Document Number: P0214R1
Date: 2016-05-28
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: LEWG / SG1

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 $\mathcal{W}_{\mathrm{T}}$ 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 starting point. Once the foundation is settled on the API will be completed.

1 CHANGELOG

1.1 CHANGES FROM R0

Previous revision: [P0214R0].

- Extended the `datapar_abi` tag types with a `fixed_size<N>` tag to handle arbitrarily sized vectors (3.1.1.1).
- Converted `memory_alignment` into a non-member trait (3.1.1.2).
- Extended implicit conversions to handle `datapar_abi::fixed_size<N>` (3.1.2.2).
- Extended binary operators to convert correctly with `datapar_abi::fixed_size<N>` (3.1.3.3).
- Dropped the section on "`datapar` logical operators". Added a note that the omission is deliberate (3.1.3.5).
- Added logical and bitwise operators to `mask` (3.1.5.3).
- Modified `mask` compares to work better with implicit conversions (3.1.5.4).
- Modified `where` to support different Abi tags on the `mask` and `datapar` arguments (3.1.5.6).

- 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 `datapar::operator[]` to return a smart reference instead of an lvalue reference.
- Modified the wording of `mask::operator[]` to match the reference type returned from `datapar::operator[]`.
- Added non-trig/pow/exp/log math functions on `datapar`.
- Added discussion on defaulting load/store flags.
- Added sum, product, min, and max reductions for `datapar`.
- 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 4.10).
- Fixed missing flag objects.

2 INTRODUCTION

2.1 SIMD REGISTERS AND OPERATIONS

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

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

- Load instructions: load $\mathcal{W}_T$ successive scalar values starting from a given address into a SIMD register.

- Store instructions: store from a SIMD register to $\mathcal{W}_{\mathrm{T}}$ 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.

2.2 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.

2.3 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.)

3 WORDING

The following is a draft of possible wording that defines a basic set of data-parallel types and operations.

3.1 Data-Parallel Types [datapar.types]

3.1.1 Header `<datapar>` synopsis [datapar.syn]

```
namespace std {
  namespace experimental {
    namespace datapar_abi {
      struct scalar {};  // always present
      template <int N> struct fixed_size {};  // always present
      // implementation-defined tag types, e.g. sse, avx, avx512, neon, ...
      typedef implementation_defined compatible;  // always present
      typedef implementation_defined native;  // always present
    }

    namespace flags {
        struct unaligned_tag {};
        struct aligned_tag {};
```

```
    using load_default = unaligned_tag;
    using store_default = unaligned_tag;
    constexpr unaligned_tag unaligned{};
    constexpr aligned_tag aligned{};
}

// traits [datapar.traits]
template <class T> struct is_datapar;
template <class T> constexpr bool is_datapar_v = is_datapar<T>::value;

template <class T> struct is_mask;
template <class T> constexpr bool is_mask_v = is_mask<T>::value;

template <class T, size_t N> struct abi_for_size { typedef implementation_defined type; };
template <class T, size_t N> using abi_for_size_t = typename abi_for_size<T, N>::type;

template <class T, class Abi = datapar_abi::compatible>
struct datapar_size : public integral_constant<size_t, implementation_defined> {};
template <class T, class Abi = datapar_abi::compatible>
constexpr size_t datapar_size_v = datapar_size<T, Abi>::value;

template <class T, class U = typename T::value_type>
constexpr size_t memory_alignment = implementation_defined;

// class template datapar [datapar]
template <class T, class Abi = datapar_abi::compatible> class datapar;

// class template mask [mask]
template <class T, class Abi = datapar_abi::compatible> class mask;

// datapar load function [datapar.load]
template <class T = void, class U, class Flags = flags::load_default>
conditional_t<is_same_v<T, void>, datapar<U>, conditional_t<is_datapar_v<T>, T, datapar<T>>> load(
    const U *, Flags = Flags{});

// datapar store functions [datapar.store]
template <class T, class Abi, class U, class Flags = flags::store_default>
void store(const datapar<T, Abi> &, U *, Flags = Flags{});

template <class T0, class A0, class U, class T1, class A1, class Flags = flags::store_default>
void store(const datapar<T0, A0> &, U *, const mask<T1, A1> &, Flags = Flags{});

// compound assignment [datapar.cassign]
template <class T, class Abi, class U> datapar<T, Abi> &operator+= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator-= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator*= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator/= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator%= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator&= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator|= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator^= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator<<=(datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator>>=(datapar<T, Abi> &, const U &);

// binary operators [datapar.binary]
template <class L, class R> using datapar_return_type = ...;   // exposition only

template <class T, class Abi, class U>
```

```
datapar_return_type<datapar<T, Abi>, U> operator+ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(const U &, datapar<T, Abi>);

// compares [datapar.comparison]
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
```

```
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (const U &, datapar<T, Abi>);

// casts [datapar.casts]
template <class T, class U, class... Us>
conditional_t<(T::size() == (U::size() + Us::size()...)), T,
              array<T, (U::size() + Us::size()...) / T::size()>> datapar_cast(U, Us...);

// mask load function [mask.load]
template <class T, class Flags = flags::load_default> T load(const bool *, Flags = Flags{});

// mask store functions [mask.store]
template <class T, class Abi, class Flags = flags::load_default>
void store(const mask<T, Abi> &, bool *, Flags = Flags{});

template <class T0, class A0, class T1, class A1, class Flags = flags::load_default>
void store(const mask<T0, A0> &, bool *, const mask<T1, A1> &, Flags = Flags{});

// mask binary operators [mask.binary]
template <class T0, class A0, class T1, class A1> using mask_return_type = ...   // exposition only
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator&&(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator||(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator& (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator| (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator^ (const mask<T0, A0> &, const mask<T1, A1> &);

// mask compares [mask.comparison]
template <class T0, class A0, class T1, class A1>
bool operator==(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
bool operator!=(const mask<T0, A0> &, const mask<T1, A1> &);

// reductions [mask.reductions]
template <class T, class Abi> bool  all_of(mask<T, Abi>);
constexpr bool  all_of(bool);
template <class T, class Abi> bool  any_of(mask<T, Abi>);
constexpr bool  any_of(bool);
template <class T, class Abi> bool none_of(mask<T, Abi>);
constexpr bool none_of(bool);
template <class T, class Abi> bool some_of(mask<T, Abi>);
constexpr bool some_of(bool);
template <class T, class Abi> int popcount(mask<T, Abi>);
constexpr int popcount(bool);
template <class T, class Abi> int find_first_set(mask<T, Abi>);
constexpr int find_first_set(bool);
template <class T, class Abi> int find_last_set(mask<T, Abi>);
constexpr int find_last_set(bool);
```

```
    // masked assignment [mask.where]
    template <class T0, class A0, class T1, class A1>
    implementation_defined where(const mask<T0, A0> &, datapar<T1, A1> &);
    template <class T> implementation_defined where(bool, T &);
  }
}
```

1 The header <datapar> defines two class templates (`datapar`, and `mask`), several tag types, and a series of related function templates for concurrent manipulation of the values in `datapar` and `mask` objects.

