Document Number: P0214R8
Date: 2018-02-12
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LWG
Target: Parallelism TS 2

Data-Parallel Vector Types & Operations

ABSTRACT

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

CONTENTS

0 REMARKS

- This documents 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, $\mathcal{W}_{\mathtt{T}}$ 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 CHANGELOG

1.1 CHANGES FROM R7

Previous revision: [P0214R7].

- Removed the explicit signatures of the special math functions in favor of a rule to translate the `<cmath>` signatures to `simd` types. This resolves an over-specification, which disallowed certain valid implementation strategies. (see [simd.math] p.1)
- Added margin-note questioning non-normative note about special math function preconditions. (see [simd.math] p.2)
- Fixed and finalized note on the intent of the data-parallel library. (see [simd.general] p.5)
- Renamed `neutral_element` to `identity_element` and specified the requirement on its value in relation to `binary_op`. Removed the note explaining an implementation detail. (see [simd.reductions])

1.2 CHANGES FROM R6

Previous revision: [P0214R6].

- Fixed return type of `isinf`. ([simd.math])

✓ Changed namespace to `std::experimental::parallelism_v2`.

✓ Shortened intro sentence in [simd.syn].

✓ Added missing `const_where_expression` overload for `where(bool, const T&`. ([simd.syn], [simd.mask.where]) (The overload is present in my reference implementation.)

✓ Removed `const` from `const_where_expression` return type. ([simd.syn])

• Fixed incorrect `const where_expression` return types in [simd.mask.where].

✓ Removed `flags` namespace [simd.syn].

✓ all of your global constexpr variables should be inline

✓ Rename template parameters `A` to `Abis` and `As...` to `Abis...`.

✓ Stable names follow a hierarchy; use [simd.class] instead of [simd]; rename [simd_mask] to [simd.mask]

✓ reduce: Moved defaulted `BinaryOperation` argument to the end.

✓ reduce: Replaced `default_natural_element` with *implementation-defined*

✓ reduce: Explained what the default `natural_element` is supposed to do. ([simd.reductions])

✓ Consistently initialze tags.

✓ Put the "see belows" in italics instead of comments.

✓ Consistently pass by const-ref. (LWG: Why are the arguments by value? Be consistent. Resolved to consistently pass by const-ref. Rationale: if an implementation wants a non-inline, pass-by-value function it can inline the public const-ref function and call an internal function. If the public function uses by value parameters it may happen that the implied copy is not optimized away.)

✓ Simplify casts using aliases or default arguments.

✓ Split second `reduce` overload in six functions, one without default `neutral_element` and five explicitly overloaded for `plus<>`, `multiplies<>`, `bit_and<>`, `bit_or`, and `bit_xor`.

- ✓ Improved requires clause on `reduce` to mention return type and generalize ABI tag type requirement.
- ✓ Require integral `value_type` for bitwise reductions.
- ✓ Removed impossible condition from `simd_cast` remark.
- ✓ Fixed masked `hmin` and `hmax` specification to use "forallmaskedi".
- ✓ Forward `min`, `max`, `minmax`, and `clamp` to `std::min/max/minmax/clamp`.
- ✓ `scalar` *is* not an alias for `fixed_size<1>`.
- ✓ "an implementation *shall* support at least *all* ..." ([simd.abi])
- ✓ Replaced "exact-bool" arguments from *implementation-defined* to *see below*.
- ✓ Reveresed "if" to "unless" logic.
- ✓ Fixed `is_mask` to `is_simd_mask`.
- ✓ Fixed incorrect `simd_mask` exposition-only member name to `mask` ([simd.where-expr]).
- ✓ Fixed constraints on generator ctor to require generator be callable with all element indexes.
- ✓ Fixed wording to allow vectorized execution of the generator.
- ✓ Moved all wording about "target" or "architecture" into non-normative notes.
- ✓ Add `is_simd_flag_type` trait and use it for all loads and stores.
- ✓ Define and use the term *vectorizable type* to simplify the wording.
- ✓ Define "selected elements" in `where_expression` to use it instead of `data[i]` where `mask[i]` is `true`.
- ✓ [simd.whereexpr] reword what the members of `where_expression` mean and where they come from
- ✓ Replaced "floating-point and integral" with "arithmetic".
- ✓ Consistently use "element" instead of "component".
- ✓ Consistently place "and"/"or" and the end of bullet points instead of the front.

- Allow shifts on signed integers as was requested by LEWG.

✓ Added introduction in [simd.general].

✓ Added wording in [simd.general] to clarify that the application of operations/-operators on elements in `simd` and `simd_mask` are unsequenced with respect to each other.

✓ Fix Order: requires, effects, sync, post-cond, returns, throws, complexity, remarks, error-cond

✓ Removed repetitions of clause names.

✓ "Let X be foo" doesn't need a "Remarks" clause.

✓ Defined "vectorizable types" as a term for "arithmetic types other than bool".

- Removed immutable masked load. Requested in LWG review session because it's too clever and may block/hinder acceptance. LEWG queried/informed via reflector. Quick review of the issue at the start of Jacksonville meeting requested.

- Introduced exposition-only `nodeduce_t` to replace `remove_const_t` ([simd.mask.where]).

- Removed `noexcept` from `hmin` and `hmax` in the synopsis, to match the declarations below.

1.3 Changes from R5

Previous revision: [P0214R5].

- Renamed `memload` to `copy_from` and `memstore` to `copy_to`. ([simd.copy], [simd.mask.copy])

- Fixed `split` to never convert the `value_type`. ([simd.casts])

- Added missing `long double` overload of `ceil`. ([simd.math])

1.4 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.

1.5 Changes from R3

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` ([simd.mask.comparison]). This resolves an inconsistency with all the other binary operators.
- Editorial: Improve `reference` member specification ([simd.overview]).
- Require `swap(v[0], v[1])` to be valid ([simd.overview]).
- Fixed inconsisteny of masked load constructor after move of `memload` to `where_expression` ([simd.whereexpr]).
- Editorial: Use Requires clause instead of Remarks to require the memory argument to loads and stores to be large enough ([simd.whereexpr], [simd.copy], [simd.mask.copy]).

- Add a note to special math functions that precondition violation is UB ([simd.math]).
- Bugfix: Binary operators involving two `simd::reference` objects must work ([simd.overview]).
- 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 ([simd.overview] p.2, [simd.mask.overview] p.2, [simd.traits] p.7).
- 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 ([simd.copy]).
- Discuss `simd_mask` and `bitset` reduction interface differences.
- Relax requirements on return type of generator function for the generator constructor ([simd.ctor]).
- 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 ([simd.abi]).
- Remove most of the text about vendor-extensions for ABI tag types, since it's QoI ([simd.abi]).
- Clarify the differences and intent of `compatible<T>` vs. `native<T>` ([simd.abi]).
- Move definition of `where_expression` out of the synopsis ([simd.whereexpr]).
- Editorial: Improve `is_simd` and `is_mask` wording ([simd.traits]).
- Make *ABI tag* a consistent term and add `is_abi_tag` trait ([simd.traits], [simd.abi]).
- Clarify that `simd_abi::fixed_size<N>` must support all `N` matching all possible implementation-defined ABI tags ([simd.abi]).
- Clarify `abi_for_size` wording ([simd.traits]).
- 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 ([simd.traits]).
- Remove exposition-only `where_expression` constructor and make exposition-only data members private ([simd.whereexpr]).
- Editorial: use "shall not participate in overload resolution unless" consistently.
- Add a note about variability of `max_fixed_size` ([simd.abi]).
- 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 ([simd.reductions]).
- Editorial: clean up the use of "supported" and resolve contradictions resulting from incorrect use of conventions in the rest of the standard text ([simd.overview] p.2, [simd.mask.overview] p.2, [simd.traits]).
- Add Section 7 Feature Detection Macros.

