Document Number:	P3932R0
Date:	2026-02-13
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LWG
Target:	C++26

Fix LWG4470: Fix integer-from in [simd]

ABSTRACT

This paper resolves LWG4470. Since the resolution needs to modify wording that LWG4414 and LWG4518 also need to modify, this paper additionally resolves LWG4414 and LWG4518.

CONTENTS

1 CHANGELOG

(placeholder)

2 THE ISSUE(S) WITH INTEGER-FROM

2.1 4470: THE USE OF INTEGER-FROM<BYTES> ALL OVER [SIMD] IS INCORRECT FOR [...]

In the discussion of LWG4238, Tim noted that *integer-from*<Bytes> does not work as intended because Bytes can be 16 after complex<double> became a vectorizable type.[1] I then opened LWG4470, which this paper aims to resolve. LWG4470:

> After the introduction of complex<double> to the set of vectorizable types, the *integer-from*<Bytes> trait does not work as intended anymore. All uses of *integer-from* need to be reviewed and the wording needs to be adjusted to work for Bytes=16.

This issue is a question on how we define masks and their ABIs. I sketched a potential solution in https://lists.isocpp.org/lib/2025/04/31251.php.

2.2 4518: SIMD::CAT RETURN TYPE REQUIRES INEFFICIENT ABI TAG CHANGE / CONVERSION

Later I noticed that a surprising change in ABI can happen on std::simd::cat. This is LWG4518:

> The return type of simd::cat is defined using *deduce-abi-t* rather than resize_t. This can lead to:
>
> ```
> basic_vec<T, Abi> x = ...
> auto [...vs] = simd::chunk<2>(x);
> auto y = simd::cat(vs...);
> static_assert(is_same_v<decltype(x), decltype(y)>); // can fail
> ```
>
> This happens when bit-mask and vec-mask types are mixed, and can also happen (in my implementation) with complex value types. simd::cat should try to be conservative wrt. the resulting ABI tag. However, simd::cat allows different ABI tags on its arguments (to allow different width). If the user gives mixed input, there's no obvious correct answer. I suggest the simple heuristic of using the first argument with resize_t.

Since the simd::cat wording currently uses *integer-from* it makes sense to resolve the two issues together.

1 It was not intended to require conforming implementations to now provide a 128-bit signed integer type.

2.3 4414: §[simd.expos.abi] deduce-abi-t is underspecified and incorrectly referenced from rebind and resize

Finally, since LWG4414 changes the same wording we need to change for LWG4470, I incorporated the proposed resolution into the wording of this paper. LWG4414:

> In 29.10.2.2 [simd.expos.abi], *deduce-abi-t* is specified to be defined for some arguments. For all remaining arguments, nothing is specified. This could be interpreted to make such specializations ill-formed. But that does not match the intent of making `simd::vec<std::string>` and `simd::vec<int, INT_MAX>` disabled specializations of `basic_vec`. (If `INT_MAX` is not supported by the implementation.)
>
> The wording needs to clarify what happens in those cases.
>
> In 29.10.4 [simd.traits], `rebind` and `resize` say "*deduce-abi-t*`<T, V::size()>` has a member type `type`". But that's not how *deduce-abi-t* is specified.

3 WORDING

3.1 FEATURE TEST MACRO

Bump? There are minor behavior changes, but I recommend no change to the macro.

3.2 MODIFY [SIMD.EXPOS]

In [simd.expos], add:

[simd.expos]

```
using simd-size-type = see below;                                  // exposition only
template<size_t Bytes> using integer-from = see below;             // exposition only

template<class T, class Abi>
  constexpr simd-size-type simd-size-v = see below;                // exposition only
template<size_t Bytes, class Abi>
  constexpr simd-size-type mask-size-v = see below;                // exposition only
template<class T> constexpr size_t mask-element-size = see below;  // exposition only
```

3.3 MODIFY [SIMD.EXPOS.DEFN]

In [simd.expos.defn], change:

[simd.expos.defn]

```
template<class T, class Abi>
  constexpr simd-size-type simd-size-v = see below;
```