3.1.1.1 `datapar` ABI tags [datapar.abi]

```
namespace datapar_abi {
  struct scalar {};
  template <int N> struct fixed_size {};
  // implementation-defined tag types, e.g. sse, avx, avx512, neon, ...
  typedef implementation_defined compatible;
  typedef implementation_defined native;
}
```

1 The ABI types are tag types to be used as the second template argument to `datapar` and `mask`.

2 The `scalar` tag is present in all implementations and forces `datapar` and `mask` to store a single component (i.e. `datapar<T, datapar_abi::scalar>::size()` returns `1`).

3 The `fixed_size` tag is present in all implementations. Use of `datapar_abi::fixed_size<N>` forces `datapar` and `mask` to store and manipulate `N` components (i.e. `datapar<T, datapar_abi::fixed_size<N>>::size()` returns `N`). An implementation must support at least any $N \in [1 \ldots 32]$. Additionally, an implementation must support any $N \in \{$`datapar<U>::size()`$, \forall U \in \{\text{arithmetic types}\}\}$. [*Note:* An implementation may choose to not ensure ABI compatibility for `datapar` and `mask` instantiations using the same `datapar_abi::fixed_size<N>` tag. In case of ABI compatibilty between differently compiled translation units, the efficiency of `datapar<T, Abi>` is likely to be better than for `datapar<T, fixed_size<datapar_size_v<T, Abi>>>` (with `Abi` not a instance of `datapar_abi::fixed_size`). — *end note*]

4 An implementation may choose to implement data-parallel execution for many different targets. [*Note:* There can certainly be more than one tag type per (micro-)architecture, e.g. to support different vector lengths or partial register usage. — *end note*] All tag types an implementation supports shall be present independent of the target architecture determined at invocation of the compiler.

5 The `datapar_abi::compatible` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that ensures the highest compatibility on the target architecture.

6 The `datapar_abi::native` tag is defined by the implementation to alias the tag type with the most efficient data parallel execution that is supported on the target system.

3.1.1.2 `datapar` type traits [datapar.traits]

```
template <class T> struct is_datapar;
```

1 The `is_datapar` type derives from `true_type` if `T` is an instance of the `datapar` class template. Otherwise it derives from `false_type`.

```
template <class T> struct is_mask;
```

2 The `is_mask` type derives from `true_type` if `T` is an instance of the `mask` class template. Otherwise it derives from `false_type`.

```
template <class T, size_t N> struct abi_for_size { typedef implementation_defined type; };
```

3 The `abi_for_size` class template defines the member type `type` to one of the tag types in `datapar_abi`. If a tag type `A` exists that satisfies

- `datapar_size_v<T, A> == N`,
- `A` is a supported `Abi` parameter to `datapar<T, Abi>` for the current compilation target, and
- `A` is not `datapar_abi::fixed_size<N>`,

then the member type `type` is an alias for `A`. Otherwise `type` is an alias for `datapar_abi::fixed_size<N>`.

4 `abi_for_size<T, N>::type` shall result in a substitution failure if `T` is not supported by `datapar` or if `N` is not supported by the implementation (cf. [3.1.1.1 p.3]).

```
template <class T, class Abi = datapar_abi::compatible>
struct datapar_size : public integral_constant<size_t, implementation_defined> {};
```

5 The `datapar_size` class template inherits from `integral_constant` with a value that equals `datapar<T, Abi>::size()`.

6 `datapar_size<T, Abi>::value` shall result in a substitution failure if any of the template arguments `T` or `Abi` are invalid template arguments to `datapar`.

```
template <class T, class U = typename T::value_type>
constexpr size_t memory_alignment = implementation_defined;
```

7 *Requires:* The template parameter `T` must be a valid instantiation of either the `datapar` or the `mask` class template.

8 *Requires:* The template parameter `U` must be a type supported by the load and store functions for `T`.

9 The value of `memory_alignment<T, U>` identifies the alignment restrictions on pointers used for (converting) loads and stores for the given type `T` on arrays of type `U`.

3.1.2 Class template `datapar` [datapar]

3.1.2.1 Class template `datapar` overview [datapar.overview]

```
namespace std {
  namespace experimental {
    template <class T, class Abi = datapar_abi::compatible> class datapar {
    public:
      typedef implementation_defined native_handle_type;
      typedef T value_type;
      typedef implementation_defined register_value_type;
      typedef implementation_defined reference;
      typedef mask<T, Abi> mask_type;
```

```
      typedef size_t size_type;
      typedef Abi abi_type;

      static constexpr size_type size();

      datapar() = default;

      datapar(const datapar &) = default;
      datapar(datapar &&) = default;
      datapar &operator=(const datapar &) = default;
      datapar &operator=(datapar &&) = default;

      // implicit broadcast constructor
      datapar(value_type);

      // implicit type conversion constructor
      template <class U> datapar(datapar<U, Abi>);

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

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

      // increment and decrement:
      datapar &operator++();
      datapar operator++(int);
      datapar &operator--();
      datapar operator--(int);

      // unary operators (for integral T)
      mask_type operator!() const;
      datapar operator~() const;

      // unary operators (for any T)
      datapar operator+() const;
      datapar operator-() const;

      // reductions
      value_type sum() const;
      value_type sum(mask_type) const;
      value_type product() const;
      value_type product(mask_type) const;
      value_type min() const;
      value_type min(mask_type) const;
      value_type max() const;
      value_type max(mask_type) const;

      // access to the internals for implementation-specific extensions
      native_handle_type &native_handle();
      const native_handle_type &native_handle() const;
    };
  }
}
```

1 The class template `datapar<T, Abi>` is a one-dimensional smart array. In contrast to `valarray` (26.6), the number of elements in the array is determined at compile time, according to the `Abi` template parameter.

2 The first template argument `T` must be an integral or floating-point fundamental type. The type `bool` is not allowed.
3 The second template argument `Abi` must be a tag type from the `datapar_abi` namespace.

```
typedef implementation_defined native_handle_type;
```

4 The `native_handle_type` member type is an alias for the `native_handle()` member function return type.

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

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

3.1.2.2 `datapar` constructors [datapar.ctor]

```
datapar() = default;
```

1 *Effects:* Constructs an object with all elements initialized to `T()`. [*Note:* This zero-initializes the object. — *end note*]

```
datapar(value_type);
```

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

NOTE 1 Should I add a generator ctor, taking a lambda to initialize the elements?

```
template <class U, class Abi2> datapar(datapar<U, Abi2> x);
```

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

- `Abi` and `Abi2` are equal and `U` and `T` are different integral types and `make_signed<U>::type` equals `make_signed<T>::type`, or
- at least one of `Abi` or `Abi2` is an instantiation of `datapar_abi::fixed_size` and `size() == x.size()` and `U` is implicitly convertible to `T`.

