Document Number: P0539R5
Date: 2019-12-17
Audience: SG6, LEWGI, LEWG
Revises: P0539R4
Reply to: Igor Klevanets
cerevra@yandex.ru, cerevra@yandex-team.ru
Antony Polukhin
antoshkka@gmail.com, antoshkka@gmail.com

A Proposal to add wide_int Template Class

I. Introduction and Motivation

Current standard provides signed and unsigned `int8_t`, `int16_t`, `int32_t`, `int64_t`. It is usually enough for every day tasks, but sometimes appears a need in big numbers: for cryptography, IPv6, very big counters etc. Non-standard type `__int128` which is provided by GCC and Clang illuminates this need. But there is no cross-platform solution and no way to satisfy future needs in even more big numbers.

This is an attempt to solve the problem in a generic way on a library level and provide wording for P0104R0: Multi-Word Integer Operations and Types.

A proof of concept implementation available at: https://github.com/cerevra/int/tree/master/v3.

II. Changelog

Differences with P0539R4:

- Sync with P1889R1:
  - Use LaTeX for wording and paper
  - Use spaceship operator

Differences with P0539R3:

- More paragraphs in "III. Design and paper limitations"
- Added requirements on `wide_integer` alignment and layout
- Dropped the `explicit` from `operator bool()`
- Permitted implementations to add overloads (mostly for adding explicit overloads that do not generate warnings)

Differences with P0539R2:

- **"Bits" won in the discussion "Words vs Bytes vs Bits" in Albuquerque**
- Changed "MachineWords" to "Bits"
- Interoperability with other types moved to a separate paper

— Removed signedness scoped enum

Differences with P0539R1:

— Added a discussion on "Words vs Bytes vs Bits"

Differences with P0539R0:

— Reworked the proposal for simpler integration with other Numerics proposals:
  — Added an interoperability section [numeric.interop]
  — Arithmetic and Integral concepts were moved to [numeric.requirements] as they seem widely useful
  — Binary non-member operations that accept Arithmetic or Integral parameter were changed to accept two Arithmetic or Integral parameters respectively and moved to [numeric.interop]
  — `int128_t` and other aliases now depend on P0102R0
— Renamed wide_int to `wide_integer`
— `wide_integer` now uses machine words count as a template parameter, not bits count
— Removed not allowed type traits specializations
— Added `to_chars` and `from_chars`

III. Design and paper limitations

`wide_integer` is designed to be as close as possible to built-in integral types:

— it does not allocate memory
— it is a standard layout type
— it is trivially copyable
— it is trivially desctructible
— it is constexpr usable

`wide_integer` is **not** a metafunction. Such metafunctions are discussed in P0102.

`wide_integer` does **not** add `noexcept` to operations that have UB for built-in types. Operations that have UB traditionally are not marked with `noexcept` by LWG. Attempt to change that behavior may be done in separate paper.

In this proposal we concentrate on the `wide_integer` class that uses machine words under the cover. Such implementations allow you to get best performance and leave behind some design questions, like "Is `(sizeof(wide_integer<X>) == X / CHAR_BIT)` or not?".

However, we do not wish to shut the door close for extending the abilities of the `wide_integer` class. Some users may wish to see `unit40_t`, or `unit48_t`.

We insist on **interface that allows specifying integers not representable by machine words count**. Such extensions of functionality may be discussed in separate papers.

`wide_integer` mimics the behavior of `int`. Because of that `wide_int<128> * wide_int<128>` results in `wide_int<128>`, not `wide_int<256>`. There are separate proposals for integers that are elastic (P0828) or safe (P0228R0).

Non template aliases for integers of particular width (for example `int128_t`) are handled in separate paper P0102R0. There was an implicit request from LEWG to remove those aliases from this paper, as they were rising a lot of questions on some platforms.

Interoperability with other arithmetic types was moved to a separate paper P0880.

