More `simd<>` Operations

Document Number	P0918R2
Date	2018-10-03
Reply-to	Tim Shen <timshen91@gmail.com>
Audience	SG1, LEWG

Abstract

N4755 [1] defines portable types for SIMD, as well as a set of common SIMD operations. However, the set of operations are not sufficient to expose some of the hardware functionality.

Specifically, some SIMD operations that are heavily used in practice, but delivered by hardware with rather subtle differences, causing importability. This proposal allows these many variations of hardware features to be abstracted into a consistent, portable API.

Revision History

P0918R1 to P0918R2

- Rebased onto N4755.
- Rephrased function descriptions to make the intent clearer.
- Added `raw()`.

P0918R0 to P0918R1

- Dropped `is_signed`, `is_integral`, and `sizeof(...)` constraints on `sum_to` and `multiply_sum_to`. The interface now accepts narrowing conversion.
- For `saturated_simd_cast`, narrowing conversion from floating point to floating point now gives infinity values, when overflow or underflow happen.
- Use the newly suggested name `simd_abi::deduce_t` by P0964R1.
- Remove overloadings of `sum_to` and `multiply_sum_to` that are without accumulators.

Proposed Functions

shuffle

```
template <size_t... indices, typename T, typename Abi>
simd<T, simd_abi::deduce_t<T, sizeof...(indices), Abi>>
shuffle(const simd<T, Abi>& v);
```

```
template <size_t... indices, typename T, typename Abi>
simd_mask<T, simd_abi::deduce_t<T, sizeof...(indices), Abi>>
shuffle(const simd_mask<T, Abi>& v);
```

Remarks: These functions shall not participate overloading resolution unless `((indices < simd_size_v<T, Abi>) && ...)`.

Returns: A new simd/simd_mask object `r`, where `r[i]` = `v[j]` and `j` is the ith element in `indices`.

The shuffle fetches elements from the input `simd<>` object. The fetched elements are specified by the input variadic pack of indices.

Note that hardware often provide interfaces that take two SIMD values, not one. For the proposed portable interface, this can be achieved by composing with `concat()`, e.g. `` `shuffle<7, 6, 5, 4, 3, 2, 1, 0>(concat(a, b))` ``, where `a` and `b` are with sizes of 4. With compiler optimizations, this comes to no performance penalty[1]. The single-argument shuffle is easier to learn and result in more explicit call sites.

Note that for variadic number of elements, users can use `std::index_sequence`:

```
template <size_t... indices>
simd<int> ElementsWithOddIndices(simd<int> a, simd<int> b,
                                 std::index_sequence<indices...>) {
  static_assert(sizeof...(indices) == a.size(), "");
  // Returns all elements with odd indices in concatenated a and b.
  return shuffle<(2 * indices + 1)...>(concat(a, b));
}
```

interleave

```
template <typename T, typename Abi>
simd<T, simd_abi::deduce_t<T, simd_size_v<T, Abi> * 2, Abi>>
interleave(const simd<T, Abi>& u, const simd<T, Abi>& v);
```

```
template <typename T, typename Abi>
simd_mask<T, simd_abi::deduce_t<T, simd_size_v<T, Abi> * 2, Abi>>
interleave(const simd_mask<T, Abi>& u, const simd_mask<T, Abi>& v);
```

Returns: `shuffle<(i / 2 + (i % 2) * simd_size_v<T, Abi>)...>(concat(u, v))`, where `i` is a variadic pack of size_t in `[0, simd_size_v<T, Abi> * 2)`.

---

[1] https://godbolt.org/g/BEXRmZ

`interleave()` takes two `simd<>` objects with equal size, and interleave them to produce a twice as long `simd<>` object.

Hardware instructions like `punpcklwd` on x86 can be achieved by combining `split()` and `interleave()`, ``interleave(split_by<2>(a)[0], split_by<2>(b)[0])`` with the assist of proper optimizations[2]; vice versa, `interleave()` itself can be implemented in terms of instructions like `punpcklwd`.

sum_to

```
template <typename AccType, typename T, typename Abi>
AccType sum_to(const simd<T, Abi>& v, const AccType& acc);
```

Let `U` be `typename AccType::value_type`.

Remarks: This function shall not participate overloading resolution unless

- `is_simd_v<AccType>`, and
- `simd<T, Abi>::size() % AccType::size() == 0`.

Returns: `r + acc`, where `r[i]` is `GENERALIZED_SUM(std::plus<>, static_cast<U>(v[S*i]), static_cast<U>(v[S*i+1]), ..., static_cast<U>(v[(S+1)*i - 1]))`, and `S` is `v.size() / AccType::size()`. For all `i`, `r[i]` has an unspecified value if the corresponding `GENERALIZED_SUM` overflows.