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

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

5 *Effects:* Constructs an object where the i-th element is initialized to `static_cast<T>(mem[i])` for all $i \in [0 \dots$ `size()`$[$.
6 *Remarks:* If `size()` returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.
7 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value is not a multiple of `memory_alignment<datapar, U>`, the behavior is undefined.

3.1.2.3 `datapar` subscript operators [datapar.subscr]

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

1 *Returns:* A temporary object with the following properties:

- *Remarks:* The object is neither *DefaultConstructible*, *CopyConstructible*, *MoveConstructible*, *CopyAssignable*, nor *MoveAssignable*.
- *Remarks:* Assignment, compound assignment, increment, and decrement operators only participate in overload resolution if called in rvalue context and the corresponding operator of type `value_type` is usable.
- *Effects:* The assignment, compound assignment, increment, and decrement operators execute the indicated operation on the i-th element in the `datapar` object.
- *Effects:* Conversion to `value_type` returns a copy of the i-th element.

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

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

3.1.2.4 `datapar` unary operators [datapar.unary]

```
datapar &operator++();
```

1 *Effects:* Increments every element of `*this` by one.

2 *Returns:* An lvalue reference to `*this` after incrementing.

3 *Remarks:* Overflow semantics follow the same semantics as for `T`.

```
datapar operator++(int);
```

4 *Effects:* Increments every element of `*this` by one.

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

6 *Remarks:* Overflow semantics follow the same semantics as for `T`.

```
datapar &operator--();
```

7 *Effects:* Decrements every element of `*this` by one.

8 *Returns:* An lvalue reference to `*this` after decrementing.

9 *Remarks:* Underflow semantics follow the same semantics as for `T`.

```
datapar operator--(int);
```

10 *Effects:* Decrements every element of `*this` by one.

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

12 *Remarks:* Underflow semantics follow the same semantics as for `T`.

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

13 *Returns:* A mask object with the i-th element set to `!operator[](i)` for all $i \in [0 \ldots$ `size()`$[$.

```
datapar operator~() const;
```

14 *Requires:* The first template argument `T` to `datapar` must be an integral type.

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

16 *Remarks:* `datapar::operator~()` shall not participate in overload resolution if `T` is a floating-point type.

```
datapar operator+() const;
```

17 *Returns:* A copy of `*this`

```
datapar operator-() const;
```

18 *Returns:* A new `datapar` object where the i-th element is initialized to `-operator[](i)` for all $i \in$ [0 ... `size()`[.

3.1.2.5 `datapar` reductions [datapar.redu]

```
value_type sum() const;
```

1 *Returns:* The sum of all the elements stored in the `datapar` object. The order of summation is arbitrary.

```
value_type sum(mask_type) const;
```

2 *Returns:* The sum of all the elements stored in the `datapar` object where the corresponding element in the first argument is `true`. The order of summation is arbitrary. If all elements in the first argument are `false`, then the return value is 0.

```
value_type product() const;
```

3 *Returns:* The product of all the elements stored in the `datapar` object. The order of multiplication is arbitrary.

```
value_type product(mask_type) const;
```

4 *Returns:* The product of all the elements stored in the `datapar` object where the corresponding element in the first argument is `true`. The order of multiplication is arbitrary. If all elements in the first argument are `false`, then the return value is 1.

```
value_type min() const;
```

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

```
value_type min(mask_type k) const;
```

6 *Returns:* The value of an element j for which `k[j] == true` and `operator[](j) <= operator[](i) || !k[i]` for all $i \in$ [0 ... `size()`[.

7 *Remarks:* If all elements in `k` are `false`, the return value is undefined.

NOTE 2 Alternatively, it could return `numeric_limits<value_type>::min()`.

```
value_type max() const;
```

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

```
value_type max(mask_type k) const;
```

9 *Returns:* The value of an element j for which `k[j] == true` and `operator[](j) >= operator[](i) || !k[i]` for all $i \in [0 \dots$ `size()` $[$.

10 *Remarks:* If all elements in `k` are `false`, the return value is undefined.

Note 3 Alternatively, it could return `numeric_limits<value_type>::max()`.

3.1.2.6 `datapar` native handles [datapar.native]

```
native_handle_type &native_handle();
```

1 *Remarks:* Whether the function exists is implementation-defined.

2 *Returns:* An lvalue reference to the implementation-defined data member.

3 *Note:* The function exposes an implementation-defined type and interface and provides no guarantee for source and/or binary compatibility.

```
const native_handle_type &native_handle() const;
```

4 *Remarks:* Whether the function exists is implementation-defined.

5 *Returns:* A `const` lvalue reference to the implementation-defined data member.

6 *Note:* The function exposes an implementation-defined type and interface and provides no guarantee for source and/or binary compatibility.

3.1.3 `datapar` non-member operations [datapar.nonmembers]

3.1.3.1 `datapar` load function [datapar.load]

Note 4 Need LEWG input: The load function could return a "load expression" instead. But we still don't have customized decay on template deduction ...

```
template <class T = void, class U, class Flags = load_default>
conditional_t<is_same_v<T, void>, datapar<U>, conditional_t<is_datapar_v<T>, T, datapar<T>>> load(
    const U *, Flags = Flags{});
```

Note 5 Should `Flags` really have a default? It might be a good idea to force users to think about alignment whenever they call load & store. cf. Section 4.11

1 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace and either

- `T` is `void` and `datapar<U>` is a valid template instance,
- or `is_datapar_v<T>` and `U` is implicitly or explicitly convertible to `T::value_type`,
- or `!is_datapar_v<T>` and `datapar<T>` is a valid template instance and `U` is implicitly or explicitly convertible to `T`.

Note 6 sufficient to only say "convertible"?

Note 7 ditto.

2 *Returns:* A new `datapar` object with each element i initialized to `static_cast<T>(x[i])` for all $i \in [0 \dots$ `size()`$[$.

3 *Remarks:* If the `size()` function of the return type returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

4 *Remarks:* If the third template parameter is of type `flags::aligned_tag` and the pointer value is not a multiple of `memory_alignment<`*`return type`*`, U>`, the behavior is undefined.

3.1.3.2 `datapar` store functions [datapar.store]

```
template <class T, class Abi, class U, class Flags = flags::store_default>
void store(const datapar<T, Abi> &x, U *y, Flags = Flags{});
```

1 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace and `T` is implicitly or explicitly convertible to `U`. Note 8 ditto.

2 *Effects:* Copies each element i as if `y[i] = static_cast<U>(x[i])` for all $i \in [0 \dots$ `size()`$[$.

3 *Remarks:* If `datapar<T, Abi>::size()` is greater than the number of values pointed to by `y`, the behavior is undefined.

4 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value of `y` is not a multiple of `memory_alignment<datapar<T, Abi>, U>`, the behavior is undefined.

```
template <class T0, class A0, class U, class T1, class A1, class Flags = flags::store_default>
void store(const datapar<T0, A0> &x, U *y, const mask<T1, A1> &k, Flags = Flags{});
```

5 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace and `T` is implicitly or explicitly convertible to `U` and `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`. Note 9 ditto.

