask`:

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

— end note]

5.1.4.2 `simd_mask` constructors

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

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

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

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

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

```cpp
template <class Flags> simd_mask(const value_type *mem, Flags);
```
4 \textit{Effects:} Constructs an object where the $i$-th element is initialized to $\text{mem}[i]$ for all $i \in [0, \text{size()})$.

5 \textit{Requires:} $\text{size()}$ is less than or equal to the number of values pointed to by $\text{mem}$.

6 \textit{Requires:} If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to $\text{memory_alignment_v<simd_mask>}$.

5.1.4.3 \textit{simd_mask load function} \hspace{1cm} [simd\_mask.load]

\texttt{template <class Flags> void copy_from(const value\_type* mem, Flags);}

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

2 \textit{Requires:} $\text{size()}$ is less than or equal to the number of values pointed to by $\text{mem}$.

3 \textit{Requires:} If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to $\text{memory_alignment_v<simd_mask>}$.

5.1.4.4 \textit{simd_mask store function} \hspace{1cm} [simd\_mask.store]

\texttt{template <class Flags> void copy_to(value\_type* mem, Flags);};

1 \textit{Effects:} Copies all $\text{simd\_mask}$ elements as if $\text{mem}[i] = \text{operator[]}(i)$ for all $i \in [0, \text{size()})$.

2 \textit{Requires:} $\text{size()}$ is less than or equal to the number of values pointed to by $\text{mem}$.

3 \textit{Requires:} If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to $\text{memory_alignment_v<simd_mask>}$.

5.1.4.5 \textit{simd_mask subscript operators} \hspace{1cm} [simd\_mask.subscr]

\texttt{reference operator[](size\_type i);};

1 \textit{Requires:} The value of $i$ is less than $\text{size()}$.

2 \textit{Returns:} A temporary object of type reference (see 5.1.2.1 p.4) with the following effects:

3 \textit{Effects:} The assignment, compound assignment, increment, and decrement operators of reference execute the indicated operation on the $i$-th element of the $\text{simd\_mask}$ object.

4 \textit{Effects:} Conversion to value\_type returns a copy of the $i$-th element.

5 \textit{Threws:} Nothing.

\texttt{value\_type operator[](size\_type i) const;};

6 \textit{Requires:} The value of $i$ is less than $\text{size()}$.

7 \textit{Returns:} A copy of the $i$-th element.

8 \textit{Threws:} Nothing.
5.1.4.6 simd_mask unary operators

_mask operator!() const noexcept;

1 Returns: A simd_mask object with the \( i \)-th element set to the logical negation for all \( i \in [0, \text{size()} \right)."

5.1.5 simd_mask non-member operations

5.1.5.1 simd_mask binary operators

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

1 Returns: A simd_mask object initialized with the results of the component-wise application of the indicated operator.

5.1.5.2 simd_mask compound assignment

friend simd_mask & operator&=(simd_mask &, const simd_mask &)
friend simd_mask & operator|=(simd_mask &, const simd_mask &)
friend simd_mask & operator^=(simd_mask &, const simd_mask &)

1 Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

5.1.5.3 simd_mask compares

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

1 Returns: A simd_mask object initialized with the results of the component-wise application of the indicated operator.

5.1.5.4 simd_mask reductions

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

1 Returns: true if all boolean elements in the function argument equal true, false otherwise.

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

2 Returns: true if at least one boolean element in the function argument equals true, false otherwise.
template <class T, class Abi> bool none_of(simd_mask<T, Abi>) noexcept;

Returns: true if none of the boolean element in the function argument equals true, false otherwise.

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

Returns: true if at least one of the boolean elements in the function argument equals true and at least one of the boolean elements in the function argument equals false, false otherwise.

