A Proposal to add Single Instruction Multiple Data Computation to the Standard Library

Pierre Estérie[1], Mathias Gaunard[2] and Joel Falcou[1]

[1]LRI, Université Paris-Sud XI
[2]Metascale

Document number: N3571=13-3571
Date: 2013-03-15
Project: Programming Language C++
Reply-to: Pierre Estérie <pierre.esterie@lri.fr>,
Mathias Gaunard <mathias.gaunard@metascale.org>,
Joel Falcou <joel.falcou@lri.fr>

Contents

1 Introduction

Since the late 90s, processor manufacturers provide specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. For example, when working with SSE2, these registers are 128 bits wide and can hold 4 values corresponding to any 32 bits type. While some architectures provide floating-point functionality only with a SIMD unit, some also provide a scalar floating point unit, like the x87 FPU on the x86 architecture. This can lead to different behaviour between scalar and SIMD code, since those two units may not operate with the same precision. Moving scalar to SIMD code can therefore result in a numerical deviation and direclty impact the results of an application in the case of floating point. In the case of integer computations, architectures always have separate scalar and SIMD units, assuming SIMD integer operations are supported.

With a constantly increasing need for performance in applications, today's processor architectures offer rich SIMD instruction sets working with larger and larger SIMD registers (table 1). For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer micro-architectures by providing a distinct set of sixteen 256-bit registers. Similary, the Intel MIC Architecture (also known as Xeon Phi) is embedded 512-bit SIMD registers. Usage of SIMD processing units can also be mandatory for performance on embedded systems as demonstrated by the NEON and NEON AArch64 ARM extensions [**?**] or the CELL-BE processor by IBM [**?**] which SPUs were designed as a SIMD-only system.

In the following section the current limitations to program such extensions will be described and our proposal introduced.

2 Motivation and Scope

Nowadays, software efficiency has become one of the main concerns since the hardware has reached the era of parallelism. Reaching the power of new processor architectures with SIMD units is still a hard task, with several choices available to the programmer.

Table 1: SIMD extensions in modern processors

Manufacturer	Extension	Registers size & nbr	Instructions
Intel	SSE	128 bits - 8	70
	SSE2	128 bits - 8/16	214
	SSE3	128 bits - 8/16	227
	SSSE3	128 bits - 8/16	227
	SSE4.1	128 bits - 8/16	274
	SSE4.2	128 bits - 8/16	281
	AVX	256 bits - 8/16	292
AMD	SSE4a	128 bits - 8/16	231
Motorola IBM	VMX	128 - 32	114
	VMX128	128 bits - 128	
	VSX	128 bits - 64	
ARM	NEON	128 bits - 16	100+

2.1 The current solutions

Using SIMD computation units requires using the dedicated instructions and registers associated to the the target extension. Writing assembly code is the most direct solution to use these instructions and ends up with vectorized code (i.e. code that use SIMD instructions). This task remains error-prone and time-consuming. With different manufacturers, the instruction sets change and between each architecture improvement some new instructions can be added. The previous statement impacts the portability and the maintainability of the application.

Compilers are now able to generate SIMD code through their embedded autovectorizers. This allows the programmer to keep a standard code that will be analyzed and transformed to a vectorized code during the code generation process. Autovectorizers have the ability to detect code fragments that can be vectorized. This automatic process finds its limits when the user code is not presenting a clear vectorizable pattern (i.e. complex data dependencies, non-contiguous memory accesses, aliasing or control flows). The SIMD code generation stays fragile and the resulting instruction flow may be suboptimal compared to an explicit vectorization. Still on the compiler side,

code directives can be used to enforce loop vectorisation (`#pragma simd` for ICC and GCC) but the code quality relies on the compiler and this feature is not available in every one of them. Dedicated compilers like ISPC or Cilk choose to add a set of keywords to the language to explicitly mark the code fragments that are candidates to the automatic vectorization process. The user code becomes non-standard and strongly dependent to specific compiler techniques. Most of these approaches also rely on generating SIMD code from scalar code, disregarding the specificities of each of these computing units.

The most common way to take advantage of a SIMD extension is to write calls to intrinsics. These low level C functions represent each SIMD instruction supported by the hardware, and while being similar to programming with assembly it is definitely more accessible and optimization-friendly. With a lot of variants to handle all SIMD register types, the set of intrinsics usually only covers functionality for which there is a dedicated instruction, often lacking orthogonality or missing more complex operations like trigonometric or exponential functions. Due to its C interface, using intrinsics forces the programmer to deal with a verbose style of programming. Furthermore, from one extension to another, the Application Programming Interface differs and the code needs to be written again due to hardware specific functionalities and optimizations.

Finally, some compilers, mainly GCC and Clang, have introduced extensions for arbitrary-sized SIMD vectors using the vector_size attribute. These however suffer from limited functionality and bad interoperability with C++. It is the intent of this proposal to propose something similar for C++ but that can also be purely implemented as a library.