3 *simd-size-v*<T, Abi> denotes the width of basic_vec<T, Abi> if the specialization basic_vec<T, Abi> is enabled, or 0 otherwise.

```
template<size_t Bytes, class Abi>
  constexpr simd-size-type mask-size-v = see below;
```

-?- *mask-size-v*<Bytes, Abi> denotes the width of basic_mask<Bytes, Abi> if the specialization basic_mask<Bytes, Abi> is enabled, or 0 otherwise.

```
template<class T> constexpr size_t mask-element-size = see below;
```

3.4 MODIFY [SIMD.EXPOS.ABI]

In [simd.expos.abi], change:

[simd.expos.abi]

4 *deduce-abi-t*<T, N> ~~is defined~~names an ABI tag type if and only if

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

Otherwise, *deduce-abi-t*<T, N> names an unspecified type.

The implementation-defined maximum for N is not smaller than 64 and can differ depending on T.

5 ~~Where present,~~If *deduce-abi-t*<T, N> names an ABI tag type ~~such that~~, the following is true:

- *simd-size-v*<T, *deduce-abi-t*<T, N>> equals N, and
- basic_vec<T, *deduce-abi-t*<T, N>> is enabled ([simd.overview])~~, and~~
- ~~basic_mask<sizeof(T), *deduce-abi-t*<*integer-from*<sizeof(T)>, N> is enabled~~.

3.5 modify [simd.syn]

In [simd.syn], change:

[simd.syn]

```
  template<class T, class... Abis>
    constexpr basic_vec<T, deduce-abi-t<T, resize_t<(basic_vec<T, Abis>::size() + ...), basic_vec<T,
Abis...[0]>>
      cat(const basic_vec<T, Abis>&...) noexcept;
  template<size_t Bytes, class... Abis>
    constexpr basic_mask<Bytes, deduce-abi-t<integer-from<Bytes>,resize_t<
                              (basic_mask<Bytes, Abis>::size() + ...), basic_mask<Bytes, Abis...[0]>>
      cat(const basic_mask<Bytes, Abis>&...) noexcept;
[...]

  // [simd.mask.class], class template basic_mask
  template<size_t Bytes, class Abi = native-abi<integer-from<Bytes>>> class basic_mask;
  template<class T, simd-size-type N = simd-size-v<T, native-abi<T>>>
    using mask = basic_mask<sizeof(T), deduce-abi-t<T, N>>vec<T, N>::mask_type;
```

3.6 modify [simd.traits]

[simd.traits]

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

4 The member `type` is present if and only if

- `V` is a data-parallel type,
- `T` is a vectorizable type, and
- *deduce-abi-t*`<T, V::size()>` has a member type typenames an ABI tag type.

5 If `V` is a specialization of `basic_vec`, let `Abi1` denote an ABI tag such that `basic_vec<T, Abi1>::size()` equals `V::size()`. If `V` is a specialization of `basic_mask`, let `Abi1` denote an ABI tag such that `basic_mask<sizeof(T), Abi1>::size()` equals `V::size()`.

6 Where present, the member typedef `type` names `basic_vec<T, Abi1>` if `V` is a specialization of `basic_vec` or `basic_mask<sizeof(T), Abi1>` if `V` is a specialization of `basic_mask`.

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

7 Let TAbi1 denote an ABI tag

- typename V::value_typesuch that *simd-size-v*`<typename V::value_type, Abi1>` equals `N` if `V` is a specialization of `basic_vec`,
- otherwise *integer-from*`<mask-element-size<V>`such that *mask-size-v*`<mask-element-size<V>, Abi1>` equals `N` if `V` is a specialization of `basic_mask`.

8 The member `type` is present if and only if

- `V` is a data-parallel type, and
- *deduce-abi-t*`<T, N>` has a member type typethere exists at least one ABI tag that satisfies the above constraints for `Abi1`.