1.6 CHANGES FROM R2

Previous revision: [P0214R2].

- Fixed return type of masked `reduce` ([simd.reductions]).
- Added binary `min`, `max`, `minmax`, and `clamp` ([simd.alg]).
- Moved member `min` and `max` to non-member `hmin` and `hmax`, which cannot easily be optimized from `reduce`, since no function object such as `std::plus` exists ([simd.reductions]).
- Fixed neutral element of masked `hmin`/`hmax` and drop UB ([simd.reductions]).
- Removed remaining reduction member functions in favor of non-member `reduce` (as requested by LEWG).
- Replaced `init` parameter of masked `reduce` with `neutral_element` ([simd.reductions]).
- Extend `where_expression` to support `const simd` objects ([simd.mask.where]).
- Fixed missing `explicit` keyword on `simd_mask(bool)` constructor ([simd.mask.ctor]).
- Made binary operators for `simd` and `simd_mask` friend functions of `simd` and `simd_mask`, simplifying the SFINAE requirements considerably ([simd.binary], [simd.mask.binary]).
- Restricted broadcasts to only allow non-narrowing conversions ([simd.ctor]).
- Restricted simd to simd conversions to only allow non-narrowing conversions with `fixed_size` ABI ([simd.ctor]).
- Added generator constructor (as discussed in LEWG in Issaquah) ([simd.ctor]).
- Renamed `copy_from` to `memload` and `copy_to` to `memstore`. ([simd.copy], [simd.mask.copy])
- Documented effect of `overaligned_tag<N>` as `Flags` parameter to load/store. ([simd.copy], [simd.mask.copy])
- Clarified cv requirements on `T` parameter of `simd` and `simd_mask`.
- Allowed all implicit `simd_mask` conversions with `fixed_size` ABI and equal size ([simd.mask.ctor]).
- Made increment and decrement of `where_expression` return `void`.

- Added `static_simd_cast` for simple casts ([simd.casts]).
- Clarified default constructor ([simd.overview], [simd.overview]).
- Clarified `simd` and `simd_mask` with invalid template parameters to be complete types with deleted constructors, destructor, and assignment ([simd.overview], [simd.overview]).
- Wrote a new subsection for a detailed description of `where_expression` ([simd.whereexpr]).
- Moved masked loads and stores from `simd` and `simd_mask` to `where_expression` ([simd.whereexpr]). This required two more overloads of `where` to support value objects of type `simd_mask` ([simd.mask.where]).
- Removed `where_expression::operator!` ([simd.whereexpr]).
- Added aliases `native_simd`, `native_mask`, `fixed_size_simd`, `fixed_size_mask` ([simd.syn]).
- Removed `bool` overloads of mask reductions awaiting a better solution ([simd.mask.reductions]).
- Removed special math functions with `f` and `l` suffix and `l` and `ll` prefix ([simd.math]).
- Modified special math functions with mixed types to use `fixed_size` instead of `abi_for_size` ([simd.math]).
- Added simple ABI cast functions `to_fixed_size`, `to_native`, and `to_compatible` ([simd.casts]).

1.7 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` ([simd.abi]).
- Added `max_fixed_size` constant ([simd.abi] 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. ([simd.copy], [simd.mask.copy]) (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. ([simd.reductions]) This made it necessary to spell out the class `where_expression`.

1.8 changes from R0

Previous revision: [P0214R0].

- Extended the `simd_abi` tag types with a `fixed_size<N>` tag to handle arbitrarily sized vectors ([simd.abi]).
- Converted `memory_alignment` into a non-member trait ([simd.traits]).
- Extended implicit conversions to handle `simd_abi::fixed_size<N>` ([simd.ctor]).
- Extended binary operators to convert correctly with `simd_abi::fixed_size<N>` ([simd.binary]).
- Dropped the section on "`simd` logical operators". Added a note that the omission is deliberate.
- Added logical and bitwise operators to `simd_mask` ([simd.mask.binary]).
- Modified `simd_mask` compares to work better with implicit conversions ([simd.mask.comparison]).
- Modified `where` to support different Abi tags on the `simd_mask` and `simd` arguments ([simd.mask.where]).
- 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.

2 STRAW POLLS

2.1 SG1 AT CHICAGO 2013

Poll: Pursue SIMD/data parallel programming via types?

SF	F	N	A	SA
1	8	5	0	0

2.2 SG1 AT URBANA 2014

- Poll: SF = ABI via namespace, SA = ABI as template parameter

SF	F	N	A	SA
0	0	6	11	2

- Poll: Apply size promotion to vector operations? SF = `shortv` + `shortv` = `intv`

SF	F	N	A	SA
1	2	0	6	11

- Poll: Apply “sign promotion” to vector operations? SF = `ushortv` + `shortv` = `ushortv`; SA = no mixed signed/unsigned arithmetic

SF	F	N	A	SA
1	5	5	7	2

2.3 SG1 AT LENEXA 2015

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

SF	F	N	A	SA
10	6	2	0	0

2.4 SG1 at Jax 2016

- Poll: Should subscript operator return an lvalue reference?

SF	F	N	A	SA
0	6	10	2	1

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

SF	F	N	A	SA
1	7	10	0	0

- Poll: Specify simd width using ABI tag, with a special template tag for fixed size.

SF	F	N	A	SA
3	7	0	0	1

- Poll: Specify simd width using <T, N, abi>, where abi is not specified by the user.

SF	F	N	A	SA
1	2	5	2	1

2.5 SG1 at Oulu 2016

- Poll: Keep `native_handle` in the wording (dropping the ampersand in the return type)?

SF	F	N	A	SA
0	6	3	3	0

- Poll: Should the interface provide a way to specify a number for over-alignment?

SF	F	N	A	SA
2	6	5	0	0

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

SF	F	N	A	SA
0	0	7	4	1

2.6 LEWG at Issaquah 2016

- Poll: Unary minus on unsigned simd should be ill formed

SF	F	N	A	SA
0	5	6	0	3

- 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 `where` and (`simd_mask` and `simd`) reductions with `bool`/builtin types in place of `simd_mask`/`simd`?