2.2 Our proposal

For maximum accessibility, programmers should be able to vectorize their code without needing a high level of expertise for every single SIMD extension. This proposal introduces a high-level abstraction to the user that gives access to SIMD computation in an instinctive way. It comes as a C++ template library, headers only that relies on a possibly full library implementation. With a high level template type for abstracting a SIMD register, the user can easily introduce SIMD in his application by instantiating this type and applying high level functions on it. Working at the register level rather than the loop nest or big array level keeps the abstraction thin, efficient and

flexible. By keeping the control of the vectorization process, the programmer is explicitly expressing the SIMD version of his algorithm, not only guaranteeing that vectorization does indeed take place, but also empowering the user to define his algorithm in a way that is vectorizable. A single generic code can be written for both the scalar and SIMD types or different code paths may be selected. The library is also modular and easily extensible by the user.

In addition to types and functions operating on them, higher-order functions to manipulate and transform data with respect to every hardware constraints are provided.

Furthermore, processing multiple data in SIMD registers breaks typical scalar dataflows when dealing with branching conditions or when shifting or shuffling values. As a result, special functions to deal with SIMD-specific idioms are also introduced.

The idea of this proposal is inspired from the Boost.SIMD open-source library (not part of the Boost C++ libraries as of this writing) developed by the authors of this paper. This library has been deployed in several academic and industrial projects where it has shown significant advantages over other approaches to optimize code for SIMD-enabled processors. Boost.SIMD is available as part of the NT^2 software project hosted on GitHub [**?**]. Publications with experimental results are available in [**?**] and [**?**]. All design decisions and implementation details are discussed in section 4 and 5 respectively.

3 Impact On the Standard

This proposal comes as a library extension that does not impact existing standard classes, functions or headers. This addition is non-intrusive; its implementation is fully standards-based and does not require any changes to the core language.

4 Design Decisions

In this section, we detail the major design of the proposed library extension by introducing the required additions to the standard.

4.1 SIMD Register Abstraction : pack<T,N>

The `pack` class is abstracting the SIMD register type. It respects the Random Access Sequence Concept and provides a tuple-like interface. For a given type `T` and a given static integral value `N` (`N` being a small power of 2), a `pack` encapsulates the best type able to store a sequence of `N` elements of type `T`. when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used. For arbitrary `T` and `N`, this type is emulating a SIMD register or aggregating several of them. This semantic provides a way to use arbitrarily large SIMD registers on any system and let the library select the best vectorizable type to handle them. By default, if `N` is not provided, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. Moreover, by carrying informations about its underlying scalar type, `pack` enables proper instruction selection even when used on extensions (like SSE2 and above) that map all integral type to a single SIMD type (`__m128i` for SSE2).

`pack` handles these low-level SIMD register types as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value or list of scalar values. In each case, the proper register loading strategy will be issued. `pack` also takes care of issues like boolean predicates support, more details are available in section 4.3.1.

4.2 Operators and Functions

The `pack` class comes with support for C++ operators:

- Arithmetic
- Comparison/relational
- Logical
- Bitwise
- Compound assignment

Operators on `pack` behave like the scalar operators on every elements of the `pack`. When an operator is requested between a `pack` and a scalar value, the behavior is equivalent to the single value being splatted in a `pack` and the operation being performed element-wise.

In addition, `pack` is completed by a set of functions that provide a simple way to interact with memory, since the hardware design of SIMD processing units introduces memory constraints such as alignment. Performance is guaranteed by accessing to the memory through dedicated intrinsics that perform register-length aligned memory accesses. The `load`/`store` functions perform the explicit memory operations and their unaligned versions (`unaligned_load`/`unaligned_store`) add flexibility to interact with unaligned data that can be held in a memory segment unsuitable for aligned access. When a scalar value needs to be repeated in a vector The `splat` function allows to repeat a scalar value in a SIMD register.

```
int main()
{
  float s;
  std::vector< float, simd::allocator<float> > v =
     {1,-2,3,-4};
  // Build pack from initialization list
  pack<float> x{1,2,3,4};
  // Usage of the load function
  pack<float> b = std::simd::load< std::simd::pack<float> >(&
     v[0]);
  // Built pack from a scalar value (splatted in register)
  pack<float> a(1.37);
  // pack default construct
  pack<float> r;
```

```
  // Operator and function calls
  r += std::simd::min(a*x+b,b);
  // RAS interface and using std::accumulate
  r[0] = 1.f + r[0];
  s = accumulate(r.begin(), r.end(), 0.f);
  return 0;
}
```

Listing 1: Working with `pack`

Classical operators and memory functions enable a clear expressiveness for the vectorized code and the algorithm is not buried in architecture-specific details.

4.3 SIMD Idioms

4.3.1 Branching Condition

Comparisons between SIMD vectors yield a vector of boolean results. While most SIMD extensions store a $0\backslash\sim 0$ bitmask in the same register type as the one used in the comparison, some, like Intel MIC, have a special register bank for those types. To handle architecture-specific predicates, an abstraction over boolean values and a set of associated operations needs to be given to the user. The `logical` class encapsulates the notion of a boolean value and can be combined with `pack`. Thus, for any type `T`, an instance of `pack< logical<T> >` encapsulates the proper SIMD register type able to store boolean values resulting from the application of a SIMD predicate over a `pack<T>`. Thus, the comparison operators will return a `pack<logical<T> >`. The branching is performed by a dedicated function `if_else` that is able to vectorise the branching process according to the target architecture.