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

Returns: The number of boolean elements that are true.

template <class T, class Abi> int find_first_set(simd_mask<T, Abi> m);

Requires: any_of(m) returns true

Returns: The lowest element index i where m[i] == true.

template <class T, class Abi> int find_last_set(simd_mask<T, Abi> m);

Requires: any_of(m) returns true

Returns: The highest element index i where m[i] == true.

bool all_of(implementation-defined) noexcept;
bool any_of(implementation-defined) noexcept;
bool none_of(implementation-defined) noexcept;
bool some_of(implementation-defined) noexcept;
int popcount(implementation-defined) noexcept;
int find_first_set(implementation-defined) noexcept;
int find_last_set(implementation-defined) noexcept;

Remarks: The functions shall not participate in overload resolution unless the argument is of type bool.

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; find_first_set and find_last_set return 0.

5.1.5.5 Masked assignment

template <class T, class A>
where_expression<simd_mask<T, A>, simd<T, A>> where{
    const typename simd<T, A>::mask_type & k, simd<T, A>& v) noexcept;

template <class T, class A>
const where_expression<simd_mask<T, A>, const simd<T, A>> where{
    const typename simd<T, A>::mask_type & k, const simd<T, A>& v) noexcept;

template <class T, class A>
where_expression<simd_mask<T, A>, simd_mask<T, A>> where{
    const remove_const_t<simd_mask<T, A>>& k, simd_mask<T, A>& v) noexcept;

template <class T, class A>
const where_expression<simd_mask<T, A>, const simd_mask<T, A>> where{
    const remove_const_t<simd_mask<T, A>> & k, const simd_mask<T, A>& v) noexcept;
6 Discussion

Note 7: `remove_const` is only used in place of a missing template id | using type = T; || for inhibiting type deduction.

1 Returns: An object of type `where_expression` (see 5.1.1.3) initialized with the predicate `k` and the value reference `v`.

```cpp
template <class T> struct id { using type = T; }
```

2 Remarks: The function shall not participate in overload resolution unless
- `T` is neither a `simd` nor a `simd_mask` instantiation, and
- the first argument is of type `bool`.

3 Returns: An object of type `where_expression` (see 5.1.1.3) initialized with the predicate `k` and the value reference `v`.

6 Discussion

6.1 Member Types

The member types may not seem obvious. Rationales:

value_type
In the spirit of the `value_type` member of STL containers, this type denotes the \textit{logical} type of the values in the vector.

reference
Used as the return type of the non-const scalar subscript operator.

mask_type
The natural `simd_mask` type for this `simd` instantiation. This type is used as return type of compares and write-mask on assignments.

simd_type
The natural `simd` type for this `simd_mask` instantiation.

size_type
Standard member type used for `size()` and `operator[]`.

abi_type
The Abi template parameter to `simd`.
6.2 conversions

The simd conversion constructor only allows implicit conversion from simd template instantiations with the same Abi type and compatible value type. Discussion in SG1 showed clear preference for only allowing implicit conversion between integral types that only differ in signedness. All other conversions could be implemented via an explicit conversion constructor. The alternative (preferred) is to use simd_cast consistently for all other conversions.

After more discussion on the LEWG reflector, in Issaquah, and between me and Jens, we modified conversions to be even more conservative. No implicit conversion will ever allow a narrowing conversion of the element type (and signed - unsigned is narrowing in both directions). Also, implicit conversions are only enabled between fixed_size instances. All other ABI tags require explicit conversions in every case. This is very conservative and I expect TS feedback on this issue to ask for more (non-narrowing) implicit conversions.

6.3 broadcast constructor

The simd broadcast constructor is not declared explicit to ease the use of scalar prvalues in expressions involving data-parallel operations. The operations where such a conversion should not be implicit consequently need to use SFINAE / concepts to inhibit the conversion.

Experience from Vc shows that the situation is different for simd_mask, where an implicit conversion from bool typically hides an error. (Since there is little use for broadcasting true or false.)