SF	F	N	A	SA
1	5	6	1	0

- Poll: Should there be a named "widen" function that widens the element type `T` from (e.g.) `int` to `long`, but rejects `int` to `short`? The number of elements is not changed.
  → unanimous consent
- Poll: Should there be a named "concat" functions that concatenates several `simd`s with the same element type, but potentially different length?
  The latter two operations are currently lumped together in `simd_cast`.
  The widen / concat questions, I think, comes down to whether we want the current `simd_cast` or not.
  → unanimous consent
- 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 `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

2.8 LEWG at Toronto 2017

- Poll: Use Option 3 (support both element type and `simd` type as cast template argument, instead of just either one of them) for `static_simd_cast` and `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)?

SF	F	N	A	SA
1	4	1	5	4

  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)?

SF	F	N	A	SA
0	1	2	5	7

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

SF	F	N	A	SA
4	6	3	1	0

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

SF	F	N	A	SA
5	1	3	2	3

  No consensus for change, revisit after TS feedback.

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

`load`	`memload`	`load_from`	`copy_from`
4	2	6	7

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

SF	F	N	A	SA
2	3	3	0	3

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

- Poll: Forward to LWG for TS

SF	F	N	A	SA
11	2	0	0	0

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 $\mathcal{W}_\mathrm{T}$ successive scalar values starting from a given address into a SIMD register.
- Store instructions: store from a SIMD register to $\mathcal{W}_\mathrm{T}$ 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 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 EXAMPLES

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.

```
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], vector_aligned);
    where(v > 100.f, v) = 100.f + (v - 100.f) * 0.1f;
    v.copy_to(&data[i], 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 targetting inclusion into the Parallelism TS 2. It defines a basic set of data-parallel types and operations.

(5.1) 8 Data-Parallel Types [simd.types]

(5.1.1) 8.1 General [simd.general]

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

2 Throughout this Clause, the term *data-parallel type* refers to all *supported* [simd.overview] instantiations of the `simd` and `simd_mask` class templates. A *data-parallel object* is an object of *data-parallel type*.

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

Note 1 this should allow GPU execution

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

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

(5.1.2) 8.2 Header `<experimental/simd>` synopsis [simd.syn]

```
namespace std::experimental {
  inline namespace parallelism_v2 {
    namespace simd_abi {
      struct scalar {};
      template <int N> struct fixed_size {};
      template <typename T> inline constexpr int max_fixed_size = implementation-defined;
      template <typename T> using compatible = implementation-defined;
      template <typename T> using native = implementation-defined;
    }

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

    // traits [simd.traits]
    template <class T> struct is_abi_tag;
    template <class T> inline constexpr bool is_abi_tag_v = is_abi_tag<T>::value;

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

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

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

    template <class T, size_t N> struct abi_for_size { using type = see below; };
    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>> struct simd_size;
    template <class T, class Abi = simd_abi::compatible<T>>
    inline constexpr size_t simd_size_v = simd_size<T, Abi>::value;
```

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

// class template simd [simd.class]
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_mask [simd.mask.class]
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>>;
template <class T, int N> using fixed_size_simd_mask = simd_mask<T, simd_abi::fixed_size<N>>;

// casts [simd.casts]
template <class T, class U, class Abi> see below simd_cast(const simd<U, Abi>&);
template <class T, class U, class Abi> see below static_simd_cast(const simd<U, Abi>&);

template <class T, class Abi>
fixed_size_simd<T, simd_size_v<T, Abi>> to_fixed_size(const simd<T, Abi>&) noexcept;
template <class T, class Abi>
fixed_size_simd_mask<T, simd_size_v<T, Abi>> to_fixed_size(const simd_mask<T, Abi>&) noexcept;
template <class T, size_t N> native_simd<T> to_native(const fixed_size_simd<T, N>&) noexcept;
template <class T, size_t N> native_simd_mask<T> to_native(const fixed_size_simd_mask<T, N>&) noexcept;
template <class T, size_t N> simd<T> to_compatible(const fixed_size_simd<T, N>&) noexcept;
template <class T, size_t N> simd_mask<T> to_compatible(const fixed_size_simd_mask<T, N>&) noexcept;

template <size_t... Sizes, class T, class Abi>
tuple<simd<T, abi_for_size_t<T, Sizes>>...> split(const simd<T, Abi>&);
template <size_t... Sizes, class T, class Abi>
tuple<simd_mask<T, abi_for_size_t<T, Sizes>>...> split(const simd_mask<T, Abi>&);
template <class V, class Abi>
array<V, simd_size_v<typename V::value_type, Abi> / V::size()> split(
    const simd<typename V::value_type, Abi>&);
template <class V, class Abi>
array<V, simd_size_v<typename V::value_type, Abi> / V::size()> split(
    const simd_mask<typename V::value_type, Abi>&);

template <class T, class... Abis>
simd<T, abi_for_size_t<T, (simd_size_v<T, Abis> + ...)>> concat(const simd<T, Abis>&...);
template <class T, class... Abis>
simd_mask<T, abi_for_size_t<T, (simd_size_v<T, Abis> + ...)>> concat(const simd_mask<T, Abis>&...);

// reductions [simd.mask.reductions]
template <class T, class Abi> bool  all_of(const simd_mask<T, Abi>&) noexcept;
template <class T, class Abi> bool  any_of(const simd_mask<T, Abi>&) noexcept;
template <class T, class Abi> bool none_of(const simd_mask<T, Abi>&) noexcept;
template <class T, class Abi> bool some_of(const simd_mask<T, Abi>&) noexcept;
template <class T, class Abi> int popcount(const simd_mask<T, Abi>&) noexcept;
template <class T, class Abi> int find_first_set(const simd_mask<T, Abi>&);
template <class T, class Abi> int find_last_set(const simd_mask<T, Abi>&);

bool  all_of(see below) noexcept;
bool  any_of(see below) noexcept;
bool none_of(see below) noexcept;
bool some_of(see below) noexcept;
int popcount(see below) noexcept;
int find_first_set(see below) noexcept;
int find_last_set(see below) noexcept;

// masked assignment [simd.whereexpr]
template <class M, class T> class const_where_expression;
template <class M, class T> class where_expression;

// masked assignment [simd.mask.where]
template <class T> struct nodeduce { using type = T; };             // exposition only
template <class T> using nodeduce_t = typename nodeduce<T>::type;   // exposition only

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

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

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

// reductions [simd.reductions]
template <class T, class Abi, class BinaryOperation = std::plus<>>
T reduce(const simd<T, Abi>&, BinaryOperation = BinaryOperation());
template <class M, class V, class BinaryOperation>
typename V::value_type reduce(const const_where_expression<M, V>& x,
                              typename V::value_type identity_element, BinaryOperation binary_op);
template <class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, plus<> binary_op = plus<>());
template <class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, multiplies<> binary_op);
template <class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, bit_and<> binary_op);
template <class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, bit_or<> binary_op);
template <class M, class V>
typename V::value_type reduce(const const_where_expression<M, V>& x, bit_xor<> binary_op);

template <class T, class Abi> T hmin(const simd<T, Abi>&);
template <class M, class V> typename V::value_type hmin(const const_where_expression<M, V>&);
template <class T, class Abi> T hmax(const simd<T, Abi>&);
template <class M, class V> typename V::value_type hmax(const const_where_expression<M, V>&);

// algorithms [simd.alg]
template <class T, class Abi> simd<T, Abi> min(const simd<T, Abi>&, const simd<T, Abi>&) noexcept;
template <class T, class Abi> simd<T, Abi> max(const simd<T, Abi>&, const simd<T, Abi>&) noexcept;
template <class T, class Abi>
std::pair<simd<T, Abi>, simd<T, Abi>> minmax(const simd<T, Abi>&, const simd<T, Abi>&) noexcept;
template <class T, class Abi>
simd<T, Abi> clamp(const simd<T, Abi>& v, const simd<T, Abi>& lo, const simd<T, Abi>& hi);
```

```
  }
}
```