Unlike scalar branching, SIMD branching does not do lazy evaluation. All branches of an algorithm are evaluated before the result is selected.

4.3.2 Masking

Working with registers can imply bit masking between two `pack` even if the types stored in the two registers differ. The only requirement is two `pack` with the same cardinal `N`. When masking is performed between floating and integral values, the operation stays valid.

4.3.3 Shuffling

A typical use case in SIMD is when the user wants to rearrange the data stored in `pack`. This operation in called *shuffling* the register. According to the cardinal of a `pack`, several permutations can be achieved between the data. To handle this, we introduce the `shuffle` function. This function accepts a metafunction class that will take as a parameter the destination index in the result register and return the correct index corresponding to the value from the source register. A second version version of the function is also available and allows the user to directly specify the indexes as template parameters. Listing 4.4.1 shows examples related to this function.

```
// A metafunction that reverses the register
struct reverse_
{
  template<class Index, class Cardinal>
  struct apply
       : std::integral_constant<int, Cardinal::value - Index
          ::value - 1> {};
};
[...]
std::simd::pack<int,4> r{11,22,3,4};
r1 = std::simd::shuffle<reverse_>(r);
r2 = std::simd::shuffle<3,2,1,0>(r);
assert(std::simd::all(std::simd::pack<int,4>{4,3,22,11} == r1
   ));
assert(std::simd::all(std::simd::pack<int,4>{4,3,22,11} == r2
   ));
```

Listing 2: `shuffle` example

4.3.4 Reduction

Intra-register operations often occur when working at the SIMD register level. For example, performing the sum of all the values stored in a register is a common use case. Our proposal covers this use case by providing four functions :

- `sum` accumulates all the values stored in a register;
- `prod` returns the product of all the elements of a register;
- `any` returns true if at least one element of the input vector is non zero;

- `all` returns true if all elements of the input vector are non zero.
- `none` returns true if no elements of the input vector are non zero.

4.4 Standard Components

4.4.1 SIMD Allocator

The hardware implementation of SIMD processing units introduces constraints related to memory handling as seen in section 4.2. This constraint may be addressed by special memory allocation strategies via OS and compiler-specific function calls (for example `aligned_malloc` on POSIX systems).

Our proposal aims to provide two STL compliant allocators dealing with this kind of alignment requirements. The first one, `simd::allocator`, wraps these OS and compiler functions in a simple STL-compliant allocator. When an existing allocator defines a specific memory allocation strategy, the user can adapt it to handle alignment by wrapping it in `simd::allocator_adaptor`. The adaptation is realized through a pointer stashing technique.

```
namespace std { namespace simd
{
  template<class T>
  struct allocator
  {
    typedef T               value_type;
    typedef T*              pointer;
    typedef T const*        const_pointer;
    typedef T&              reference;
    typedef T const&        const_reference;
    typedef size_t          size_type;
    typedef ptrdiff_t       difference_type;

    template<class U>
    struct rebind
    {
      typedef allocator<U> other;
    };

    allocator();

    template<class U>
    allocator(allocator<U> const&);
```

```
  pointer       address(reference r);
  const_pointer address(const_reference r);

  size_type max_size() const;

  void construct(pointer p, const T& t);
  void destroy(pointer p);

  pointer allocate( size_type c, const void* = 0 );
  void deallocate (pointer p, size_type s);
};

template<class T>
bool operator==(allocator<T> const&, allocator<T> const&);

template<class T>
bool operator!=(allocator<T> const&, allocator<T> const&);

template<class Allocator>
struct allocator_adaptor
{
  typedef Allocator base_type;

  typedef typename base_type::value_type       value_type;
  typedef typename base_type::pointer          pointer;
  typedef typename base_type::const_pointer
     const_pointer;
  typedef typename base_type::reference        reference;
  typedef typename base_type::const_reference
     const_reference;
  typedef typename base_type::size_type        size_type;
  typedef typename base_type::difference_type
     difference_type;

  template<class U>
  struct rebind
  {
    typedef allocator_adaptor<typename Allocator::rebind<U
       >::other> other;
  };

  allocator_adaptor();
  allocator_adaptor(Allocator const& alloc);
  ~allocator_adaptor();
```

```
    template<class U>
    allocator_adaptor(allocator_adaptor<U> const& src);

    base_type&        base();
    base_type const&  base() const;

    pointer allocate( size_type c, const void* = 0 );
    void deallocate (pointer p, size_type s);
  };

  template<class T>
  bool operator==(allocator_adaptor<T> const& a,
     allocator_adaptor<T> const& b);

  template<class T>
  bool operator!=(allocator_adaptor<T> const& a,
     allocator_adaptor<T> const& b);
} }
```