6.4 aliasing of subscript operators

The subscript operators return an rvalue. The const overload returns a copy of the element. The non-const overload returns a smart reference. This reference behaves mostly like an lvalue reference, but without the requirement to implement assignment via type punning. At this point the specification of the smart reference is very conservative / restrictive: The reference type is neither copyable nor movable. The intention is to avoid users to program like the operator returned an lvalue reference. The return type is significantly larger than an lvalue reference and harder to optimize when passed around. The restriction thus forces users to do element modification directly on the simd/ simd_mask objects.
Guidance from SG1 at JAX 2016:
Poll: Should subscript operator return an lvalue reference?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Should subscript operator return a “smart reference”?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

6.5 Compound Assignment

The semantics of compound assignment would allow less strict implicit conversion rules. Consider `simd<int>() *= double()`: the corresponding binary multiplication operator would not compile because the implicit conversion to `simd<double>` is non-portable. Compound assignment, on the other hand, implies an implicit conversion back to the type of the expression on the left of the assignment operator. Thus, it is possible to define compound operators that execute the operation correctly on the promoted type without sacrificing portability. There are two arguments for not relaxing the rules for compound assignment, though:

1. Consistency: The conversion of an expression with compound assignment to a binary operator might make it ill-formed.

2. The implicit conversion in the `int * double` case could be expensive and unintended. This is already a problem for built-in types, where many developers multiply `float` variables with `double` prvalues, though.

6.6 Return Type of Masked Assignment Operators

The assignment operators of the type returned by `where(mask, simd)` could return one of:

- A reference to the `simd` object that was modified.

- A temporary `simd` object that only contains the elements where the `simd_mask` is true.

- A reference to the `where_expression` object.

- Nothing (i.e. `void`).
Listing 1: Possible declaration of the class template parameters of a `simd` class with arbitrary width.

My first choice was a reference to the modified `simd` object. However, then the statement `where(x < 0, x) *= -1 += 2` may be surprising: it adds 2 to all vector entries, independent of the mask. Likewise, `y += (where(x < 0, x) *= -1)` has a possibly confusing interpretation because of the `simd_mask` in the middle of the expression.

Consider that write-masked assignment is used as a replacement for `if`-statements. Using `void` as return type therefore is a more fitting choice because `if`-statements have no return value. By declaring the return type as `void` the above expressions become ill-formed, which seems to be the best solution for guiding users to write maintainable code and express intent clearly.

### 6.7 Fundamental SIMD Type or Not?

#### The Issue

There was substantial discussion on the reflectors and SG1 meetings over the question whether C++ should define a fundamental, native SIMD type (let us call it `fundamental<T>`) and additionally a generic data-parallel type which supports an arbitrary number of elements (call it `arbitrary<T, N>`). The alternative to defining both types is to only define `arbitrary<T, N = default_size<T>>`, since it encompasses the `fundamental<T>` type.

With regard to this proposal this second approach would add a third template parameter to `simd` and `simd_mask` as shown in Listing 1.

#### Standpoints

The controversy is about how the flexibility of a type with arbitrary `N` is presented to the users. Is there a (clear) distinction between a “fundamental” type with target-dependent (i.e. fixed) `N` and a higher-level abstraction with arbitrary `N` which can potentially compile to inefficient machine code? Or should the C++ standard only define `arbitrary` and set it to a default `N` value that corresponds to the target-dependent `N`. Thus, the default `N`, of `arbitrary` would correspond to `fundamental`.

It is interesting to note that `arbitrary<T, 1>` is the class variant of `T`. Consequently, if we say there is no need for a `fundamental` type then we could argue for
the deprecation of the builtin arithmetic types, in favor of arbitrary<T, 1>. [Note: This is an academic discussion, of course. — end note]

The author has implemented a library where a clear distinction is made between fundamental<T, Abi> and arbitrary<T, N>. The documentation and all teaching material says that the user should program with fundamental. The arbitrary type should be used in special circumstances, or wherever fundamental works with the arbitrary type in its interfaces (e.g. for gather & scatter or the ldexp & frexp functions).