6 *Effects:* Copies each element i where `k[i]` is `true` as if `y[i] = static_cast<U>(x[i])` for all $i \in [0 \dots$ `size()`$[$.

7 *Remarks:* If the largest i where `k[i]` is `true` is greater than the number of values pointed to by `y`, the behavior is undefined. [*Note:* Masked stores only write to the bytes in memory selected by the `k` argument. This prohibits implementations that load, blend, and store the complete vector. — *end note*]

8 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value of `y` is not a multiple of `memory_alignment<datapar<T0, A0>, U>`, the behavior is undefined.

3.1.3.3 `datapar` binary operators [datapar.binary]

```
template <class L, class R> using datapar_return_type = ...;  // exposition only
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
```

```
datapar_return_type<datapar<T, Abi>, U> operator% (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator+ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator- (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator* (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator/ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator% (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator& (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator| (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator^ (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator<<(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
datapar_return_type<datapar<T, Abi>, U> operator>>(const U &, datapar<T, Abi>);
```

1 *Remarks:* The return type of these operators (`datapar_return_type<datapar<T, Abi>, U>`) shall be deduced according to the following rules:

- Let *common*`(A, B)` identify the type: NOTE 10 *unusual arithmetic conversions ;-)*
  - `A` if `A` equals `B`.
  - Otherwise, `A` if `B` is not a fundamental arithmetic type.
  - Otherwise, `B` if `A` is not a fundamental arithmetic type.
  - Otherwise, `decltype(A() + B())` if either one of the types `A` or `B` is a floating-point type.
  - Otherwise, `A` if `sizeof(A) > sizeof(B)`.
  - Otherwise, `B` if `sizeof(A) < sizeof(B)`.
  - Otherwise, `C` shall identify the type with greater integer conversion rank of the types `A` and `B` and:
    * `C` is used if `is_signed_v<A> == is_signed_v<B>`, and
    * `make_unsigned_t<C>` otherwise.
- Let *commonabi*`(A0, A1, T)` identify the type:
  - `A0` if `A0` equals `A1`.
  - Otherwise, `abi_for_size_t<T, datapar_size_v<T, A0>>` if it is equal to either `A0` or `A1`.
  - Otherwise, `datapar_abi::fixed_size<datapar_size_v<T, A0>>`.

- If `is_datapar_v<U> == true` then the return type is `datapar<`*common*`(T, U::value_type), `*commonabi*`(Abi, U::abi_type, `*common*`(T, U::value_type))>`. [*Note:* This rule also matches if `datapar_size_v<T, Abi> != U::size()`. The overload resolution participation condition in the next paragraph discards the operator. — *end note*]
- Otherwise, if `T` is integral and `U` is `int` the return type shall be `datapar<T, Abi>`.
- Otherwise, if `T` is integral and `U` is `unsigned int` the return type shall be `datapar<make_unsigned_t<T>, Abi>`.
- Otherwise, if `U` is a fundamental arithmetic type or `U` is convertible to `int` then the return type shall be `datapar<`*common*`(T, U), `*commonabi*`(Abi, datapar_abi::fixed_size<datapar_size_v<T, Abi>>, `*common*`(T, U))>`.
- Otherwise, if `U` is implicitly convertible to `datapar<V, A>`, where `V` and `A` are determined according to standard template type deduction, then the return type shall be `datapar<`*common*`(T, V), `*commonabi*`(Abi, A, `*common*`(T, V))>`.
- Otherwise, if `U` is implicitly convertible to `datapar<T, Abi>`, the return type shall be `datapar<T, Abi>`.
- Otherwise the operator does not participate in overload resolution.

Note 11 This seems a strange place to put this. Alternatively, modify the above rule to unconditionally use `datapar<T, Abi>`. The paragraph below would lead to the same effect.

2 *Remarks:* Each of these operators only participate in overload resolution if all of the following hold:

- The indicated operator can be applied to objects of type `datapar_return_type<datapar<T, Abi>, U>::value_type`.
- `datapar<T, Abi>` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `U` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.

Note 12 I think this allows `datapar<float, fixed_size<4>>() + double()` and returns `datapar<double, fixed_size<4>>`. I also think that's what we want.

3 *Remarks:* The operators with `const U &` as first parameter shall not participate in overload resolution if `is_datapar_v<U> == true`.

4 *Returns:* A new `datapar` object initialized with the results of the component-wise application of the indicated operator after both operands have been converted to the return type.

3.1.3.4 `datapar` compound assignment [datapar.cassign]

```
template <class T, class Abi, class U> datapar<T, Abi> &operator+= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator-= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator*= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator/= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator%= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator&= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator|= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator^= (datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator<<=(datapar<T, Abi> &, const U &);
template <class T, class Abi, class U> datapar<T, Abi> &operator>>=(datapar<T, Abi> &, const U &);
```

1 *Remarks:* Each of these operators only participates in overload resolution if all of the following hold:

- The indicated operator can be applied to objects of type `datapar_return_type<datapar<T, Abi>, U>::value_type`.
- `datapar<T, Abi>` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.

- `U` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `datapar_return_type<datapar<T, Abi>, U>` is implicitly convertible to `datapar<T, Abi>`.

2 *Effects:* Each of these operators performs the indicated operation component-wise on each of the elements of the first argument and the corresponding element of the second argument after conversion to `datapar<T, Abi>`.

3 *Returns:* A reference to the first argument.

3.1.3.5 `datapar` logical operators [datapar.logical]

Note: The omission of logical operators is deliberate.

3.1.3.6 `datapar` compare operators [datapar.comparison]

```
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (datapar<T, Abi>, const U &);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator==(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator!=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator>=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator<=(const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator> (const U &, datapar<T, Abi>);
template <class T, class Abi, class U>
typename datapar_return_type<datapar<T, Abi>, U>::mask_type operator< (const U &, datapar<T, Abi>);
```

1 *Remarks:* The return type of these operators shall be the `mask_type` member type of the type deduced according to the rules defined in [datapar.binary].

2 *Remarks:* Each of these operators only participates in overload resolution if all of the following hold:

- `datapar<T, Abi>` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.
- `U` is implicitly convertible to `datapar_return_type<datapar<T, Abi>, U>`.

3 *Remarks:* The operators with `const U &` as first parameter shall not participate in overload resolution if `is_datapar_v<U> == true`.

4 *Returns:* A new `mask` object initialized with the results of the component-wise application of the indicated operator after both operands have been converted to `datapar_return_type<datapar<T, Abi>, U>`.