We double checked that constexpr on default constructor does not require zero initialization in non constexpr contexts and still allows zero initialization if explicitly asked:

```
int main() {
      //constexpr wide_integer<128, unsigned> wi; // Not initialized in constexpr context - compile time error
      wide_integer<128, unsigned> wi_no_init;        // Not initialized - OK
      constexpr wide_integer<128, unsigned> wi{};    // Zero initialized - OK
}
```

Current revision of the paper mimics behavior of int even in cases that are considered dangerous. It is possible to make `wide_integer` type more safe by making explicit:

— Conversion from any signed to any unsigned type.

— Narrowing conversions.

— Conversion to and from floating point types.

— Conversion to `bool`.

Such change will break the compatibility of `wide_integer` and `int`. Consider the case, when you have some template function `foo(Arithmetic a)` that works with arithmetic types. Function is huge and it was developed a long time ago. If interface of `wide_integer` is same as the interface of int then `foo()` could work with it. But if we make some of the conversions explicit, then `foo()` function must be adjusted.

Also note that adding explicit would affect only `wide_integer`s, while there could be a better solution for all the integral types. For example all the integrals could be fixed by some `safe_integer<Integral>` class, that is very explicit and does additional checks on demand. In that case adding restrictions into `wide_integer` would just break the interface compatibility with `int` without big benefit.

IV. Acknowledgements

Many thanks to Alex Strelnikov and John McFarlane for sharing useful ideas and thoughts.

V. Feature-testing macro

For the purposes of SG10 we recommend the feature-testing macro name `__cpp_lib_wide_integer`.

VI. Proposed wording

1 Wide Integers [wide_integer]

1.1 Class template `numeric_limits` [numeric.limits]

Add the following sentence after the sentence "Specializations shall be provided for each arithmetic type, both floating-point and integer, including bool." (first sentence in fourth paragraph in [numeric.limits]):

Specializations shall be also provided for `wide_integer` type.

[*Note:* If there is a built-in integral type `Integral` that has the same signedness and width as `wide_integer<Bits, S>`, then `numeric_limits<wide_integer<Bits, S>>` specialized in the same way as `numeric_limits<Integral>` — *end note*]