6.7.3 Issues

The definition of two separate class templates can alleviate some source compatibility issues resulting from different \(N\) on different target systems. Consider the simplest example of a multiplication of an int vector with a float vector:

```cpp
arbitrary<float>() * arbitrary<int>(); // compiles for some targets, fails for others
fundamental<float>() * fundamental<int>(); // never compiles, requires explicit cast
```

The simd<T> operators are specified in such a way that source compatibility is ensured. For a type with user definable \(N\), the binary operators should work slightly different with regard to implicit conversions. Most importantly, arbitrary<T, N> solves the issue of portable code containing mixed integral and floating-point values. A user would typically create aliases such as:

```cpp
using floatvec = simd<float>;
using intvec = arbitrary<int, floatvec::size>();
using doublevec = arbitrary<int, floatvec::size>();
```

Objects of types floatvec, intvec, and doublevec will work together, independent of the target system.

Obviously, these type aliases are basically the same if the \(N\) parameter of arbitrary has a default value:

```cpp
using floatvec = arbitrary<float>;
using intvec = arbitrary<int, floatvec::size>();
using doublevec = arbitrary<int, floatvec::size>();
```

The ability to create these aliases is not the issue. Seeing the need for using such a pattern is the issue. Typically, a developer will think no more of it if his code compiles on his machine. If arbitrary<float>() * arbitrary<int>() just happens to compile (which is likely), then this is the code that will get checked in to the repository. Note that with the existence of the fundamental class template, the \(N\) parameter of the arbitrary class would not have a default value and thus force the user to think a second longer about portability.
SG1 Guidance at JAX 2016:
Poll: Specify simd width using ABI tag, with a special template tag for fixed size.

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

Poll: Specify simd width using \(<T, N, \text{abi}\rangle\), where \(\text{abi}\) is not specified by the user.

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

At the Jacksonville meeting, SG1 decided to continue with the \(\text{simd}<T, \text{Abi}\rangle\) class template, with the addition of a new Abi type that denotes a user-requested number of elements in the vector (\(\text{simd\_abi}\::\text{fixed\_size}<N\rangle\)). This has the following implications:

- There is only one class template with a common interface for \textit{fundamental} and \textit{arbitrary} (fixed\_size) vector types.

- There are slight differences in the conversion semantics for \textit{simd} types with the fixed\_size Abi type. This may look like the \texttt{vector<bool>} mistake all over again. I’ll argue below why I believe this is not the case.

- The \textit{fundamental} class instances could be implemented in such a way that they do not guarantee ABI compatibility on a given architecture where translation units are compiled with different compiler flags (for micro-architectural differences).

- The fixed\_size class instances, on the other hand, could be implemented to be the ABI stable types (if an implementation thinks this is an important feature). In implementation terms this means that \textit{fundamental} types are allowed to be passed via registers on function calls. \textit{fixed\_size} types can be implemented in such a way that they are only passed via the stack, and thus an implementation only needs to ensure equal alignment and memory representation across TU borders for a given \(T, N\).

The conversion differences between the \textit{fundamental} and \textit{fixed\_size} class template instances are the main motivation for having a distinction (cf. discussion above). The differences are chosen such that, in general, \textit{fundamental} types are more restrictive and do not turn into \textit{fixed\_size} types on any operation that involves no \textit{fixed\_size} types. Operations of \textit{fixed\_size} types allow easier use of mixed precision code as long as no elements need to be dropped / generated (i.e. the number of elements of all involved simd objects is equal or a builtin arithmetic type is broadcast).
Examples:

1. **Mixed int–float operations**

   ```cpp
   using floatv = simd<float>; // native ABI
   using float_sized_abi = simd_abi::fixed_size<floatv::size>();
   using intv = simd<int, float_sized_abi>;
   auto x = floatv() + intv();
   intv y = floatv() + intv();
   ```

   Line 5 is well-formed: It states that \( N = \text{floatv::size()} \) additions shall be executed concurrently. The type of \( x \) is \( \text{simd<float>} \), because it stores \( N \) elements and both types \( \text{intv} \) and \( \text{floatv} \) are implicitly convertible to \( \text{simd<float>} \). Line 6 is also well-formed because implicit conversion from \( \text{simd<T, Abi>} \) to \( \text{simd<U, simd_abi::fixed_size<N>)} \) is allowed whenever \( N = \text{simd<T, Abi>::size()} \).