4.4.2 SIMD Algorithms

Applying the `transform` or `accumulate` algorithms with SIMD requires not only that the data be well-aligned, but also that the data size be an exact multiple of the SIMD register width. Additionally, in the case of accumulate, additional operations at the end of the call to accumulate the register itself are required. To alleviate these limitations, our proposal introduces variants of the `transform` and `accumulate` algorithms that take care of the potential unaligned or trailing data, with the difference that the provided function object must be defined for both scalar and SIMD types.

```
std::vector<int> v(100), r(100);
simd::transform ( v.data()
                , v.data() + v.size()
                , r.data()
                , [](auto p){ return -p; }
                );
```

Listing 3: Iterators with SIMD algorithm

SIMD-aware algorithms are therefore a particularly good use case of polymorphic lambas.

```
namespace std { namespace simd
{
  template<class T, class U, class UnOp>
```

```
  U* transform(T const* begin, T const* end, U* out, UnOp f);

  template<class T1, class T2, class U, class BinOp>
  U* transform(T1 const* begin1, T1 const* end, T2 const*
     begin2, U* out, BinOp f);

  template<class T, class U, class F>
  U accumulate(T const* begin, T const* end, U init, F f);
} }

template<class T, class U, class UnOp>
U* transform(T const* begin, T const* end, U* out, UnOp f);
```

Requires: `UnOp` is `Callable<U(T)>` and `Callable<pack<U>(pack<T>)>`

Effects: Writes to out the result of the application of `f` for each element in the range [`begin`, `end`[, by either loading scalar or SIMD values from the range if sufficient aligned data is available.

Returns: The iterator past the last written-to position

```
template<class T1, class T2, class U, class BinOp>
U* transform(T1 const* begin1, T1 const* end, T2 const*
   begin2, U* out, BinOp f);
```

Requires: `BinOp` is `Callable<U(T1, T2)>` and `Callable<pack<U>(pack<T1>, pack<T2>)>`

Effects: Writes to out the result of the application of `f` for each pair of elements in the ranges [`begin1`, `end`[and [`begin2`, `begin2+(end-begin1)`[, by either loading scalar or SIMD values from the ranges if sufficient aligned data is available.

Returns: The iterator past the last written-to position

```
template<class T, class U, class F>
U accumulate(T const* begin, T const* end, U init, F f);
```

Requires: `F` is `Callable<U(U, T)>` and `Callable<pack<U>(pack<U>, pack<T>)>`

Effects: Accumulate the result of the application of `f` with the accumulation state and each element of the range [`begin`, `end`[, potentially scalar by scalar of SIMD vector by vector if sufficient aligned data is available.

Returns: The final accumulation state

Non-normative Note: possible implementation of binary transform

```
template<class T1, class T2, class U, class BinOp>
U* transform(T1 const* begin1, T1 const* end, T2 const*
    begin2, U* out, BinOp f)
{
  // vectorization step based on ideal for output type
  typedef pack<U> vU;
  static const size_t N = vU::static_size;

  typedef pack<T1, N> vT1;
  typedef pack<T2, N> vT2;

  std::size_t align = N*sizeof(U);
  std::size_t shift = reinterpret_cast<U*>((reinterpret_cast<
     uinptr_t>(out)+align-1) & ~(align-1)) - out;
  T1 const* end2 = begin1 + shift;
  T1 const* end3 = end2 + (end - end2)/N*N;

  // prologue until 'out' aligned
  for(; begin1!=end2; ++begin1, ++begin2, ++out)
    *out = f(*begin1, *begin2);

  // vectorized body while more than N elements
  for(; begin1!=end3; begin1 += N, begin2 += N, out += N)
    simd::store(f(simd::unaligned_load<vT1>(begin1), simd::
       unaligned_load<vT2>(begin2)), out);

  // epilogue for remaining elements
  for(; begin1!=end; ++begin1, ++begin2, ++out)
    *out = f(*begin1, *begin2);

  return out;
}
```

4.5 Implementation Considerations

4.5.1 Intrinsics or built-ins

To be able to generate SIMD instructions as intended, this library requires compiler-specific support. Compiler-specific implementations of the proposal can use different techniques, such as attributes, built-ins or autovectorization

directives, but should lead to the SIMD instructions matching the operation The most straightforward technique is however to work directly with platform-specific intrinsics as standardized by hardware manufacturers, since those are portable across compilers and ensure the instantiation of the instruction required.

4.5.2 ABI and inlining

The main issue with implementing this library efficiently is tied to how the compiler will handle the `pack` type, in particular how the ABI defines that objects of these types be passed to functions. Indeed, since `pack` is defined as a struct, many ABIs (with the notable exception of Intel x86-64 on Linux) will be unable to pass that structure directly in registers. Certain ABIs will also reject passing those types by value due to the alignment requirement being often higher than that of the stack.

As a result "stack dance" – the unnecessary writing and reading of SIMD register contents to stack memory – might occur whenever a non-inlined function is called. A possible way to solve this problem is to force a wrapper function to be inlined and make it call a function that uses the native type of the platform to be more friendly with the ABI.

Using the native type directly is not generally possible since is usually lacks the desired typing information.