1.2 Header `<wide_integer>` synopsis [numeric.wide_integer.syn]

```
#include <compare>

namespace std {

  // 26.??.2 class template wide_integer
  template<size_t Bits, typename S> class wide_integer;

  // 26.??.?? type traits specializations
  template<size_t Bits, typename S, size_t Bits2, typename S2>
    struct common_type<wide_integer<Bits, S>, wide_integer<Bits2, S2>>;

  template<size_t Bits, typename S, typename Arithmetic>
    struct common_type<wide_integer<Bits, S>, Arithmetic>;

  template<typename Arithmetic, size_t Bits, typename S>
    struct common_type<Arithmetic, wide_integer<Bits, S>>
  : common_type<wide_integer<Bits, S>, Arithmetic>
  ;

  // 26.??.?? unary operations
  template<size_t Bits, typename S>
    constexpr wide_integer<Bits, S> operator~(const wide_integer<Bits, S>& val) noexcept;
  template<size_t Bits, typename S>
    constexpr wide_integer<Bits, S> operator-(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);
  template<size_t Bits, typename S>
    constexpr wide_integer<Bits, S> operator+(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);

  // 26.??.?? binary operations
  template<size_t Bits, typename S, size_t Bits2, typename S2>
  common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
    constexpr operator*(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

  template<size_t Bits, typename S, size_t Bits2, typename S2>
  common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
    constexpr operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

  template<size_t Bits, typename S, size_t Bits2, typename S2>
  common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
```

```
  constexpr operator+(const wide_integer<Bits, S>& lhs,
                      const wide_integer<Bits2, S2>& rhs) noexcept(is_unsigned_v<S>);

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator-(const wide_integer<Bits, S>& lhs,
                      const wide_integer<Bits2, S2>& rhs) noexcept(is_unsigned_v<S>);

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator&(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator|(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr  operator^(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
  constexpr  operator<<(const wide_integer<Bits, S>& lhs, size_t rhs);

template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
  constexpr  operator>>(const wide_integer<Bits, S>& lhs, size_t rhs);

template<size_t Bits, typename S, size_t Bits2, typename S2>
  constexpr bool operator==(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
  constexpr strong_ordering operator<=>(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) n

// 26.??.?? numeric conversions
template<size_t Bits, typename S> std::string to_string(const wide_integer<Bits, S>& val);
template<size_t Bits, typename S> std::wstring to_wstring(const wide_integer<Bits, S>& val);

// 26.??.?? iostream specializations
template<class Char, class Traits, size_t Bits, typename S>
  basic_ostream<Char, Traits>& operator<<(basic_ostream<Char, Traits>& os,
                                          const wide_integer<Bits, S>& val);

template<class Char, class Traits, size_t Bits, typename S>
  basic_istream<Char, Traits>& operator>>(basic_istream<Char, Traits>& is,
                                          wide_integer<Bits, S>& val) noexcept;

// 26.??.?? hash support
template<class T> struct hash;
template<size_t Bits, typename S> struct hash<wide_integer<Bits, S>>;
```

```
  template <size_t Bits, typename S>
    to_chars_result to_chars(char* first, char* last, const wide_integer<Bits, S>& value,
                             int base = 10);

  template <size_t Bits, typename S>
    from_chars_result from_chars(const char* first, const char* last, wide_integer<Bits, S>& value,
                                 int base = 10);

  template <size_t Bits>
  using wide_int = wide_integer<Bits, signed>;

  template <size_t Bits>
  using wide_uint = wide_integer<Bits, unsigned>

  // optional literals
  inline namespace literals {
  inline namespace wide_int_literals {

  constexpr wide_int<128> operator "" _int128(const char*);
  constexpr wide_int<256> operator "" _int256(const char*);
  constexpr wide_int<512> operator "" _int512(const char*);
  constexpr wide_uint<128> operator "" _uint128(const char*);
  constexpr wide_uint<256> operator "" _uint256(const char*);
  constexpr wide_uint<512> operator "" _uint512(const char*);

  } // namespace wide_int_literals
  } // namespace literals

} // namespace std
```

The header <wide_integer> defines class template wide_integer and a set of operators for representing and manipulating integers of specified width.

Example:

```
using int128_t = wide_int<128>;
constexpr int128_t c = std::numeric_limits<int128_t>::min();
static_assert(c == 0x80000000000000000000000000000000_uint128);

int256_t a = 13;
a += 0xFF;
a *= 2.0;
a -= 12_int128;
assert(a > 0);
```

1.3 Template class wide_integer overview [numeric.wide_integer.overview]

```
namespace std {
  template<size_t Bits, typename S>
  class wide_integer {
  public:
    // 26.??.2.?? construct:
    constexpr wide_integer() noexcept = default;
    constexpr wide_integer(const wide_integer<Bits, S>& ) noexcept = default;
    template<typename Arithmetic> constexpr wide_integer(const Arithmetic& other) noexcept;
    template<size_t Bits2, typename S2> constexpr wide_integer(const wide_integer<Bits2, S2>& other) noexcept;
```

```
// 26.??.2.?? assignment:
constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits, S>& ) noexcept = default;
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator=(const Arithmetic& other) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits2, S2>& other) noexcept;

// 26.??.2.?? compound assignment:
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator*=(const Arithmetic&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator*=(const wide_integer<Bits2, S2>&);

template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator/=(const Arithmetic&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator/=(const wide_integer<Bits2, S2>&);

template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator+=(const Arithmetic&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator+=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);

template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator-=(const Arithmetic&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator-=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);

template<typename Integral>
  constexpr wide_integer<Bits, S>& operator%=(const Integral&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator%=(const wide_integer<Bits2, S2>&);

template<typename Integral>
  constexpr wide_integer<Bits, S>& operator&=(const Integral&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator&=(const wide_integer<Bits2, S2>&) noexcept;

template<typename Integral>
  constexpr wide_integer<Bits, S>& operator|=(const Integral&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&) noexcept;

template<typename Integral>
  constexpr wide_integer<Bits, S>& operator^=(const Integral&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator^=(const wide_integer<Bits2, S2>&) noexcept;

template<typename Integral>
  constexpr wide_integer<Bits, S>& operator<<=(const Integral&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator<<=(const wide_integer<Bits2, S2>&);

template<typename Integral>
```

```
      constexpr wide_integer<Bits, S>& operator>>=(const Integral&) noexcept;
    template<size_t Bits2, typename S2>
      constexpr wide_integer<Bits, S>& operator>>=(const wide_integer<Bits2, S2>&) noexcept;

    constexpr wide_integer<Bits, S>& operator++() noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S> operator++(int) noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S>& operator--() noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S> operator--(int) noexcept(is_unsigned_v<S>);

    // 26.??.2.?? observers:
    template <typename Arithmetic> constexpr operator Arithmetic() const noexcept;
      constexpr explicit operator bool() const noexcept;
  private:
    byte data[Bits / CHAR_BIT]; // exposition only
  };
} // namespace std
```