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

(5.1.2.1) 8.2.1 `simd` ABI tags [simd.abi]

```
namespace simd_abi {
  struct scalar {};
  template <int N> struct fixed_size {};
  template <typename T> inline constexpr int max_fixed_size = implementation-defined;
  template <typename T> using compatible = implementation-defined;
  template <typename T> using native = implementation-defined;
}
```

1 An *ABI tag* is a type in the `simd_abi` namespace that indicates a choice of size and binary representation for objects of data-parallel type. [*Note:* The intent is for the size and binary representation to depend on the target architecture. — *end note*] The ABI tag, together with a given element type implies a number of elements. ABI tag types are used as the second template argument to `simd` and `simd_mask`. [*Note:* The ABI tag is orthogonal to selecting the machine instruction set. The selected machine instruction set limits the usable ABI tag types, though (see [simd.overview] p.2). The ABI tags enable users to safely pass objects of data-parallel type between translation unit boundaries (e.g. function calls or I/O). — *end note*]

2 Use of the `scalar` tag type requires data-parallel types to store a single element (i.e., `simd<T, simd_abi::scalar>::size()` returns `1`). [*Note:* `scalar` is not an alias for `fixed_size<1>`. — *end note*]

3 Use of the `simd_abi::fixed_size<N>` tag type requires data-parallel types to store and manipulate `N` elements (i.e. `simd<T, simd_abi::fixed_size<N>>::size()` returns `N`). An implementation shall support at least all `N` $\in [1 \dots 32]$. Additionally, for every supported `simd<T, Abi>` (see [simd.overview] p.2), where `Abi` is an implementation-defined ABI tag, `N` $=$ `simd<T, Abi>::size()` shall 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*]

4 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*]

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

6 `compatible<T>` is an implementation-defined alias for an ABI tag. [*Note:* The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type `T` that ensures ABI compatibility between translation units on the target architecture. — *end note*]

[*Example:* Consider a target architecture supporting the 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*]

7 `native<T>` is an implementation-defined alias for an ABI tag. [*Note:* The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type `T` that is supported on the currently targeted system. For target architectures without ISA extensions, the `native<T>` and `compatible<T>` aliases will likely be the same. For target architectures with ISA extensions, compiler flags may influence the `native<T>` alias while `compatible<T>` will be the same independent of such flags. — *end note*]

[*Example:* Consider a target architecture 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*]

(5.1.2.2) 8.2.2 `simd` type traits [simd.traits]

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

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

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

2 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_simd_mask { see below };
```

3 The type `is_simd_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> struct is_simd_flag_type { see below };
```

4 The type `is_simd_flag_type<T>` is a `UnaryTypeTrait` with a `BaseCharacteristic` of `true_type` if `T` is one of

- `element_aligned_tag`, or
- `vector_aligned_tag`, or
- `overaligned_tag<N>` with arbitrary value `N`

and `false_type` otherwise.

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

5 The member `type` shall be omitted unless

- `T` is a cv-unqualified type, and
- `T` is a vectorizable type, and
- `simd_abi::fixed_size<N>` is supported (see [simd.abi] p.3).

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

- `simd_size_v<T, type> == N`, and
- `simd<T, type>` is default constructible (see [simd.overview] p.2),

`simd_abi::scalar` takes precedence over `simd_abi::fixed_size<1>`. The precedence of implementation-defined ABI tags over `simd_abi::fixed_size<N>` is implementation-defined. [*Note:* It is expected that implementation-defined ABI tags can produce better optimizations and thus take precedence over `simd_abi::fixed_size<N>`. — *end note*]

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

7 `simd_size<T, Abi>` shall have no member `value` unless

- `T` is a cv-unqualified type, and
- `T` is a vectorizable type, and
- `is_abi_tag_v<Abi>` is `true`.

[*Note:* The rules are different from [simd.overview] p.2 — *end note*]

8 Otherwise, the type `simd_size<T, Abi>` is a `BinaryTypeTrait` with a `BaseCharacteristic` of `integral_constant<size_t, N>` with `N` equal to the number of elements in a `simd<T, Abi>` object. [*Note:* If `simd<T, Abi>` is not supported for the currently targeted system, `simd_size<T, Abi>::value` produces the value `simd<T, Abi>::size()` would return if it were supported. — *end note*]

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

9 `memory_alignment<T, U>` shall have no member `value` if either

- `T` is cv-qualified, or
- `U` is cv-qualified, or
- `!is_simd_v<T> && !is_simd_mask_v<T>`, or
- `is_simd_v<T>` and `U` is not an arithmetic type or `U` is `bool`, or
- `is_simd_mask_v<T>` and `U` is not `bool`.

10 Otherwise, the type `memory_alignment<T, U>` is a `BinaryTypeTrait` with a `BaseCharacteristic` of `integral_constant<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 [simd.copy] and [simd.mask.copy]). — *end note*]

(5.1.2.3) 8.2.3 Class templates `const_where_expression` and `where_expression` [simd.whereexpr]

```
namespace std::experimental {
  inline namespace parallelism_v2 {
    template <class M, class T> class const_where_expression {
      const M& mask;  // exposition only
      T& data;        // exposition only
    public:
      const_where_expression(const const_where_expression&) = delete;
      const_where_expression& operator=(const const_where_expression&) = delete;

      remove_const_t<T> operator-() const &&;

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

    template <class M, class T>
    class where_expression : public const_where_expression<M, T> {
    public:
      where_expression(const where_expression&) = delete;
      where_expression& operator=(const where_expression&) = delete;

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

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

1 The following refers to an object of type `M` as exposition-only data member `mask` and to a reference to `T` as exposition-only data member `data`.

Note 2 i.e. subclauses [simd.whereexpr], [simd.reductions], and [simd.mask.where]

2 In the following, if `M` is `bool`, `data[0]` is used interchangeably for `data`, `mask[0]` is used interchangeably for `mask`, and `M::size()` is used interchangeably for `1`.

Note 3 i.e. subclause [simd.whereexpr]

3 The class templates `const_where_expression` and `where_expression` abstract the notion of *selected elements* of a given object of arithmetic or data-parallel type. Selected elements signifies the elements `data[i]` for all $\mathtt{i} \in \{j \in \mathbb{N}_0 | j < \mathtt{M::size()} \bigwedge \mathtt{mask}[j]\}$.

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

5 If `M` is `bool`, `T` shall be a non-volatile arithmetic type. Otherwise, `T` shall either be `M`, `const M`, `typename M::simd_type`, or `const typename M::simd_type`.

6 In the following, the type `value_type` is an alias for `remove_const_t<T>` if `M` is `bool`, or an alias for `typename T::value_type` if `is_simd_mask_v<M>` is `true`.

Note 4 i.e. subclause [simd.whereexpr]

7 [*Note:* The `where` functions [simd.mask.where] initialize `mask` with the first argument to `where` and `data` with the second argument to `where`. — *end note*]