4.5.3 Aliasing

The interface of `pack` requires being able to read and write to the register with *operator[]*. A possible way to achieve this is to cast a pointer to the SIMD register to a pointer to the first element of an array of scalar objects, but this may require specific compiler extensions to be compatible with the strict aliasing rules.

5 Technical Specifications

Here, we present the public interface required for the proposal.

5.1 `pack<T,N>` class

```
namespace std { namespace simd
{
  template<class T, std::size_t N = unspecified >
  struct alignas(sizeof(T)*N) pack
  {
    typedef T                    value_type;
    typedef value_type&          reference;
    typedef value_type const&    const_reference;
    typedef T*                   iterator;
    typedef T const*             const_iterator;

    static const size_t static_size = N;

    // does not initialize values of pack
    pack();

    // copy constructor
    pack(pack const& p);

    // splat t N times into pack
    pack(T t);

    // fill pack with values from init
    template<class T>
    pack(initializer_list<T> init);

    reference         operator[](std::size_t i);
    const_reference   operator[](std::size_t i) const;
    iterator          begin();
    const_iterator    begin() const;
    iterator          end();
    const_iterator    end() const;
    std::size_t       size() const;
    bool              empty() const;
  };
} }
```

5.2 logical<T> class

```
template<typename T>
struct logical;
```

Listing 4: The `logical` structure

`logical` is a marker for packs of boolean results and cannot be used in scalar mode.

5.3 Operators overload for `pack<T,N>`

5.3.1 Unary Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator+(pack<T,N> p);

  template<class T, std::size_t N>
  pack<T,N> operator-(pack<T,N> p);

  template<class T, std::size_t N>
  typename as\_logical< pack<T,N> >::type operator!(pack<T,N>
      p);

  template<class T, std::size_t N>
  pack<T,N> operator~(pack<T,N> p);
} }
```

```
template<class T, std::size_t N>
pack<T,N> operator+(pack<T,N> p);
```

Requires: `T` is not a `logical` type.

Effects: Apply unary `operator+` on every element of `p`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = + p[i]`

```
template<class T, std::size_t N>
pack<T,N> operator-(pack<T,N> p);
```

Requires: `T` is not a `logical` type.

Effects: Apply unary `operator-` on every element of `p`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = - p[i]`

```
template<class T, std::size_t N>
typename as_logical< pack<T,N> >::type operator!(pack<T,N> p);
```

Effects: Apply unary `operator!` on every element of `p`

Returns: A `as_logical< pack<T,N> >::type` value `r` so that $\forall i \in [0, N[$, `r[i] = ! p[i]`

```
template<class T, std::size_t N>
pack<T,N> operator~(pack<T,N> p);
```

Effects: Apply unary `operator~` to every element of `p`

Returns: A `pack<T,N>` value `r` so that :

- If `T` is an integral type: $\forall i \in [0, N[$, `r[i] = ~ p[i]`;
- If `T` is a floating point type, the operation is performed on $r[i]$ bit pattern;
- If `T` is a `logical` type: $\forall i \in [0, N[$, `r[i] = !p[i]`.

5.3.2 Binary Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator+(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator+(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator+(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator-(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator-(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator-(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator*(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator*(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator*(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator/(pack<T,N> p,pack<T,N> q);
```

```
  template<class T, class U, std::size_t N>
  pack<T,N> operator/(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator/(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator%(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator%(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator%(U p, pack<T,N> q);


  template<class T, std::size_t N>
  pack<T,N> operator&(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator&(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator&(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator|(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator|(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator|(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator^(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator^(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator^(U p, pack<T,N> q);
} }
```

```
template<class T, std::size_t N>
pack<T,N> operator+(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator+` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] + q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator+(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator+` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] + static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator+(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator+` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) + q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator-(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator-` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] - q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator-(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator-` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] - static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator-(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator-` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) - q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator*(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator*` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] * q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator*(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator*` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] * static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator*(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator*` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) * q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator/(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator/` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] / q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator/(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator/` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] / static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator/(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator/` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) / q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator%(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator%` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] % q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator%(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator%` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] % static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator%(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator%` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) % q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator&(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator&` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] & q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator&(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator&` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] & static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator&(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator&` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) & q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator|(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator|` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] | q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator|(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator|` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] | static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator|(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator|` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) | q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator^(pack<T,N> p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator^` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] ^ q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator^(pack<T,N> p, U q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator^` between every element of `p` and `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] ^ static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator^(U p, pack<T,N> q);
```

Requires: `T` is not a `logical` type.