The class template wide_integer<size_t Bits, typename S> is a trivial standard layout class that behaves as an integer type of a compile time specified bitness.

Template parameter Bits specifies exact bits count to store the integer value. `Bits % (sizeof(int) * CHAR_BIT)` is equal to `0`. `sizeof(wide_integer<Bits, unsigned>)` and `sizeof(wide_integer<Bits, signed>)` are required to be equal to `Bits * CHAR_BIT`.

When size of wide_integer is equal to a size of builtin integral type then the alignment and layout of that `wide_integer` is equal to the alignment and layout of the builtin type.

Template parameter S specifies signedness of the stored integer value and is either signed or unsigned.

Implementations are permitted to add explicit conversion operators and explicit or implicit constructors for Arithmetic and for Integral types.

Example:

```
template <class Arithmetic>
[[deprecated("Implicit conversions to builtin arithmetic types are not safe!")]]
  constexpr operator Arithmetic() const noexcept;

explicit constexpr operator bool() const noexcept;
explicit constexpr operator int() const noexcept;
...
```

1.3.1 `wide_integer` construsctors [numeric.wide_integer.cons]

```
constexpr wide_integer() noexcept = default;
```

Effects: Constructs an object with undefined value.

```
template<typename Arithmetic>
  constexpr wide_integer(const Arithmetic& other) noexcept;
```

Effects: Constructs an object from `other` using the integral conversion rules [conv.integral].

```
template<size_t Bits2, typename S2>
  constexpr wide_integer(const wide_integer<Bits2, S2>& other) noexcept;
```

Effects: Constructs an object from `other` using the integral conversion rules [conv.integral].

1.3.2 `wide_integer` assignments [numeric.wide_integer.assign]

```
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator=(const Arithmetic& other) noexcept;
```

Effects: Constructs an object from `other` using the integral conversion rules [conv.integral].

```
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits2, S2>& other) noexcept;
```

Effects: Constructs an object from `other` using the integral conversion rules [conv.integral].

1.3.3 `wide_integer` compound assignments [numeric.wide_integer.cassign]

```
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator*=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator/=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator+=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator-=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator%=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator&=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator^=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator<<=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator>>=(const wide_integer<Bits2, S2>&) noexcept;
constexpr wide_integer<Bits, S>& operator++() noexcept(is_unsigned_v<S>);
constexpr wide_integer<Bits, S> operator++(int) noexcept(is_unsigned_v<S>);
constexpr wide_integer<Bits, S>& operator--() noexcept(is_unsigned_v<S>);
constexpr wide_integer<Bits, S> operator--(int) noexcept(is_unsigned_v<S>);
```

Effects: Behavior of the above operators is similar to operators for built-in integral types.

```
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator*=(const Arithmetic&);
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator/=(const Arithmetic&);
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator+=(const Arithmetic&) noexcept(is_unsigned_v<S>);
template<typename Arithmetic>
  constexpr wide_integer<Bits, S>& operator-=(const Arithmetic&) noexcept(is_unsigned_v<S>);
```