3.1.3.7 `datapar` casts [datapar.casts]

```
template <class T, class U, class... Us>
conditional_t<(T::size() == (U::size() + Us::size()...)), T,
          array<T, (U::size() + Us::size()...) / T::size()>> datapar_cast(U, Us...);
```

1 *Remarks:* The `datapar_cast` function only participates in overload resolution if all of the following hold:

- `is_datapar_v<T>`
- `is_datapar_v<U>`
- All types in the template parameter pack `Us` are equal to `U`.
- `U::size() + Us::size()...` is an integral multiple of `T::size()`.

2 *Returns:* A new `datapar` object initialized with the converted values as one object of `T` or an array of `T`. All scalar elements x_i of the function argument(s) are converted as if y_i `= static_cast<typename T::value_type>(`x_i`)` is executed. The resulting y_i intialize the return object(s) of type `T`. [*Note:* For `T::size() == 2 * U::size()` the following holds: `datapar_cast<T>(x0, x1)[i] == static_cast<typename T::value_type>(array<U, 2>{x0, x1}[i / U::size()][i % U::size()])`. For `2 * T::size() == U::size()` the following holds: `datapar_cast<T>(x)[i][j] == static_cast<typename T::value_type>(x[i * T::size() + j])`. — *end note*]

3.1.3.8 `datapar` math library [datapar.math]

1 Table 1 summarizes the `<cmath>` and `<cstdlib>` functions that are overloaded with `datapar` arguments.

Math Functions:				
`abs`	`cbrt`	`ceil`	`copysign`	`fdim`
`floor`	`fma`	`fmax`	`fmin`	`fmod`
`frexp`	`hypot`	`ilogb`	`ldexp`	`logb`
`lrint`	`lround`	`modf`	`nan`	`nanf`
`nanl`	`nearbyint`	`nextafter`	`nexttoward`	`remainder`
`remquo`	`rint`	`round`	`scalbln`	`scalbn`
`sqrt`	`trunc`			
Classification/comparison Functions:				
`fpclassify`	`isfinite`	`isgreater`	`isgreaterequal`	`isinf`
`isless`	`islessequal`	`islessgreater`	`isnan`	`isnormal`
`isunordered`	`signbit`			
Integer Functions:				
`abs`	`div`	`labs`	`ldiv`	`llabs`
`lldiv`				

Table 1: Overloads of `<cmath>` and `<cstdlib>` functions for `datapar`

2 Each of these functions is provided for arguments of types `datapar<float, Abi>`, `datapar<double, Abi>`, and `datapar<long double, Abi>`, where `Abi` is any of the `datapar_abi` types supported by the implementation. The detailed signatures are:

NOTE 13 Is there any function missing that we want to see in the first round? SG1 voted against trig functions (and I assume that includes `exp` and `log` as well).