```
remove_const_t<T> operator-() const &&;
```

8 *Returns:* A copy of `data` with unary minus applied to all selected elements.

9 *Throws:* Nothing.

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

10 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<remove_const_t<T>, U>`. If the template parameter `Flags` is `overaligned_tag<N>`, the pointer value shall represent an address aligned to `N`. If `M` is not `bool`, the largest i $\in$ `[0, M::size())` where `mask[i]` is `true` is less than the number of values pointed to by `mem`.

11 *Effects:* Copies the selected elements as if `mem[i] = static_cast<U>(data[i])` for all `i` $\in \{j \in \mathbb{N}_0 | j <$ `M::size()` $\bigwedge$ `mask[`j`]`$\}$.

12 *Throws:* Nothing.

13 *Remarks:* If `remove_const_t<T>` is `bool` or `is_simd_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 a vectorizable type and `is_simd_flag_type_v<Flags>` is `true`.

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

14 *Effects:* Overwrites the selected elements with the corresponding elements from the application of the indicated operator on `data` and `forward<U>(x)`.

15 *Remarks:* Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `T`. It is unspecified whether the binary operator, implied by the compound assignment operator, is executed on all elements or only on the selected elements.

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

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

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

```
template <class U, class Flags> void copy_from(const U *mem, Flags);
```

18 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<T, U>`. If the template parameter `Flags` is `overaligned_tag<N>`, the pointer value shall represent an address aligned to `N`. If `M` is not `bool`, the largest i $\in$ `[0, M::size())` where `mask[i]` is `true` is less than the number of values pointed to by `mem`.

19 *Effects:* Overwrites the selected elements with the corresponding values `static_cast<value_type>(mem[i])` for all `i` $\in \{j \in \mathbb{N}_0 | j <$ `M::size()` $\bigwedge$ `mask[`j`]`$\}$.

20 *Remarks:* If `T` is `bool` or `is_simd_mask_v<T>`, this function shall not participate in overload resolution unless `U` is `bool`.

(5.1.3) 8.3 Class template `simd` [simd.class]

(5.1.3.1) 8.3.1 Class template `simd` overview [simd.overview]

```
namespace std::experimental {
  inline namespace parallelism_v2 {
    template <class T, class Abi> class simd {
    public:
      using value_type = T;
      using reference = see below;
      using mask_type = simd_mask<T, Abi>;
      using abi_type = Abi;

      static constexpr size_t size() noexcept;

      simd() = default;

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

      // implicit broadcast constructor (see below for constraints)
      template <class U> simd(U&& value);

      // generator constructor (see below for constraints)
      template <class G> explicit simd(G&& gen);

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

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

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

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

      // unary operators [simd.unary]
      simd& operator++();
      simd operator++(int);
      simd& operator--();
      simd operator--(int);
      mask_type operator!() const;
      simd operator~() const;   // see below
      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&, 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+= (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&);
      friend simd& operator<<=(simd&, const simd&);
      friend simd& operator>>=(simd&, const simd&);
      friend simd& operator<<=(simd&, int);
      friend simd& operator>>=(simd&, int);

      // compares [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&);
    };
  }
}
```

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

2 Each instantiation of `simd` 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 vectorizable type.
- The second template argument `Abi` is an ABI tag.
- 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 [simd.abi] 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 other than `long double` and no support for the `gpu_y` ABI tag, then:

- `simd<T, simd_abi::gpu_y>` is not supported for any `T` and 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*]

3 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*]

4 The member type `reference` is an unspecified type acting as a reference to an element of a data-parallel 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 for `value_type` is usable.
- Application of an assignment, compound assignment, increment, or decrement operator on a `reference` object is applied to the referenced element.
- Objects of type `reference` are implicitly convertible to `value_type` returning the value of the referenced element.
- 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_t 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_cast`ing from and to implementation-defined types. This would add one or more of the following declarations to class `simd`:

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

— *end note*]

(5.1.3.2) 8.3.2 `simd` constructors [simd.ctor]

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

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

2 *Throws:* Any exception thrown while converting the argument to `value_type`.

3 *Remarks:* This constructor shall not participate in overload resolution unless:

- `U` is a vectorizable type 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

Q 5 Mention forwarding on conversion to `value_type`?

Q 6 `U` is cv- and ref-qualified, is the wording below OK?

- U is int, or
- U is unsigned int and value_type is an unsigned integral type.

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

4 *Effects:* Constructs an object where the i-th element equals static_cast<T>(x[i]) for all i ∈ [0, size()).

5 *Remarks:* This constructor shall not participate in overload resolution unless

- abi_type is 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 [conv.rank] of value_type is greater than the integer conversion rank of U.

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

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

7 *Remarks:* This constructor shall not participate in overload resolution unless simd(gen(integral_constant<size_t, i>())) is well-formed for all i ∈ [0, size()). The calls to gen are unsequenced with respect to each other. [*Note:* This allows vectorized execution of the gen calls. — *end note*]

```
template <class U, class Flags> simd(const U *mem, Flags);
```

8 *Requires:* If the template parameter Flags is vector_aligned_tag, the pointer value shall represent an address aligned to memory_alignment_v<simd, U>. If the template parameter Flags is overaligned_tag<N>, the pointer value shall represent an address aligned to N. size() is less than or equal to the number of values pointed to by mem.

9 *Effects:* Constructs an object where the i-th element is initialized to static_cast<T>(mem[i]) for all i ∈ [0, size()).

10 *Remarks:* This constructor shall not participate in overload resolution unless U is a vectorizable type and is_simd_flag_type_v<Flags> is true.

(5.1.3.3) 8.3.3 simd copy functions [simd.copy]

```
template <class U, class Flags> void copy_from(const U *mem, Flags);
```

1 *Requires:* If the template parameter Flags is vector_aligned_tag, the pointer value shall represent an address aligned to memory_alignment_v<simd, U>. If the template parameter Flags is overaligned_tag<N>, the pointer value shall represent an address aligned to N. size() is less than or equal to the number of values pointed to by mem.

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

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