Effects: As if an object `wi` of type `wide_integer<Bits, S>` was created from input value and the corresponding operator was called for `*this` and the `wi`.

```
template<typename Integral>
  constexpr wide_integer<Bits, S>& operator%=(const Integral&);
template<typename Integral>
  constexpr wide_integer<Bits, S>& operator&=(const Integral&) noexcept;
template<typename Integral>
```

```
  constexpr wide_integer<Bits, S>& operator|=(const Integral&) noexcept;
template<typename Integral>
  constexpr wide_integer<Bits, S>& operator^=(const Integral&) noexcept;
template<typename Integral>
  constexpr wide_integer<Bits, S>& operator<<=(const Integral&);
template<typename Integral>
  constexpr wide_integer<Bits, S>& operator>>=(const Integral&) noexcept;
```

Effects: As if an object `wi` of type `wide_integer<Bits, S>` was created from input value and the corresponding operator was called for `*this` and the `wi`.

1.3.4 `wide_integer` observers [numeric.wide_integer.observers]

```
template <typename Arithmetic> constexpr operator Arithmetic() const noexcept;
```

Returns: If `is_integral_v<Arithmetic>` then `Arithmetic` is constructed from `*this` using the integral conversion rules [conv.integral]. If `is_floating_point_v<Arithmetic>`, then `Arithmetic` is constructed from `*this` using the floating-integral conversion rules [conv.fpint]. Otherwise the operator shall not participate in overload resolution.

1.4 Specializations of common_type [numeric.wide_integer.traits.specializations]

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
struct common_type<wide_integer<Bits, S>, wide_integer<Bits2, S2>> {
    using type = wide_integer<max(Bits, Bits2), see below>;
};
```

The signed template parameter indicated by this specialization is following:

- `(is_signed_v<S> && is_signed_v<S2> ? signed : unsigned)` if `Bits == Bits2`
- `S` if `Bits > Bits2`
- `S2` otherwise

[Note: common_type follows the usual arithmetic conversions design. - end note]

[Note: common_type attempts to follow the usual arithmetic conversions design here for interoperability between different numeric types. Following two specializations must be moved to a more generic place and enriched with usual arithmetic conversion rules for all the other numeric classes that specialize `std::numeric_limits`- end note]

```
template<size_t Bits, typename S, typename Arithmetic>
struct common_type<wide_integer<Bits, S>, Arithmetic> {
    using type = see below;
};

template<typename Arithmetic, size_t Bits, typename S>
struct common_type<Arithmetic, wide_integer<Bits, S>>
: common_type<wide_integer<Bits, S>, Arithmetic>;
```

The member typedef type is following:

- `Arithmetic` if `numeric_limits<Arithmetic>::is_integer` is false
- `wide_integer<Bits, S>` if `sizeof(wide_integer<Bits, S>) > sizeof(Arithmetic)`
- `Arithmetic` if `sizeof(wide_integer<Bits, S>) < sizeof(Arithmetic)`
- `Arithmetic` if `sizeof(wide_integer<Bits, S>) == sizeof(Arithmetic) && is_signed_v<S>`

— `Arithmetic` if `sizeof(wide_integer<Bits, S>) == sizeof(Arithmetic) && numeric_limits<wide_integer<Bits, S>>::is_signed == numeric_limits<Arithmetic>::is_signed`

— `wide_integer<Bits, S>` otherwise

1.5 Unary operators [numeric.wide_integer.unary_ops]

```
template<size_t Bits, typename S>
  constexpr wide_integer<Bits, S> operator~(const wide_integer<Bits, S>& val) noexcept;
```

Returns: value with inverted significant bits of `val`.

```
template<size_t Bits, typename S>
  constexpr wide_integer<Bits, S> operator-(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);
```

Returns: `val *= -1` if `S` is `true`, otherwise the result is unspecified.