NOTE 14 I would not mind dropping `long double` support.

```
namespace std {
  namespace experimental {
    template <class Abi> using floatv = datapar<float, Abi>;  // exposition only
    template <class Abi> using doublev = datapar<double, Abi>;  // exposition only
    template <class Abi> using ldoublev = datapar<long double, Abi>;  // exposition only
    template <class T, class V>
    using samesize = datapar<T, abi_for_size_t<V::size()>>;  // exposition only

    template <class Abi> floatv<Abi> abs(floatv<Abi>);
    template <class Abi> floatv<Abi> cbrt(floatv<Abi>);
    template <class Abi> floatv<Abi> ceil(floatv<Abi>);
    template <class Abi> floatv<Abi> copysign(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> fdim(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> floor(floatv<Abi>);
    template <class Abi> floatv<Abi> fma(floatv<Abi>, floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> fmax(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> fmin(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> fmod(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> frexp(floatv<Abi>, samesize<int, floatv<Abi>> *);
    template <class Abi> floatv<Abi> hypot(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> hypot(floatv<Abi>, floatv<Abi>, floatv<Abi>);
    template <class Abi> samesize<int, floatv<Abi>> ilogb(floatv<Abi>);
    template <class Abi> floatv<Abi> ldexp(floatv<Abi>, samesize<int, floatv<Abi>>);
    template <class Abi> floatv<Abi> logb(floatv<Abi>);
    template <class Abi> samesize<long, floatv<Abi>> lrint(floatv<Abi>);
    template <class Abi> samesize<long, floatv<Abi>> lround(floatv<Abi>);
    template <class Abi> floatv<Abi> modf(floatv<Abi>, floatv<Abi> *);
    template <class Abi> floatv<Abi> nan(floatv<Abi>);
    template <class Abi> floatv<Abi> nanf(floatv<Abi>);
    template <class Abi> floatv<Abi> nanl(floatv<Abi>);
    template <class Abi> floatv<Abi> nearbyint(floatv<Abi>);
    template <class Abi> floatv<Abi> nextafter(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> nexttoward(floatv<Abi>, samesize<long double, floatv<Abi>>);
    template <class Abi> floatv<Abi> remainder(floatv<Abi>, floatv<Abi>);
    template <class Abi> floatv<Abi> remquo(floatv<Abi>, floatv<Abi>, samesize<int, floatv<Abi>> *);
    template <class Abi> floatv<Abi> rint(floatv<Abi>);
    template <class Abi> floatv<Abi> round(floatv<Abi>);
    template <class Abi> floatv<Abi> scalbln(floatv<Abi>, samesize<long, floatv<Abi>>);
    template <class Abi> floatv<Abi> scalbn(floatv<Abi>, samesize<int, floatv<Abi>>);
    template <class Abi> floatv<Abi> sqrt(floatv<Abi>);
    template <class Abi> floatv<Abi> trunc(floatv<Abi>);

    template <class Abi> doublev<Abi> abs(doublev<Abi>);
    template <class Abi> doublev<Abi> cbrt(doublev<Abi>);
    template <class Abi> doublev<Abi> ceil(doublev<Abi>);
    template <class Abi> doublev<Abi> copysign(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> fdim(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> floor(doublev<Abi>);
    template <class Abi> doublev<Abi> fma(doublev<Abi>, doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> fmax(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> fmin(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> fmod(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> frexp(doublev<Abi>, samesize<int, doublev<Abi>> *);
    template <class Abi> doublev<Abi> hypot(doublev<Abi>, doublev<Abi>);
    template <class Abi> doublev<Abi> hypot(doublev<Abi>, doublev<Abi>, doublev<Abi>);
    template <class Abi> samesize<int, doublev<Abi>> ilogb(doublev<Abi>);
    template <class Abi> doublev<Abi> ldexp(doublev<Abi>, samesize<int, doublev<Abi>>);
    template <class Abi> doublev<Abi> logb(doublev<Abi>);
```

```
template <class Abi> samesize<long, doublev<Abi>> lrint(doublev<Abi>);
template <class Abi> samesize<long, doublev<Abi>> lround(doublev<Abi>);
template <class Abi> doublev<Abi> modf(doublev<Abi>, doublev<Abi> *);
template <class Abi> doublev<Abi> nan(doublev<Abi>);
template <class Abi> doublev<Abi> nanf(doublev<Abi>);
template <class Abi> doublev<Abi> nanl(doublev<Abi>);
template <class Abi> doublev<Abi> nearbyint(doublev<Abi>);
template <class Abi> doublev<Abi> nextafter(doublev<Abi>, doublev<Abi>);
template <class Abi> doublev<Abi> nexttoward(doublev<Abi>, samesize<long double, doublev<Abi>>);
template <class Abi> doublev<Abi> remainder(doublev<Abi>, doublev<Abi>);
template <class Abi> doublev<Abi> remquo(doublev<Abi>, doublev<Abi>, samesize<int, doublev<Abi>> *);
template <class Abi> doublev<Abi> rint(doublev<Abi>);
template <class Abi> doublev<Abi> round(doublev<Abi>);
template <class Abi> doublev<Abi> scalbln(doublev<Abi>, samesize<long, doublev<Abi>>);
template <class Abi> doublev<Abi> scalbn(doublev<Abi>, samesize<int, doublev<Abi>>);
template <class Abi> doublev<Abi> sqrt(doublev<Abi>);
template <class Abi> doublev<Abi> trunc(doublev<Abi>);

template <class Abi> ldoublev<Abi> abs(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> cbrt(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> ceil(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> copysign(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> fdim(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> floor(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> fma(ldoublev<Abi>, ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> fmax(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> fmin(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> fmod(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> frexp(ldoublev<Abi>, samesize<int, ldoublev<Abi>> *);
template <class Abi> ldoublev<Abi> hypot(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> hypot(ldoublev<Abi>, ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> samesize<int, ldoublev<Abi>> ilogb(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> ldexp(ldoublev<Abi>, samesize<int, ldoublev<Abi>>);
template <class Abi> ldoublev<Abi> logb(ldoublev<Abi>);
template <class Abi> samesize<long, ldoublev<Abi>> lrint(ldoublev<Abi>);
template <class Abi> samesize<long, ldoublev<Abi>> lround(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> modf(ldoublev<Abi>, ldoublev<Abi> *);
template <class Abi> ldoublev<Abi> nan(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> nanf(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> nanl(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> nearbyint(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> nextafter(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> nexttoward(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> remainder(ldoublev<Abi>, ldoublev<Abi>);
template <class Abi> ldoublev<Abi> remquo(ldoublev<Abi>, ldoublev<Abi>, samesize<int, ldoublev<Abi>> *);
template <class Abi> ldoublev<Abi> rint(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> round(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> scalbln(ldoublev<Abi>, samesize<long, ldoublev<Abi>>);
template <class Abi> ldoublev<Abi> scalbn(ldoublev<Abi>, samesize<int, ldoublev<Abi>>);
template <class Abi> ldoublev<Abi> sqrt(ldoublev<Abi>);
template <class Abi> ldoublev<Abi> trunc(ldoublev<Abi>);

template <class Abi> samesize<int, floatv<Abi>> fpclassify(floatv<Abi>);
template <class Abi> mask<float, Abi> isfinite(floatv<Abi>);
template <class Abi> mask<float, Abi> isgreater(floatv<Abi>);
template <class Abi> mask<float, Abi> isgreaterequal(floatv<Abi>);
template <class Abi> mask<float, Abi> isinf(floatv<Abi>);
template <class Abi> mask<float, Abi> isless(floatv<Abi>);
```

```
    template <class Abi> mask<float, Abi> islessequal(floatv<Abi>);
    template <class Abi> mask<float, Abi> islessgreater(floatv<Abi>);
    template <class Abi> mask<float, Abi> isnan(floatv<Abi>);
    template <class Abi> mask<float, Abi> isnormal(floatv<Abi>);
    template <class Abi> mask<float, Abi> isunordered(floatv<Abi>);
    template <class Abi> mask<float, Abi> signbit(floatv<Abi>);

    template <class Abi> samesize<int, doublev<Abi>> fpclassify(doublev<Abi>);
    template <class Abi> mask<double, Abi> isfinite(doublev<Abi>);
    template <class Abi> mask<double, Abi> isgreater(doublev<Abi>);
    template <class Abi> mask<double, Abi> isgreaterequal(doublev<Abi>);
    template <class Abi> mask<double, Abi> isinf(doublev<Abi>);
    template <class Abi> mask<double, Abi> isless(doublev<Abi>);
    template <class Abi> mask<double, Abi> islessequal(doublev<Abi>);
    template <class Abi> mask<double, Abi> islessgreater(doublev<Abi>);
    template <class Abi> mask<double, Abi> isnan(doublev<Abi>);
    template <class Abi> mask<double, Abi> isnormal(doublev<Abi>);
    template <class Abi> mask<double, Abi> isunordered(doublev<Abi>);
    template <class Abi> mask<double, Abi> signbit(doublev<Abi>);

    template <class Abi> samesize<int, ldoublev<Abi>> fpclassify(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isfinite(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isgreater(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isgreaterequal(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isinf(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isless(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> islessequal(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> islessgreater(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isnan(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isnormal(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> isunordered(ldoublev<Abi>);
    template <class Abi> mask<long double, Abi> signbit(ldoublev<Abi>);
  }
}
```

3 The signatures of the integer functions are:

```
namespace std {
  namespace experimental {
    template <class Abi> datapar<int, Abi> abs(datapar<int, Abi>);
    template <class Abi> datapar<long, Abi> abs(datapar<long, Abi>);
    template <class Abi> datapar<long long, Abi> abs(datapar<long long, Abi>);
    template <class Abi> datapar<long, Abi> labs(datapar<long, Abi>);
    template <class Abi> datapar<long long, Abi> llabs(datapar<long long, Abi>);

    template <class V> struct datapar_div_t { V quot, rem; };
    template <class Abi> datapar_div_t<datapar<int, Abi>> div(datapar<int, Abi>);
    template <class Abi> datapar_div_t<datapar<long, Abi>> ldiv(datapar<long, Abi>);
    template <class Abi> datapar_div_t<datapar<long long, Abi>> lldiv(datapar<long long, Abi>);
  }
}
```

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

3.1.4 Class template `mask` [mask]

3.1.4.1 Class template `mask` overview [mask.overview]

```
namespace std {
  namespace experimental {
    template <class T, class Abi = datapar_abi::compatible> class mask {
    public:
      typedef implementation_defined native_handle_type;
      typedef bool value_type;
      typedef implementation_defined register_value_type;
      typedef implementation_defined reference;
      typedef datapar<T, Abi> datapar_type;
      typedef size_t size_type;
      typedef Abi abi_type;

      static constexpr size_type size();

      mask() = default;

      mask(const mask &) = default;
      mask(mask &&) = default;
      mask &operator=(const mask &) = default;
      mask &operator=(mask &&) = default;

      // implicit broadcast constructor
      mask(value_type);

      // implicit type conversion constructor
      template <class U> mask(mask<U, Abi>);

      // load constructor
      template <class Flags> mask(const bool *mem, Flags);

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

      // negation:
      mask operator!() const;

      // access to the internals for implementation-specific extensions
      native_handle_type &native_handle();
      const native_handle_type &native_handle() const;
    };
  }
}
```

1 The class template `mask<T, Abi>` is a one-dimensional smart array of booleans. The number of elements in the array is determined at compile time, equal to the number of elements in `datapar<T, Abi>`.