```
template <class U, class Flags> void copy_to(U *mem, Flags);
```

4 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<simd, U>`. If the template parameter `Flags` is `overaligned_tag<N>`, the pointer value shall represent an address aligned to `N`. `size()` is less than or equal to the number of values pointed to by `mem`.

5 *Effects:* Copies all `simd` elements as if `mem[i] = static_cast<U>(operator[](i))` for all `i` ∈ `[0, size())`.

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

(5.1.3.4) 8.3.4 `simd` subscript operators [simd.subscr]

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

1 *Requires:* `i < size()`

2 *Returns:* A temporary object of type `reference` (see [simd.overview] p.4) that references the i-th element.

3 *Throws:* Nothing.

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

4 *Requires:* `i < size()`

5 *Returns:* A copy of the i-th element.

6 *Throws:* Nothing.

(5.1.3.5) 8.3.5 `simd` unary operators [simd.unary]

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

1 *Effects:* Is a unary element-wise operation that applies `operator++`.

2 *Returns:* `*this`

3 *Throws:* Nothing.

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

4 *Effects:* Is a unary element-wise operation that applies `operator++`.

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

6 *Throws:* Nothing.

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

7 *Effects:* Is a unary element-wise operation that applies `operator--`.

8 *Returns:* `*this`

9 *Throws:* Nothing.

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

10 *Effects:* Is a unary element-wise operation that applies `operator--`.

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

12 *Throws:* Nothing.

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

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

14 *Throws:* Nothing.

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

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

16 *Throws:* Nothing.

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

Note 7 "shall either not be declared or not participate in overload resolution"? impl. via base class or concepts

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

18 *Returns:* `*this`

19 *Throws:* Nothing.

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

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

21 *Throws:* Nothing.

(5.1.4) 8.4 `simd` non-member operations [simd.nonmembers]

(5.1.4.1) 8.4.1 `simd` binary operators [simd.binary]

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

1 *Returns:* A `simd` object initialized with the results of the element-wise application of the indicated operator.

2 *Throws:* Nothing.

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

Note 8 "shall either not be declared or not participate in overload resolution"? impl. via base class or concepts

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

4 *Returns:* A `simd` object where the i-th element is initialized to the result of applying the indicated operator to `v[i]` and `n` for all `i` ∈ `[0, size())`.

5 *Throws:* Nothing.

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

Note 9 "shall either not be declared or not participate in overload resolution"? impl. via base class or concepts

(5.1.4.2) 8.4.2 `simd` compound assignment [simd.cassign]

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

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

2 *Returns:* `lhs`.

3 *Throws:* Nothing.

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

Note 10 "shall either not be declared or not participate in overload resolution"? impl. via base class or concepts

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

5 *Effects:* Performs the indicated shift by `n` operation on the i-th element of `v` for all `i` ∈ `[0, size())`.

6 *Returns:* `v`.

7 *Throws:* Nothing.

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

Note 11 "shall either not be declared or not participate in overload resolution"? impl. via base class or concepts

(5.1.4.3) 8.4.3 `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&);
```

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

2 *Throws:* Nothing.

(5.1.4.4) 8.4.4 `simd` reductions [simd.reductions]

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

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

2 *Returns:* *`GENERALIZED_SUM`*`(binary_op, x.data[i], …)` for all `i` ∈ `[0, size())`.

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

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

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

5 *Returns:* If `none_of(x.mask)`, returns `identity_element`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[i], …)` for all `i` ∈ $\{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

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

6 *Returns:* If `none_of(x.mask)`, returns 0. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[i], …)` for all `i` ∈ $\{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

7 *Throws:* Nothing.

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

8 *Returns:* If `none_of(x.mask)`, returns 1. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[i], …)` for all `i` ∈ $\{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

9 *Throws:* Nothing.

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

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

11 *Returns:* If `none_of(x.mask)`, returns (V::value_type()). Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[i], …)` for all `i` ∈ $\{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

12 *Throws:* Nothing.

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

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

14 *Returns:* If `none_of(x.mask)`, returns 0. Otherwise, returns *GENERALIZED_SUM*`(binary_op, x.data[i], ...)` for all i $\in \{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

15 *Throws:* Nothing.

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

16 *Returns:* The value of an element `x[j]` for which `x[j] <= x[i]` for all i $\in$ `[0, size())`.

17 *Throws:* Nothing.

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

18 *Returns:* If `none_of(x.mask)`, the return value is `numeric_limits<V::value_type>::max()`. Otherwise, returns the value of an element `x.data[j]` for which `x.mask[j] == true` and `x.data[j] <= x.data[i]` for all i $\in \{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

19 *Throws:* Nothing.

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

20 *Returns:* The value of an element `x[j]` for which `x[j] >= x[i]` for all i $\in$ `[0, size())`.

21 *Throws:* Nothing.

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

22 *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 $\in \{j \in \mathbb{N}_0 | j <$ `size()` $\bigwedge$ `x.mask[`j`]`$\}$.

23 *Throws:* Nothing.

(5.1.4.5) 8.4.5 `simd` casts [simd.casts]

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

1 Let `To` identify `T::value_type` if `is_simd_v<T>` is `true`, or `T` otherwise.

2 *Returns:* A `simd` object with the i-th element initialized to `static_cast<To>(x[i])`.

3 *Throws:* Nothing.

4 *Remarks:* The function shall not participate in overload resolution unless

- every possible value of type `U` can be represented with type `To`, and
- either `is_simd_v<T>` is `false`, or `T::size() == simd<U, Abi>::size()` is `true`.

If `is_simd_v<T>` is `true`, the return type is `T`. Otherwise, if `U` is `T`, the return type is `simd<T, Abi>`. Otherwise, the return type is `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`.

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

5 Let `To` identify `T::value_type` if `is_simd_v<T>` or `T` otherwise.

6 *Returns:* A `simd` object with the i-th element initialized to `static_cast<To>(x[i])`.

7 *Throws:* Nothing.

8 *Remarks:* The function shall not participate in overload resolution unless either `is_simd_v<T>` is `false` or `T::size() == simd<U, Abi>::size()` is `true`. If `is_simd_v<T>` is `true`, the return type is `T`. Otherwise, if either `U` is `T` or `U` and `T` are integral types that only differ in signedness, the return type is `simd<T, Abi>`. Otherwise, the return type is `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`.

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

9 *Returns:* An object of the return type with the i-th element initialized to `x[i]`.

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

10 *Returns:* An object of the return type with the i-th element initialized to `x[i]`.

11 *Remarks:* These functions shall not participate in overload resolution unless `simd_size_v<T, simd_abi::native<T>> == N` is `true`.

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

12 *Returns:* An object of the return type with the i-th element initialized to `x[i]`.