9 ~~If `V` is a specialization of `basic_vec`, let `Abi1` denote an ABI tag such that `basic_vec<T, Abi1>::size()` equals `N`. If `V` is a specialization of `basic_mask`, let `Abi1` denote an ABI tag such that `basic_mask<sizeof(T), Abi1>::size()` equals `N`.~~

10 Where present, the member typedef `type` names `basic_vec<`~~`T`~~`typename V::value_type, Abi1>` if `V` is a specialization of `basic_vec` or `basic_mask<`~~`sizeof(T)`~~*`mask-element-size`*`<V>, Abi1>` if `V` is a specialization of `basic_mask`.

3.7 MODIFY [SIMD.OVERVIEW]

[simd.overview]

1 Every specialization of `basic_vec` is a complete type. The specialization of `basic_vec<T, Abi>` is

- enabled, if `T` is a vectorizable type, and there exists value `N` in the range $[1, 64]$, such that `Abi` ~~is~~names the ABI tag type denoted by *`deduce-abi-t`*`<T, N>`,
- otherwise, disabled, if `T` is not a vectorizable type,
- otherwise, it is implementation-defined if such a specialization is enabled.

If `basic_vec<T, Abi>` is disabled, then the specialization has a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. In addition only the `value_type`, `abi_type`, and `mask_type` members are present.

If `basic_vec<T, Abi>` is enabled, then

- `basic_vec<T, Abi>` is trivially copyable,
- default-initialization of an object of such a type default-initializes all elements, ~~and~~
- value-initialization value-initializes all elements ([dcl.init.general]),
- `basic_vec<T, Abi>::mask_type` is an alias for an enabled specialization of `basic_mask`, and
- `basic_vec<T, Abi>::size()` is equal to `basic_vec<T, Abi>::mask_type::size()`.

3.8 MODIFY [SIMD.CREATION]

[simd.creation]

```
template<class T, class... Abis>
  constexpr vec<T, resize_t<(basic_vec<T, Abis>::size() + ...), basic_vec<T, Abis...[0]>>
    cat(const basic_vec<T, Abis>&... xs) noexcept;
template<size_t Bytes, class... Abis>
  constexpr basic_mask<Bytes, deduce-abi-t<integer-from<Bytes>,resize_t<
                            (basic_mask<Bytes, Abis>::size() + ...), basic_mask<Bytes, Abis...[0]>>
    cat(const basic_mask<Bytes, Abis>&... xs) noexcept;
```

6 *Constraints:*

- For the first overload `vec<T, (basic_vec<T, Abis>::size() + ...)>` is enabled.
- For the second overload `basic_mask<Bytes, deduce-abi-t<integer-from<Bytes>, (basic_mask<Bytes, Abis>::size() + ...)>` is enabled.

7 *Returns:* A data-parallel object initialized with the concatenated values in the `xs` pack of data-parallel objects: The i^{th} `basic_vec`/`basic_mask` element of the j^{th} parameter in the `xs` pack is copied to the return value's element with index i + the sum of the width of the first j parameters in the `xs` pack.

3.9 MODIFY [SIMD.MASK.OVERVIEW]

[simd.mask.overview]

```
    constexpr default_sentinel_t cend() const noexcept { return {}; }

    static constexpr integral_constant<simd-size-type, simdmask-size-v<integer-from<Bytes>, Abi>>
      size {};

    constexpr basic_mask() noexcept = default;
```

[...]

1 Every specialization of `basic_mask` is a complete type. The specialization of `basic_mask<Bytes, Abi>` is:

- disabled, if there is no vectorizable type `T` such that `Bytes` is equal to `sizeof(T)`,
- otherwise, enabled, if there exists a vectorizable type `T` and a value `N` in the range $[1, 64]$ such that `Bytes` is equal to `sizeof(T)` and `Abi` is names the ABI tag type denoted by *`deduce-abi-t`*`<T, N>`,
- otherwise, it is implementation-defined if such a specialization is enabled.

If `basic_mask<Bytes, Abi>` is disabled, the specialization has a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. In addition only the `value_type` and `abi_type` members are present.

If `basic_mask<Bytes, Abi>` is enabled, `basic_mask<Bytes, Abi>` is trivially copyable.