2 The first template argument `T` must be an integral or floating-point fundamental type. The type `bool` is not allowed.

3 The second template argument `Abi` must be a tag type from the `datapar_abi` namespace.

```
typedef implementation_defined native_handle_type;
```

4 The `native_handle_type` member type is an alias for the `native_handle()` member function return type. It is used to expose an implementation-defined handle for implementation- and target-specific extensions.

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

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

3.1.4.2 `mask` constructors [mask.ctor]

```
mask() = default;
```

1 *Effects:* Constructs an object with all elements initialized to `bool()`. [*Note:* This zero-initializes the object. — *end note*]

```
mask(value_type);
```

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

```
template <class U> mask(mask<U, Abi> x);
```

3 *Remarks:* This constructor shall not participate in overload resolution unless `datapar<U, Abi>` is implicitly convertible to `datapar<T, Abi>`.

4 *Effects:* Constructs an object of type `mask` where the i-th element equals `x[i]` for all $i \in [0 \dots$ `size()`$[$.

```
template <class Flags> mask(const bool *mem, Flags);
```

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

6 *Remarks:* If `size()` returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

7 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value is not a multiple of `memory_alignment<mask >`, the behavior is undefined.

3.1.4.3 `mask` subscript operators [mask.subscr]

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

1 *Returns:* A temporary object with the following properties:

- *Remarks:* The object is neither *DefaultConstructible*, *CopyConstructible*, *MoveConstructible*, *CopyAssignable*, nor *MoveAssignable*.
- *Remarks:* Assignment, compound assignment, increment, and decrement operators only participate in overload resolution if called in rvalue context and the corresponding operator of type `value_type` is usable.
- *Effects:* The assignment, compound assignment, increment, and decrement operators execute the indicated operation on the i-th element in the `datapar` object.
- *Effects:* Conversion to `value_type` returns a copy of the i-th element.

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

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

3.1.4.4 mask unary operators [mask.unary]

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

1 *Returns:* A mask object with the i-th element set to the logical negation for all $i \in [0 \ldots \texttt{size()}[$.

3.1.4.5 mask native handles [mask.native]

```
native_handle_type &native_handle();
```

1 *Returns:* An lvalue reference to the implementation-defined data member.

2 *Note:* The function exposes an implementation-defined type and interface and provides no guarantee for source and/or binary compatibility.

```
const native_handle_type &native_handle() const;
```

3 *Returns:* A `const` lvalue reference to the implementation-defined data member.

4 *Note:* The function exposes an implementation-defined type and interface and provides no guarantee for source and/or binary compatibility.

3.1.5 mask non-member operations [mask.nonmembers]

3.1.5.1 mask load function [mask.load]

```
template <class T, class Flags = flags::load_default> T load(const bool *, Flags = Flags{});
```

1 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace and `is_mask_v<T>`.

2 *Returns:* A new `mask` object with each element i initialized to `x[i]` for all $i \in [0 \ldots \texttt{size()}[$.

3 *Remarks:* If `T::size()` returns a value greater than the number of values pointed to by the first argument, the behavior is undefined.

4 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value is not a multiple of `memory_alignment<T>`, the behavior is undefined.

3.1.5.2 mask store functions [mask.store]

```
template <class T, class Abi, class Flags = flags::store_default>
void store(const mask<T, Abi> &x, bool *y, Flags = Flags{});
```

1 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace.

2 *Effects:* Copies each element `x[i]` to `y[i]` for all $i \in [0 \ldots \texttt{size()}[$.

3 *Remarks:* If `mask<T, Abi> ::size()` is greater than the number of values pointed to by `y`, the behavior is undefined.

4 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value of `y` is not a multiple of `memory_alignment<mask<T, Abi> >`, the behavior is undefined.

```
template <class T0, class A0, class T1, class A1, class Flags = flags::store_default>
void store(const mask<T0, A0> &x, bool *y, const mask<T1, A1> &k, Flags = Flags{});
```

5 *Remarks:* This function shall not participate in overload resolution unless `Flags` is one of the tag types in the `flags` namespace and `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`.

6 *Effects:* Copies each element `x[i]` where `k[i]` is `true` to `y[i]` for all $i \in [0 \ldots$ `size()` $[$.

7 *Remarks:* If the largest i where `k[i]` is `true` is greater than the number of values pointed to by `y`, the behavior is undefined. [*Note:* Masked stores only write to the bytes in memory selected by the `k` argument. This prohibits implementations that load, blend, and store the complete vector. — *end note*]

8 *Remarks:* If the `Flags` template parameter is of type `flags::aligned_tag` and the pointer value of `y` is not a multiple of `memory_alignment<datapar<T0, A0>, U>`, the behavior is undefined.

3.1.5.3 `mask` binary operators [mask.binary]

```
template <class T0, class A0, class T1, class A1> using mask_return_type = ...   // exposition only
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator&&(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator||(const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator& (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator| (const mask<T0, A0> &, const mask<T1, A1> &);
template <class T0, class A0, class T1, class A1>
mask_return_type<T0, A0, T1, A1> operator^ (const mask<T0, A0> &, const mask<T1, A1> &);
```

Note 15 I think we need an additional `operator@(const mask<T, A> &, const mask<T, A> &)` overload to enable use of objects that are implicitly convertible to more than one `mask` instantiation with equal priority.

1 *Remarks:* The return type, `mask_return_type<T, Abi, U>`, shall be `mask<`*`common`*`(T0, T1),` *`commonabi`*`(A0, A1,` *`common`*`(T0, T1))>`. The functions *common* and *commonabi* identify the functions described in 3.1.3.3 p.1.

2 *Remarks:* Each of these operators only participate in overload resolution if both arguments are implicitly convertible to `mask_return_type<T, Abi, U>`.

3 *Returns:* A new `mask` object initialized with the results of the component-wise application of the indicated operator.

3.1.5.4 `mask` compares [mask.comparison]

```
template <class T0, class A0, class T1, class A1>
bool operator==(const mask<T0, A0> &, const mask<T1, A1> &);
```

Note 16 ditto.

1 *Remarks:* This operator only participates in overload resolution if `mask<T0, A0>` is implicitly convertible to `mask<T1, A1>` or `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`.

2 *Returns:* `true` if all boolean elements of the first argument equal the corresponding element of the second argument. It returns `false` otherwise.

```
template <class T0, class A0, class T1, class A1>
bool operator!=(const mask<T0, A0> &a, const mask<T1, A1> &b);
```

NOTE 17 ditto.

3 *Remarks:* This operator only participates in overload resolution if `mask<T0, A0>` is implicitly convertible to `mask<T1, A1>` or `mask<T1, A1>` is implicitly convertible to `mask<T0, A0>`.