13 *Remarks:* These functions shall not participate in overload resolution unless `simd_size_v<T, simd_abi::compatible<T>> == N` is `true`.

```
template <size_t... Sizes, class T, class Abi>
tuple<simd<T, abi_for_size_t<T, Sizes>>...> split(const simd<T, Abi>& x);
template <size_t... Sizes, class T, class Abi>
tuple<simd_mask<T, abi_for_size_t<T, Sizes>>...> split(const simd_mask<T, Abi>& x);
```

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

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

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

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

17 *Remarks:* These functions shall not participate in overload resolution unless

- `is_simd_v<V>` is `true` for the first signature / `is_mask_v<V>` is `true` for the second signature, and
- `simd_size_v<typename V::value_type, Abi>` is an integral multiple of `V::size()`.

```
template <class T, class... Abis>
simd<T, abi_for_size_t<T, (simd_size_v<T, Abis> + ...)>> concat(const simd<T, Abis>&... xs);
template <class T, class... Abis>
simd_mask<T, abi_for_size_t<T, (simd_size_v<T, Abis> + ...)>> concat(const simd_mask<T, Abis>&... xs);
```

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

(5.1.4.6) 8.4.6 `simd` algorithms [simd.alg]

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

1 *Returns:* The result of binary element-wise application of `std::min(a[i], b[i])` for all `i` ∈ `[0, size())`.

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

2 *Returns:* The result of binary element-wise application of `std::max(a[i], b[i])` for all `i` ∈ `[0, size())`.

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

3 *Returns:* A pair initialized with

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

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

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

5 *Returns:* The result of element-wise application of `std::clamp(a[i], lo[i], hi[i])` for all `i` ∈ `[0, size())`.

Note 12 do we really need a definition of *ternary element-wise*?

(5.1.4.7) 8.4.7 simd math library [simd.math]

1 For each set of overloaded functions within `<cmath>`, there shall be additional overloads sufficient to ensure that if any argument corresponding to a `double` parameter has type `simd<T, Abi>`, where `is_floating_point_v<T>` is `true`, then:

- All arguments corresponding to `double` parameters shall be convertible to `simd<T, Abi>`.
- All arguments corresponding to `double*` parameters shall be of type `simd<T, Abi>*`.
- All arguments corresponding to parameters of integral type `U` shall be convertible to `fixed_size_simd<U, simd_size_v<T, Abi>>`.
- All arguments corresponding to `U*`, where `U` is integral, shall be of type `fixed_size_simd<U, simd_size_v<T, Abi>>*`.
- If the corresponding return type is `double`, the return type of the additional overload is `simd<T, Abi>`. Otherwise, if the corresponding return type is `bool`, the return type of the additional overload is `simd_mask<T, Abi>`. Otherwise, the return type is `fixed_size_simd<R, simd_size_v<T, Abi>>`, with `R` denoting the corresponding return type.

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

2 Each function overload produced by the above rules concurrently applies the indicated mathematical function element-wise. The results per element are not required to be bitwise 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*]

Note 13 Neither the C nor the C++ standard say anything about expected error/precision. It seems returning 0 from all functions is a conforming implementation — just bad QoI.

3 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.

(5.1.5) 8.5 Class template simd_mask [simd.mask.class]

(5.1.5.1) 8.5.1 Class template simd_mask overview [simd.mask.overview]

Note 14 Make this normative? And prefer implementation-defined or unspecified?

```
namespace std::experimental {
  inline namespace parallelism_v2 {
    template <class T, class Abi> class simd_mask {
    public:
      using value_type = bool;
      using reference = see below;
      using simd_type = simd<T, Abi>;
      using abi_type = Abi;

      static constexpr size_t size() noexcept;

      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 [simd.mask.copy]
      template <class Flags> void copy_from(const value_type* mem, Flags);
      template <class Flags> void copy_to(value_type* mem, Flags) const;

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

      // unary operators [simd.mask.unary]
      simd_mask operator!() const noexcept;

      // simd_mask binary operators [simd.mask.binary]
      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;

      // simd_mask compound assignment [simd.mask.cassign]
      friend simd_mask& operator&=(simd_mask&, const simd_mask&) noexcept;
      friend simd_mask& operator|=(simd_mask&, const simd_mask&) noexcept;
      friend simd_mask& operator^=(simd_mask&, const simd_mask&) noexcept;

      // simd_mask compares [simd.mask.comparison]
      friend simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
      friend simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;
    };
  }
}
```

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

2 Each instantiation of `simd_mask` 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 vectorizable type.
- The second template argument `Abi` is an ABI tag.
- 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 [simd.abi] 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*]

3 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*]

4 The member type `reference` follows the specification of the `simd<T, Abi>::reference` member type.

```
static constexpr size_t size() noexcept;
```

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

6 [*Note:* Implementations are encouraged to enable `static_cast`ing from and to implementation-defined types. This would add one or more of the following declarations to class `simd_mask`:

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

— *end note*]

(5.1.5.2) 8.5.2 `simd_mask` constructors [simd.mask.ctor]

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

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

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

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

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

```
template <class Flags> simd_mask(const value_type *mem, Flags);
```

4 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<simd_mask>`. If the template parameter `Flags` is `overaligned_tag<N>`, the pointer value shall represent an address aligned to `N`. `size()` is less than or equal to the number of values pointed to by `mem`.

5 *Effects:* Constructs an object where the i-th element is initialized to `mem[i]` for all `i` ∈ `[0, size())`.

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

(5.1.5.3) 8.5.3 `simd_mask` copy functions [simd.mask.copy]

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

1 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<simd_mask>`. If the template parameter `Flags` is `overaligned_tag<N>`, the pointer value shall represent an address aligned to `N`. `size()` is less than or equal to the number of values pointed to by `mem`.

2 *Effects:* Replaces the elements of the `simd_mask` object such that the i-th element is replaced with `mem[i]` for all `i` ∈ `[0, size())`.

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

```
template <class Flags> void copy_to(value_type *mem, Flags);
```

4 *Requires:* If the template parameter `Flags` is `vector_aligned_tag`, the pointer value shall represent an address aligned to `memory_alignment_v<simd_mask>`. If the template parameter `Flags` is `over-aligned_tag<N>`, the pointer value shall represent an address aligned to `N`. `size()` is less than or equal to the number of values pointed to by `mem`.

5 *Effects:* Copies all `simd_mask` elements as if `mem[i] = operator[](i)` for all `i` ∈ `[0, size())`.

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

(5.1.5.4) 8.5.4 `simd_mask` subscript operators [simd.mask.subscr]

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

1 *Requires:* `i < size()`

2 *Returns:* A temporary object of type `reference` (see [simd.overview] p.4) that references the i-th element.

3 *Throws:* Nothing.

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

4 *Requires:* The value of `i` is less than `size()`.

5 *Returns:* A copy of the i-th element.

6 *Throws:* Nothing.

(5.1.5.5) 8.5.5 `simd_mask` unary operators [simd.mask.unary]

```
mask operator!() const noexcept;
```

1 *Returns:* The result of unary element-wise application of `operator!`.

2 *Throws:* Nothing.

(5.1.6) 8.6 `simd_mask` non-member operations [simd.mask.nonmembers]

(5.1.6.1) 8.6.1 `simd_mask` binary operators [simd.mask.binary]

```
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 element-wise application of the indicated operator.

2 *Throws:* Nothing.

(5.1.6.2) 8.6.2 `simd_mask` compound assignment [simd.mask.cassign]

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

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

2 *Returns:* `lhs`.

3 *Throws:* Nothing.

(5.1.6.3) 8.6.3 `simd_mask` compares [simd.mask.comparison]

```
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 element-wise application of the indicated operator.