2. **Native int vectors**

   ```cpp
   using intv = simd<int>; // native ABI
   using int_sized_abi = simd_abi::fixed_size<intv::size>();
   using floatv = simd<float, int_sized_abi>;
   auto x = floatv() + intv();
   intv y = floatv() + intv();
   ```

   Line 5 is well-formed: It states that \( N = \text{intv::size()} \) additions shall be executed concurrently. The type of \( x \) is \( \text{simd<float_v, int_sized_abi)} \) (i.e. \( \text{floatv} \)) and never \( \text{simd<float>} \), because ...

   ... the Abi types of \( \text{intv} \) and \( \text{floatv} \) are not equal.

   ... either \( \text{simd<float>::size()}! = N \) or \( \text{intv} \) is not implicitly convertible to \( \text{simd<float>} \).

   ... the last rule for \( \text{commonabi(V0, V1, T)} \) sets the Abi type to \( \text{int_sized_abi} \).

   Line 6 is also well-formed because implicit conversion from \( \text{simd<T, simd_abi::fixed_size<N>)} \) to \( \text{simd<U, Abi)} \) is allowed whenever \( N = \text{simd<U, Abi>::size()} \).

### 6.8 Native Handle

The presence of a native handle function for accessing an internal data member such as e.g. a vector builtin or SIMD intrinsic type is seen as an important feature for adoption in the target communities. Without such a handle the user is constrained to work within the (limited) API defined by the standard. Many SIMD instruction sets
have domain-specific instructions that will not easily be usable (if at all) via the stand-
dardized interface. A user considering whether to use simd or a SIMD extension such
as vector builtins or SIMD intrinsics might decide against simd just for fear of not
being able to access all functionality.¹

I would be happy to settle on an alternative to exposing an lvalue reference to
a data member. Consider implementation-defined support casting (static_cast?)
between simd and non-standard SIMD extension types. My understanding is that
there could not be any normative wording about such a feature. However, I think it
could be useful to add a non-normative note about making static_cast(?) able to
convert between such non-standard extensions and simd.

Guidance from SG1 at Oulu 2016:
Poll: Keep native_handle in the wording?

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

6.9 LOAD & STORE FLAGS

SIMD loads and stores require at least an alignment option. This is in contrast to
implicit loads and stores present in C++, where alignment is always assumed. Many
SIMD instruction sets allow more options, though:

• Streaming, non-temporal loads and stores

• Software prefetching

In the Vc library I have added these as options in the load store flag parameter of
the load and store functions. However, non-temporal loads & stores and prefetching
are also useful for the existing builtin types. I would like guidance on this question:
should the general direction be to stick to only alignment options for simd loads and
stores?

The other question is on the default of the load and store flags. Some argue for
setting the default to aligned, as that’s what the user should always aim for and is
most efficient. Others argue for unaligned since this is safe per default. The Vc library
before version 1.0 used aligned loads and stores per default. After the guidance from
SG1 I changed the default to unaligned loads and stores with the Vc 1.0 release.
Changing the default is probably the worst that could be done, though.² For Vc 2.0 I
will drop the default.

For simd I prefer no default:

---
¹ Whether that’s a reasonable fear is a different discussion.
² As I realized too late.
• This makes it obvious that the API has the alignment option. Users should not just take the default and think no more of it.

• If we decide to keep the load constructor, the alignment parameter (without default) nicely disambiguate the load from the broadcast.

• The right default would be application/domain/experience specific.

• Users can write their own load/store wrapper functions that implement their chosen default.