4 *Returns:* `!operator==(a, b)`.

3.1.5.5 `mask` reductions [mask.reductions]

```
template <class T, class Abi> bool  all_of(mask<T, Abi>);
constexpr bool  all_of(bool);
```

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

```
template <class T, class Abi> bool  any_of(mask<T, Abi>);
constexpr bool  any_of(bool);
```

2 *Returns:* `true` if at least one boolean element in the function argument equals `true`, `false` otherwise.

```
template <class T, class Abi> bool none_of(mask<T, Abi>);
constexpr bool none_of(bool);
```

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

```
template <class T, class Abi> bool some_of(mask<T, Abi>);
constexpr bool some_of(bool);
```

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

5 *Note:* `some_of(bool)` unconditionally returns `false`.

```
template <class T, class Abi> int popcount(mask<T, Abi>);
constexpr int popcount(bool);
```

6 *Returns:* The number of boolean elements that are `true`.

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

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

8 *Remarks:* If `none_of(m) == true` the behavior is undefined.

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

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

10 *Remarks:* If `none_of(m) == true` the behavior is undefined.

```
constexpr int find_first_set(bool);
constexpr int find_last_set(bool);
```

11 *Returns:* 0 if the argument is `true`.

3.1.5.6 Masked assigment [mask.where]

```
template <class T0, class A0, class T1, class A1>
implementation_defined where(const mask<T0, A0> &m, datapar<T1, A1> &v);
```

1 *Remarks:* The function only participates in overload resolution if mask<T0, A0> is implicitly convertible to mask<T1, A1>.

2 *Returns:* A temporary object with the following properties:

- The object is not *CopyConstructible*.
- Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for datapar<T1, A1> is usable.
- *Effects:* Assignment and compound assignment implement the same semantics as the corresponding operator for datapar<T1, A1> with the exception that elements of v stay unmodified if the corresponding boolean element in m is false.
- The assignment and compound assignment operators return void.

```
template <class T> implementation_defined where(bool, T &);
```

3 *Remarks:* The function only participates in overload resolution if T is a fundamental arithmetic type.

4 *Returns:* A temporary object with the following properties:

- The object is not *CopyConstructible*.
- Assignment and compound assignment operators only participate in overload resolution if the corresponding operator for T is usable.
- *Effects:* If the first argument is false, the assignment operators do nothing. If the first argument is true, the assignment operators forward to the corresponding builtin assignment operator.
- The assignment and compound assignment operators return void.

---

4 DISCUSSION

4.1 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.

register_value_type
: On some targets it may be beneficial to implement datapar instantiations of some T with a different type register_value_type, which has higher precision

than `T`. This is mostly an implementation detail, but can be important to know in some situations, especially whenever `native_handle_type` is involved.

Requesting Guidance: A better name might be `native_value_type`.

`native_handle_type`
: The type used for enabling access to an implementation-defined member object (via the `native_handle()` function).

`reference`
: Used as the return type of the non-const scalar subscript operator. This may use implementation-defined means to solve possible type aliasing issues.

`const_reference`
: Used as the return type of the const scalar subscript operator. From my experience with Vc, it is safest to actually not use a const lvalue reference here, but a temporary.

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

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

`abi_type`
: The `Abi` template parameter to `datapar`.

4.2 Default Construction

The default constructors of `datapar` and `mask` zero-initialize the object. This is important for compatibility with `T()`, which zero-initializes fundamental types. There may be a concern that unnecessary initialization could lead to unnecessary instructions. I consider this a QoI issue. Implementations are certainly able to recognize unnecessary initializations in many cases.

4.3 Conversions

The `datapar` conversion constructor only allows implicit conversion from `datapar` 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 `datapar_cast` consistently for all other conversions.

4.4 Broadcast constructor

The broadcast constructor is not declared as `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.

4.5 Aliasing of subscript operators

Note that the way the subscript operators are declared, some kind of type punning needs to happen.[1] An lvalue reference to `register_value_type` needs to reference the same object as is contained as one element of the `datapar` object. An alternative to an lvalue reference would be a smart reference object. This would require progress on language improvements for smart references first.

The subscript operator of the `mask` type, on the other hand, may not use lvalue references and must use a smart reference wrapper instead. This is necessary because there are systems where a single boolean element is stored as a single bit. To ensure source compatibility the return type must therefore be a smart reference on all implementations.

4.6 Parameters of binary and compare operators

It is easier to implement and possibly easier to specify these operators if the signature is specified as:

```
template <class T, class U>
datapar_return_type<T, U> operator+(const T &, const U &);
```

1 Note: The vector builtins of clang do not suffice to implement the subscript operators, even though they support subscripting the vector object. An implementation might have to use a mechanism such as the `gnu::may_alias` attribute.

The motivation for using a variant where at least one function parameter is constrained to the `datapar` class template is compilation speed. The compiler can drop the operator from the overload resolution set quicker. With concepts it might be worthwhile to revisit this decision.

4.7 COMPOUND ASSIGNMENT

The semantics of compound assignment would allow less strict implicit conversion rules. Consider `datapar<int>() *= double()`: the corresponding multiplication operator would not compile because the implicit conversion to `datapar<float>` 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 suddenly would not compile anymore.
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.

4.8 RETURN TYPE OF MASKED ASSIGNMENT OPERATORS

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

- A reference to the `datapar` object that was modified.
- A temporary `datapar` object that only contains the elements where the `mask` is `true`.
- The object returned from the `where` function.
- Nothing (i. e. `void`).

My first choice was a reference to the modified `datapar` 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 `mask` in the middle of the expression.

```
template <class T, size_t N = datapar_size_v<T, datapar_abi::compatible>,
          class Abi = datapar_abi::compatible>
class datapar;
```

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

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.

4.9 Fundamental SIMD type or not?

4.9.1 The issue

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

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

4.9.2 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).

4.9.3 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 `datapar<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 = datapar<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.

4.10 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 `datapar` or a SIMD extension such as vector builtins or SIMD intrinsics might decide against `datapar` just for fear of not being able to access all functionality.[2]

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 `datapar` 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 `datapar`.

4.11 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 `datapar` 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.[3] For Vc 2.0 I will drop the default.

2 Whether that's a reasonable fear is a different discussion.
3 As I realized too late.

For `datapar` 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.

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).

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: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4184.pdf.

[N4185] Matthias Kretz. *N4185: SIMD Types: The Mask Type & Write-Masking*. ISO/IEC C++ Standards Committee Paper. 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4185.pdf.

[N4395] Matthias Kretz. *N4395: SIMD Types: ABI Considerations*. ISO/IEC C++ Standards Committee Paper. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4395.pdf.

[N4454] Matthias Kretz. *N4454: SIMD Types Example: Matrix Multiplication*. ISO/IEC C++ Standards Committee Paper. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4454.pdf.

[P0214R0] Matthias Kretz. *P0214R0: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2016. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0214r0.pdf.