Effects: Apply binary `operator^` between `p` and every element of `q`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) ^ q[i]`

5.3.3 Logical Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator<(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator>(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator<=(pack<T,N> p,pack<T,N> q);
```

```
  template<class T, class U, std::size_t N>
  pack<T,N> operator<=(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<=(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator>=(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>=(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>=(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator==(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator==(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator==(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator!=(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator!=(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator!=(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator&&(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator&&(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator&&(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator||(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator||(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator||(U p, pack<T,N> q);


} }

template<class T, std::size_t N>
as_logical<pack<T,N>> operator<(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator<` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] < q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator<(pack<T,N> p, U q);
```

Effects: Apply binary `operator<` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] < static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator<(U p, pack<T,N> q);
```

Effects: Apply binary `operator<` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) < q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator>(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator>` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] > q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator>(pack<T,N> p, U q);
```

Effects: Apply binary `operator>` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] > static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator>(U p, pack<T,N> q);
```

Effects: Apply binary `operator>` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) > q[i]`

```
template<class T, std::size_t N>
```

```
as_logical<pack<T,N>> operator<=(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator<=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] <= q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator<=(pack<T,N> p, U q);
```

Effects: Apply binary `operator<=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] <= static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator<=(U p, pack<T,N> q);
```

Effects: Apply binary `operator<=` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) <= q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator>=(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator>=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] >= q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator>=(pack<T,N> p, U q);
```

Effects: Apply binary `operator>=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] >= static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator>=(U p, pack<T,N> q);
```

Effects: Apply binary `operator>=` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) >= q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator==(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator==` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] == q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator==(pack<T,N> p, U q);
```

Effects: Apply binary `operator==` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] == static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator==(U p, pack<T,N> q);
```

Effects: Apply binary `operator==` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) == q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator!=(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator!=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] != q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator!=(pack<T,N> p, U q);
```

Effects: Apply binary `operator!=` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] != static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator!=(U p, pack<T,N> q);
```

Effects: Apply binary `operator!=` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p)`

```
!= q[i]
```

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator&&(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator&&` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] && q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator&&(pack<T,N> p, U q);
```

Effects: Apply binary `operator&&` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] && static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator&&(U p, pack<T,N> q);
```

Effects: Apply binary `operator&&` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) && q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N>> operator||(pack<T,N> p, pack<T,N> q);
```

Effects: Apply binary `operator||` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] || q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator||(pack<T,N> p, U q);
```

Effects: Apply binary `operator||` between every element of `p` and `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] || static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N>> operator||(U p, pack<T,N> q);
```

Effects: Apply binary `operator||` between `p` and every element of `q`

Returns: A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) || q[i]`

5.3.4 Ternary Operators

```
namespace std { namespace simd
{
  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c,pack<T,N> t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(U c,pack<T,N> t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c, T t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c, pack<T,N> t,T f);
}  }

template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,pack<T,N> t,pack<T,N> f);
```

Effects: Apply ternary `operator?:` between every element of `c`, `t` and `f`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t[i] : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(U c,pack<T,N> t,pack<T,N> f);
```

Effects: Apply ternary `operator?:` between `c` and every element of `t` and `f`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c ? t[i] : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,T t,pack<T,N> f);
```

Effects: Apply ternary `operator?:` between `t` and every element of `c` and `f`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,pack<T,N>  t,T f);
```

Effects: Apply ternary `operator?:` between `f` and every element of `c` and `t`

Returns: A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t[i] : f`

Non-Normative Note If `c` is a `logical` type or a `pack` of `logical` type, implementation of `if_else` can be optimized by not requiring the conversion of `c` to an actual SIMD bitmask.

5.4 Functions

5.4.1 Memory related Functions

```
namespace std { namespace simd
{
  // replicate a scalar value in a pack
  template<class T, class U>
  T splat(U v);

  // convert a pack
  template<class T, std::size_t N, class U>
  T splat(pack<U, N> v);

  // aligned load
  template<class T, class U>
  T load(U* p);

  template<class T, class U>
  T load(U* p, std::ptrdiff_t o);

  // aligned gather
  template<class T, class U, class V, std::size_t N>
  T load(U* p, pack<V, N> o);

  // load with static misalignment
  template<class T, std::ptrdiff_t A, class U>
  T load(U* p);
```

```
  template<class T, std::ptrdiff_t A, class U>
  T load(U* p, std::ptrdiff_t o);

  // gather with static misalignment
  template<class T, std::ptrdiff_t A, class U, class V, std::
     size_t N>
  T load(U* p, pack<V, N> o);

  // unaligned load
  template<class T, class U>
  T unaligned_load(U* p);

  template<class T, class U>
  T unaligned_load(U* p, std::ptrdiff_t o);

  // gather
  template<class T, class U, class V, std::size_t N>
  T unaligned_load(U* p, pack<V, N> o);

  // aligned store
  template<class T, class U>
  void store(T v, U* p);

  template<class T, class U>
  void store(T v, U* p, std::ptrdiff_t o);

  // aligned scatter
  template<class T, class U, class V, std::size_t N>
  void store(pack<T, N> v, U* p, pack<V, N> o);

  // unaligned store
  template<class T, class U>
  void unaligned_store(T v, U* p);

  template<class T, class U>
  void unaligned_store(T v, U* p, std::ptrdiff_t o);

  // scatter
  template<class T, class U, class V, std::size_t N>
  void unaligned_store(pack<T, N> v, U* p, pack<V, N> o);
}  }

template<class T, class U>
T splat(U v);
```

Effects: Convert the value `v` to the type `T`, replicate the value if `T` is a pack.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(v)`.
else `r = static_cast<T>(v)`.