```
template<size_t Bits, typename S>
  constexpr wide_integer<Bits, S> operator+(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);
```

Returns: `val`.

1.6 Binary operators [numeric.wide_integer.binary_ops]

In the function descriptions that follow, `CT` represents `common_type_t<A, B>`, where `A` and `B` are the types of the two arguments to the function.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
  common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator*(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
```

Returns: `CT(lhs) *= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
```

Returns: `CT(lhs) /= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator+(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs)
                      noexcept(is_unsigned_v<S>);
```

Returns: `CT(lhs) += rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator-(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs)
                      noexcept(is_unsigned_v<S>);
```

Returns: `CT(lhs) -= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
```

Returns: `CT(lhs) %= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator&(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: `CT(lhs) &= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr operator|(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: `CT(lhs) |= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
  constexpr  operator^(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: `CT(lhs) ^= rhs`.

```
template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
  constexpr operator<<(const wide_integer<Bits, S>& lhs, size_t rhs);
```

Returns: `CT(lhs) <<= rhs`.

```
template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
  constexpr operator>>(const wide_integer<Bits, S>& lhs, size_t rhs) noexcept;
```

Returns: `CT(lhs) >>= rhs`.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
  constexpr bool operator==(const wide_integer<Bits, S>& lhs,
                            const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: `true` if significant bits of `CT(lhs)` and `CT(rhs)` are the same.

```
template<size_t Bits, typename S, size_t Bits2, typename S2>
  constexpr strong_ordering operator<=>(const wide_integer<Bits, S>& lhs,
                                        const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: `strong_ordering::equal` if `CT(lhs) - CT(rhs) == 0`; otherwise returns `strong_ordering::less` if `CT(lhs) - CT(rhs) < 0`; otherwise `strong_ordering::greater`.

1.7 Numeric conversions [numeric.wide_integer.conversions]

```
template<size_t Bits, typename S> std::string to_string(const wide_integer<Bits, S>& val);
template<size_t Bits, typename S> std::wstring to_wstring(const wide_integer<Bits, S>& val);
```

Returns: Each function returns an object holding the character representation of the value of its argument. All the significant bits of the argument are outputed as a signed decimal in the style [-]dddd.

```
template <size_t Bits, typename S>
  to_chars_result to_chars(char* first, char* last, const wide_integer<Bits, S>& value,
                           int base = 10);
```

Behavior of `wide_integer` overload is subject to the usual rules of primitive numeric output conversion functions [utility.to.chars].

```
template <size_t Bits, typename S>
  from_chars_result from_chars(const char* first, const char* last, wide_integer<Bits, S>& value,
                               int base = 10);
```

Behavior of `wide_integer` overload is subject to the usual rules of primitive numeric input conversion functions [utility.from.chars].

1.8 iostream specializations [numeric.wide_integer.io]

```
template<class Char, class Traits, size_t Bits, typename S>
  basic_ostream<Char, Traits>& operator<<(basic_ostream<Char, Traits>& os,
                                          const wide_integer<Bits, S>& val);
```

1 *Effects:* As if by: `os << to_string(val)`.

2 *Returns:* `os`.

```
template<class Char, class Traits, size_t Bits, typename S>
  basic_istream<Char, Traits>& operator>>(basic_istream<Char, Traits>& is,
                                          wide_integer<Bits, S>& val);
```

3 *Effects:* Extracts a `wide_integer` that is represented as a decimal number in the `is`. If bad input is encountered, calls `is.setstate(ios_base::failbit)` (which may throw `ios::failure` ([iostate.flags])).

4 *Returns:* `is`.

1.9 Hash support [numeric.wide_integer.hash]

```
template<size_t Bits, typename S> struct hash<wide_integer<Bits, S>>;
```

The specialization is enabled (20.14.14). If there is a built-in integral type `Integral` that has the same typename and width as `wide_integer<Bits, S>`, and `wi` is an object of type `wide_integer<Bits, S>`, then `hash<wide_integer<MachineWords, S>>()(wi) == hash<Integral>()(Integral(wi))`.