Guidance from SG1 at Oulu 2016:
Poll: Should the interface provide a way to specify a number for over-alignment?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Should loads and stores have a default load/store flag?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

The discussion made it clear that we only want to support alignment flags in the load and store operations. The other functionality is orthogonal.

6.10 unary minus return type

The return type of `simd<T, Abi>::operator-()` is `simd<T, Abi>`. This is slightly different to the behavior of the underlying element type `T`, if `T` is an integral type of lower integer conversion rank than `int`. In this case integral promotion promotes the type to `int` before applying unary minus. Thus, the expression `-T()` is of type `int` for all `T` with lower integer conversion rank than `int`. This is widening of the element size is likely unintended for SIMD vector types.

Fundamental types with integer conversion rank greater than `int` are not promoted and thus a unary minus expression has unchanged type. This behavior is copied to element types of lower integer conversion rank for `simd`.

There may be one interesting alternative to pursue here: We can make it ill-formed to apply unary minus to unsigned integral types. Anyone who wants to have the modulo behavior of a unary minus could still write `0u - x`.

6.11 max_fixed_size

In Kona, LEWG asked why `max_fixed_size` is not dependent on `T`. After some consideration I am convinced that the correct solution is to make `max_fixed_size` a variable template, dependent on `T`.
The reason to restrict the number of elements $N$ in a fixed-size simd type at all, is to inhibit misuse of the feature. The intended use of the fixed-size ABI, is to work with a number of elements that is somewhere in the region of the number of elements that can be processed efficiently concurrently in hardware. Implementations may want to use recursion to implement the fixed-size ABI. While such an implementation can, in theory, scale to any $N$, experience shows that compiler memory usage and compile times grow significantly for “too large” $N$. The optimizer also has a hard time to optimize register / stack allocation optimally if $N$ becomes “too large”. Unsuspecting users might not think of such issues and try to map their complete problem to a single simd object. Allowing implementations to restrict $N$ to a value that they can and want to support thus is useful for users and implementations. The value itself should not be prescribed by the standard as it is really just a QoI issue.

However, why should the user be able to query the maximum $N$ supported by the implementation?

- In principle, a user can always determine the number using SFINAE to find the maximum $N$ that he can still instantiate without substitution failure. Not providing the number thus provides no “safety” against “bad usage”.

- A developer may want to use the value to document assumptions / requirements about the implementation, e.g. with a static assertion.

- A developer may want to use the value to make code portable between implementations that use a different max_fixed_size.

Making the max_fixed_size dependent on $T$ makes sense because most hardware can process a different number of elements in parallel depending on $T$. Thus, if an implementation wants to restrict $N$ to some sensible multiple of the hardware capabilities, the number must be dependent on $T$.

In Kona, LEWG also asked whether there should be a provision in the standard to ensure that a native simd of 8-bit element type is convertible to a fixed-size simd of 64-bit element type. It was already there (5.1.1.1 p.3: “for every supported simd<T, A> (see 5.1.2.1 p.2), where A is an implementation-defined ABI tag, $N = \text{simd<T, A>::size()}$ must be supported”). Note that this does not place a lower bound on max_fixed_size. The wording allows implementations to support values for fixed-size simd that are larger than max_fixed_size, i.e. $N \leq \text{max_fixed_size}$ works; whether $N > \text{max_fixed_size}$ works is unspecified.
For the purposes of SD-6, feature detection initially will be provided through the shipping vehicle (TS) itself. For a standalone feature detection macro, I recommend __cpp_lib_simd. If LEWG decides to rename the simd class template, the feature detection macro needs to be renamed accordingly.

**ACKNOWLEDGEMENTS**

- This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

- Jens Maurer contributed important feedback and suggestions on the API. Thanks also for presenting the paper in Kona 2017 and Toronto 2017.

- Thanks to Hartmut Kaiser for presenting in Issaquah 2016.

- Geoffrey Romer did a very helpful review of the wording.

**BIBLIOGRAPHY**