```
template<class T, class U, std::size_t N>
T splat(pack<U, N> v);
```

Requires: `T` is `pack<T2,N>`.

Effects: Convert each element of `v` from `U` to `T2`.

Returns: Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(v[i])`.

Note: While the cardinal of the two packs is the same, the size of the element and therefore the register type being used may change arbitrarily between the input and output of this function.

```
template<class T, class U>
T load(U* p);
```

Requires: `U` is not a pack type, `p` is aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from aligned memory, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, class U>
T load(U* p, std::ptrdiff_t o);
```

Requires: `U` is not a pack type, `p+o` is aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from aligned memory, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, class U, class V, std::size_t N>
T load(U* p, pack<V, N> o);
```

Requires: `U` is not a pack type, `T` is `pack<T2,N>` and all of of `p+o[i]` are aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from aligned indexed memory, possibly after doing a type conversion.

Returns: Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

Note: This is usually known as a *gather* operation.

```
template<class T, std::ptrdiff_t A, class U>
T load(U* p);
```

Requires: `U` is not a pack type, `p-A` is aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from memory whose misalignment is A, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, std::ptrdiff_t A, class U>
T load(U* p, std::ptrdiff_t o);
```

Requires: `U` is not a pack type, `p+o-A` is aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from memory whose misalignment is A, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, std::ptrdiff_t A, class U, class V, std::
    size_t N>
T load(U* p, pack<V, N> o);
```

Requires: `U` is not a pack type, `T` is `pack<T2,N>` and all of of `p+o[i]-A` are aligned on a boundary suitable for loading objects of type `T`.

Effects: Load an object of type `T` from indexed memory whose misalignment is A, possibly after doing a type conversion.

Returns: Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

Note: This is usually known as a *gather* operation.

```
template<class T, class U>
T unaligned_load(U* p);
```

Requires: `U` is not a pack type.

Effects: Load an object of type `T` from memory, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, class U>
T unaligned_load(U* p, std::ptrdiff_t o);
```

Requires: `U` is not a pack type.

Effects: Load an object of type `T` from memory, possibly after doing a type conversion.

Returns: If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, class U, class V, std::size_t N>
T unaligned_load(U* p, pack<V, N> o);
```

Requires: `U` is not a pack type, `T` is `pack<T2,N>`.

Effects: Load an object of type `T` from indexed memory, possibly after doing a type conversion.

Returns: Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

Note: This is usually known as a *gather* operation.

```
template<class T, class U>
void store(T v, U* p);
```

Requires: `U` is not a pack type, `p` is aligned on a boundary suitable for storing objects of type `T`.

Effects: Store the object `v` to memory to aligned memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[i] = static_cast<T2>(v[i])`.
else `*p = static_cast<T>(v)`.

```
template<class T, class U>
void store(T v, U* p, std::ptrdiff_t o);
```

Requires: `U` is not a pack type, `p+o` is aligned on a boundary suitable for storing objects of type `T`.

Effects: Store the object `v` to memory to aligned memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[o+i] = static_cast<T2>(v[i])`.
else `p[o] = static_cast<T>(v)`.

```
template<class T, class U, class V, std::size_t N>
void store(pack<T, N> v, U* p, pack<V, N> o);
```

Requires: `U` is not a pack type and all of `p+o[i]` are aligned on a boundary suitable for storing objects of type `T`.

Effects: Store the object `v` to aligned indexed memory, possibly after doing a type conversion.
Return a value `r` so that $\forall i \in [0, N[$, `p[o[i]] = static_cast<T2>(v[i])`.

Note: This is usually known as a *scatter* operation.

```
template<class T, class U>
void unaligned_store(T v, U* p);
```

Requires: `U` is not a pack type.

Effects: Store the object `v` to memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[i] = static_cast<T2>(v[i])`.
else `*p = static_cast<T>(v)`.

```
template<class T, class U>
void unaligned_store(T v, U* p, std::ptrdiff_t o);
```

Requires: `U` is not a pack type.
Effects: Store the object `v` to memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[o+i] = static_cast<T2>(v[i])`.
else `p[o] = static_cast<T>(v)`.

```
template<class T, class U, class V, std::size_t N>
void unaligned_store(pack<T, N> v, U* p, pack<V, N> o);
```

Requires: `U` is not a pack type and all of `p+o[i]` are aligned on a boundary suitable for storing objects of type `T`.

Effects: Store the object `v` to indexed memory, possibly after doing a type conversion.
Return a value `r` so that $\forall i \in [0, N[$, `p[o[i]] = static_cast<T2>(v[i])`.

Note: This is usually known as a *scatter* operation.

5.4.2 Shuffling Functions

```
namespace std { namespace simd
{
  template<class F, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p);
}  }
```

Requires: `F` is a metafunction class.

Effects: fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = p[F::template apply<i,N>::value]` $\forall i \in [0, N[$.

Returns: The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<std::ptrdiff_t... I, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p);
}  }
```

Requires: `sizeof...(I)` is equal to `N` and `I` belongs to $[0, N[$.