(5.1.6.4) 8.6.4 `simd_mask` reductions [simd.mask.reductions]

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

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

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

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

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

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

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

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

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

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

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

6 *Requires:* `any_of(k)` returns `true`

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

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

8 *Requires:* `any_of(k)` returns `true`

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

```
bool  all_of(see below) noexcept;
bool  any_of(see below) noexcept;
bool none_of(see below) noexcept;
bool some_of(see below) noexcept;
int popcount(see below) noexcept;
int find_first_set(see below) noexcept;
int find_last_set(see below) noexcept;
```

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

11 *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.

(5.1.6.5) 8.6.5 Masked assigment [simd.mask.where]

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

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

1 *Returns:* A where expression object [simd.whereexpr] as if the exposition-only data members `mask` and `data` are initialized with `k` and `v` respectively.

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

2 *Remarks:* The functions shall not participate in overload resolution unless

- `T` is neither a `simd` nor a `simd_mask` instantiation, and
- the first argument is of type `bool`.

3 *Returns:* A where expression object [simd.whereexpr] as if the exposition-only data members `mask` and `data` are initialized with `k` and `v` respectively.

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 *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?

SF	F	N	A	SA
0	6	10	2	1

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

SF	F	N	A	SA
1	7	10	0	0

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 builtin 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`).

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?

6.7.1 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.

```
template <class T, size_t N = simd_size_v<T, simd_abi::compatible<T>>,
          class Abi = simd_abi::compatible<T>>
class simd;
```

Listing 1: Possible declaration of the class template parameters of a `simd` class with arbitrary width.

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

6.7.2 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:

```
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:

```
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:

```
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.

6.7.4 Progress

SG1 Guidance at JAX 2016:
Poll: Specify simd width using ABI tag, with a special template tag for fixed size.

SF	F	N	A	SA
3	7	0	0	1

Poll: Specify simd width using <T, N, abi>, where abi is not specified by the user.

SF	F	N	A	SA
1	2	5	2	1

At the Jacksonville meeting, SG1 decided to continue with the `simd<T, Abi>` class template, with the addition of a new Abi type that denotes a user-requested number of elements in the vector (`simd_abi::fixed_size<N>`). This has the following implications:

- There is only one class template with a common interface for *fundamental* and *arbitrary* (`fixed_size`) vector types.

- There are slight differences in the conversion semantics for `simd` types with the `fixed_size` Abi type. This may look like the `vector<bool>` mistake all over again. I'll argue below why I believe this is not the case.
- The *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 *fundamental* types are allowed to be passed via registers on function calls. `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 *fundamental* and `fixed_size` class template instances are the main motivation for having a distinction (cf. discussion above). The differences are chosen such that, in general, *fundamental* types are more restrictive and do not turn into `fixed_size` types on any operation that involves no `fixed_size` types. Operations of `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

```
1 using floatv = simd<float>;  // native ABI
2 using float_sized_abi = simd_abi::fixed_size<floatv::size()>;
3 using intv = simd<int, float_sized_abi>;
4
5 auto x = floatv() + intv();/*! \label{lstline:1} */
6 intv y = floatv() + intv();/*! \label{lstline:2} */
```

   Line **??** is well-formed: It states that N (= `floatv::size()`) additions shall be executed concurrently. The type of `x` is `simd<float>`, because it stores N elements and both types `intv` and `floatv` are implicitly convertible to `simd<float>`. Line **??** is also well-formed because implicit conversion from `simd<T, Abi>` to `simd<U, simd_abi::fixed_size<N>>` is allowed whenever `N == 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();/*! \label{lstline:3} */
intv y = floatv() + intv();/*! \label{lstline:4} */
```

Line **??** is well-formed: It states that N (= `intv::size()`) additions shall be executed concurrently. The type of `x` is `simd<simd<float>, int_sized_abi>` (i.e. `floatv`) and never `simd<float>`, because …

… the `Abi` types of `intv` and `floatv` are not equal.

… either `simd<float>::size() != N` or `intv` is not implicitly convertible to `simd<float>`.

… the last rule for *commonabi*`(V0, V1, T)` sets the `Abi` type to `int_sized_abi`.

Line **??** is also well-formed because implicit conversion from `simd<T, simd_abi::fixed_size<N>>` to `simd<U, Abi>` is allowed whenever `N == 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 standardized 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`.

1 Whether that's a reasonable fear is a different discussion.

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

SF	F	N	A	SA
0	6	3	3	0

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:

- 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.

2 As I realized too late.

Guidance from SG1 at Oulu 2016:

Poll: Should the interface provide a way to specify a number for over-alignment?

SF	F	N	A	SA
2	6	5	0	0

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

SF	F	N	A	SA
0	0	7	4	1

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 ([simd.abi] p.3: "for every supported `simd<T, A>` (see [simd.overview] p.2), where `A` is an implementation-defined ABI tag, `N` = `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. $N \leq$ `max_fixed_size` works; whether $N >$ `max_fixed_size` works is unspecified.

7 FEATURE DETECTION MACROS

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`.

A 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.

B BIBLIOGRAPHY

[1] Matthias Kretz. "Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types." Frankfurt (Main), Univ. PhD thesis. 2015. DOI: `10.13140/RG.2.1.2355.4323`. URL: `http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415`.

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

[N4185] Matthias Kretz. *N4185: SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. 2014. URL: `https://wg21.link/n4185`.

[N4395] Matthias Kretz. *N4395: SIMD Types: ABI Considerations*. ISO/IEC C++ Standards Committee Paper. 2015. URL: `https://wg21.link/n4395`.

[N4454] Matthias Kretz. *N4454: SIMD Types Example: Matrix Multiplication*. ISO/IEC C++ Standards Committee Paper. 2015. URL: `https://wg21.link/n4454`.

[P0214R0] Matthias Kretz. *P0214R0: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: `https://wg21.link/p0214r0`.

[P0214R1] Matthias Kretz. *P0214R1: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: `https://wg21.link/p0214r1`.

[P0214R2] Matthias Kretz. *P0214R2: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: `https://wg21.link/p0214r2`.

[P0214R3] Matthias Kretz. *P0214R3: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0214r3`.

[P0214R4] Matthias Kretz. *P0214R4: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0214r4`.

[P0214R5] Matthias Kretz. *P0214R5: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0214r5`.

[P0214R6] Matthias Kretz. *P0214R6: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0214r6`.

[P0214R7] Matthias Kretz. *P0214R7: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2017. URL: `https://wg21.link/p0214r7`.

[2] Bjarne Stroustrup. "An Overview of the C++ Programming Language." In: *Handbook of Object Technology*. Ed. by Saba Zamir. Boca Raton, Florida: CRC Press LLC, 1999. ISBN: 0849331358.