`sum_to` takes a `simd<>` object and an accumulator, and tries to do a "partial-reduction" on the `simd<>` object, then add the result to the accumulator. "partial-reduction" means that reducing N elements down to M by partially summing them up, where N has to be a multiple of M. Only adjacent input elements will be summed up.

On some architectures - x86 for example - this can be used to implement an efficient[3] full summation over a large buffer of integers:

```
// Returns the sum of all uint8_ts in the buffer.
int64_t Sum(uint8_t* buf, int n) {
  constexpr size_t stride = native_simd<uint8_t>::size();
  native_simd<int64_t> acc(0);
  int i;
  for (i = 0; n - i >= stride; i += stride) {
    acc = sum_to(native_simd<uint8_t>(buf + i), acc);
```

[2] https://godbolt.org/g/svsyfh

[3] `_mm_sad_epu8` is the fastest approach in the benchmark: https://gist.github.com/timshen91/0f321fe2c5cfb04015917c0529052158

```
  }
  // handle leftovers in [i, n)
  return reduce(acc);
}
```

In practice, summation usage does not always fit in one or more calls to `Sum()`, e.g. multiple summations with their loops fused. Therefore, it makes sense to let the accumulator `acc` and the loop exposed in the user code.

This provides a simple and consistent interface for various flavors of hardware summation instructions:

- Elements are not widened, and total number of bytes is changed: `phaddd` on x86, `VPADD` on ARM.
- Elements are widened, but total number of bytes isn't changed: `psadbw`, `pmaddwd` on x86, `vmsumshm` on PowerPC.
- Full sum, e.g. `ADDV` on ARMv8.

Note that the efficiency of `sum_to()` is architecture-specific for a given `(T, Abi, AccType)` combination. Users do need architectural knowledge to pick the most efficient `AccType` on that architecture, as well as using `sum_to()` or not. Implementations are suggested to document which instruction is generated by which instantiation, and warn about uses of inefficient ones.

multiply_sum_to

```
template <typename AccType, typename T, typename Abi>
AccType multiply_sum_to(
    const simd<T, Abi>& v, const simd<T, Abi>& u, const AccType& acc);
```

Let `U` be `typename AccType::value_type`.

Remarks: This function shall not participate overloading resolution unless

- `is_simd_v<AccType>`, and
- `simd<T, Abi>::size() % AccType::size() == 0`.

Returns: `sum_to(static_simd_cast<U>(v) * static_simd_cast<U>(u), acc)`.

This function does element-wise multiply, followed by a sum_to.

The main purpose is to provide a specialization point so that the implementation can have guaranteed single-instruction per function call. For example

- `pmaddwd` on x86
- `vmsumshm` on PowerPC
- `VMLAL` on ARM

In practice, this is often used for implementing integral dot product. It makes sense to expose the accumulator to the users for the same reason as `sum_to()` does.

saturated_simd_cast

```
template <typename U, typename T, typename Abi>
simd<U, simd_abi::deduce_t<U, simd_size_v<T, Abi>, Abi>>
saturated_simd_cast(const simd<T, Abi>& v);
```

If `is_integral_v<U>`, then let `L` be `numeric_limits<U>::min()` and `R` be `numeric_limits<U>::max()`.

If `is_floating_point_v<U>`, then `L` is `-numeric_limits<U>::infinity()` and `R` is `numeric_limits<U>::infinity()`.

Remarks: This function shall not participate overloading resolution unless `U` is a vectorizable type

Returns: A simd object `r`, where `r[i]` is

- `L`, if `v[i]` underflows when converting to `U`, or
- `R`, if `v[i]` overflows when converting to `U`, or
- `static_cast<U>(v[i])`.

This function is similar to `simd_cast()`, but clamps the result when overflow happens.

This captures many of the uses of "saturated pack" integral operation, which effectively narrows down each element by half of its size, and clamps each narrowed value.

It also provides floating point -> integer saturated conversion.

Hardware instruction examples include:

- `packsswb`, `packuswb` on x86
- `vpkswss`, `vpkswus`, `vctsxs` on PowerPC
- `VQMOVN`, `VQMOVUN` on ARM

raw

Change [9.3.1]p4 to the following:
Implementations should enable explicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class `simd`:

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

When only one such implementation-defined type exists for a given `simd` type, implementations should also have a `raw()` member function as specified:

```
    implementation-defined raw() const {
      return static_cast<implementation-defined>(*this);
    }
```

`raw()` is proposed for convenience, and the users don't have to spell out the implementation-defined type.

Prototype

Dimsum [2] implements variations of `shuffle()`, `interleave()` (with the name `zip`), `sum_to()` (with the name `reduce_add`), and `multiply_sum_to()` (with the name `mul_sum`).

Reference

[1] N4755, the SIMD proposal
[2] Dimsum, the prototype