Effects: fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = p[F::template apply<i,N>::value]` $\forall i \in [0, N[$.

Returns: The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<class F, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p1, pack<T,N> p2);
}  }
```

Requires: `F` is a metafunction class.

Effects: fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = (F::template apply<i,N>::value<N) ? p[F::template apply<i,N>::value] :  p[F::template apply<i,N>::value`

- N] $\forall i \in [0, N[$.

Returns: The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<std::ptrdiff_t... I, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p1, pack<T,N> p2);
}  }
```

Requires: `sizeof...(I)` is equal to `N` and `I` belongs to $[0, N[$.

Effects: fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i]=(F::template apply<i,N>::value<N)?p[F::templa apply<i,N>::value]:p[F::template apply<i,N>::value-N]` $\forall i \in [0, N[$.

Returns: The resulting `pack<T,N>`.

5.4.3 Reduction Functions

```
namespace std { namespace simd
{
  template<class T                > T sum(T p);
  template<class T, std::size_t N> T sum(pack<T,N> p);
  template<class T                > T prod(T p);
  template<class T, std::size_t N> T prod(pack<T,N> p);
  template<class T                > T min(T p);
  template<class T, std::size_t N> T min(pack<T,N> p);
  template<class T                > T max(T p);
  template<class T, std::size_t N> T max(pack<T,N> p);

  template<class T                > bool all(T p);
  template<class T, std::size_t N> bool all(pack<T,N> p);
  template<class T                > bool any(T p);
  template<class T, std::size_t N> bool any(pack<T,N> p);
  template<class T                > bool none(T p);
  template<class T, std::size_t N> bool none(pack<T,N> p);
}  }
```

5.4.4 `cmath` Functions

The function supported includes all of the mathematical functions available in the `cmath` header 2.

Table 2: Functions on `pack`

Generic Name	Description
abs	computes the absolute value
div	the quotient and remainder of integer division
fmod	remainder of the floating point division operation
remainder	signed remainder of the division operation
fma	fused multiply-add operation
max	larger of two values
min	smaller of two values
dim	positive difference of two floating point values
nan	not-a-number
exp	returns e raised to the given power
exp2	returns 2 raised to the given power
expm1	returns e raised to the given power, minus one
log	computes natural (base e) logarithm (to base e)
log10	computes common (base 10) logarithm
log1p	natural logarithm (to base e) of 1 plus the given number
log2p	base 2 logarithm of the given number
sqrt	computes square root
cbrt	computes cubic root
hypot	computes square root of the sum of the squares of two given numbers
pow	raises a number to the given power
sin and variants	computes sine (arc sine, hyperbolic sine)
cos and variants	computes cosine (arc cosine, hyperbolic cosine)
tan and variants	computes tangent (arc tangent, hyperbolic tangent)
erf	error function
erfc	complementary error function
lgamma	natural logarithm of the gamma function
tgamma	gamma function
ceil	nearest integer not less than the given value
floor	nearest integer not greater than the given value
trunc	nearest integer not greater in magnitude than the given value
round	nearest integer, rounding away from zero in halfway cases
nearbyint	nearest integer using current rounding mode
rint	nearest integer using current rounding mode with exception if the result differs
frexp	decomposes a number into significand and a power of 2
ldexp	multiplies a number by 2 raised to a power
modf	decomposes a number into integer and fractional parts
logb	extracts exponent of the number
nextafter/nexttoward	next representable floating point value towards the given value
copysign	copies the sign of a value
isfinite	checks if the given number has finite value
isinf	checks if the given number is infinite
isnan	checks if the given number is NaN
isnormal	checks if the given number is normal

Table 2: Functions on `pack`

Generic Name	Description
signbit	checks if the given number is negative
isgreater	checks if the first floating-point argument is greater than the second
isgreaterequal	checks if the first floating-point argument is greater or equal than the second
isless	checks if the first floating-point argument is less than the second
islessequal	checks if the first floating-point argument is less or equal than the second
islessgreater	checks if the first floating-point argument is less or greater than the second
isunordered	checks if two floating-point values are unordered

5.5 Traits and metafunctions

```
namespace std { namespace simd
{
  template<class T> struct scalar_of;

  template<class T> struct cardinal_of;

  template<class T> struct as_logical;
}  }

template<class T> struct scalar_of;
```

Returns: If `T` is a cv or reference qualified `pack<T2, N>` type, return `T2` with the same cv and reference qualifiers. Otherwise return `T`.

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

Returns: If `T` is a cv or reference qualified `pack<T2, N>` type, return `integral_constant<size_t, N>`. Otherwise return `integral_constant<size_t, 1>`.

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

Returns: If `T` is a cv or reference qualified `pack<T2, N>` type with `T2` a non-logical type, return `pack<logical<T2>, N>` with the same cv and reference qualifiers. Else if `T` is a cv or reference qualified non-logical type `T2`, return `logical<T2>` with the same cv and reference qualifiers. Otherwise return `T`.