simd_invoke

Daniel Towner, Ruslan Arutyunyan

Interacting with intrinsics

- The target architecture may have exotic instructions which will not have a standard API (e.g., addsub).
- Simd values can be cast to their compiler-builtin types, and back again:

```
simd<float>
addsub(simd<float> a, simd<float> b) {

    __m256 t = _mm256_addsub_ps(static_cast<__m256>(a),
                                static_cast<__m256>(b));

    return simd<float>(t);
}
```

A mixture of native sizes

- Intel has 128-bit and 256-bit versions of addsub.

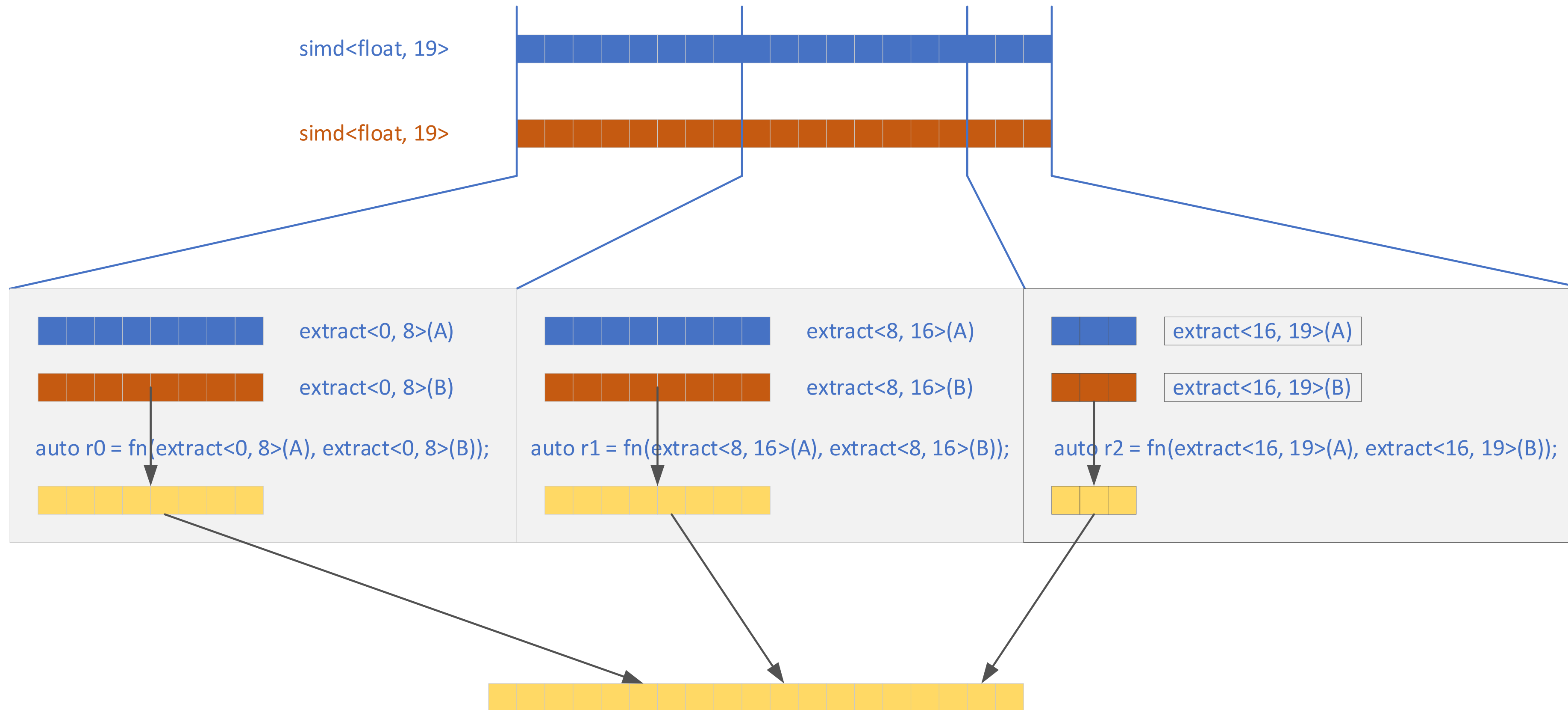
```
auto
addsub(auto a, auto b) {

  if constexpr (a.size <= 4)
    return simd<float, a.size>(_mm_addsub_ps(static_cast<__m128>(a), static_cast<__m128>(b)));
  else if constexpr (a.size <= 8)
    return simd<float, a.size>(_mm256_addsub_ps(static_cast<__m256>(a), static_cast<__m256>(b)));
  else
    error(); // Invalid native register
}
```

Handling larger sizes

```
simd<float>
addsub(simd<float, 16> a, simd<float, 16> b) {

  // Get register-sized pieces
  auto {lowA, highA} = simd_split<simd<float>>(a);
  auto {lowB, highB} = simd_split<simd<float>>(b);

  // Call the intrinsic on each pair of pieces.
  auto resultLow = simd<float, 16>(_mm256_addsub_ps(static_cast<__m256>(lowA),
                                                     static_cast<__m256>(lowB)));
  auto resultHigh = simd<float, 16>(_mm256_addsub_ps(static_cast<__m256>(highA),
                                                      static_cast<__m256>(highB)));

  // Glue the individual results back together.
  return simd_concat(resultLow, resultHigh);
}
```

Introducing `simd_invoke`

```
simd<float, 19> A;
simd<float, 19> B;
auto r = simd_invoke<8>(fn, A, B);
```




```
result = simd_concat(r0, r1, r2);
```

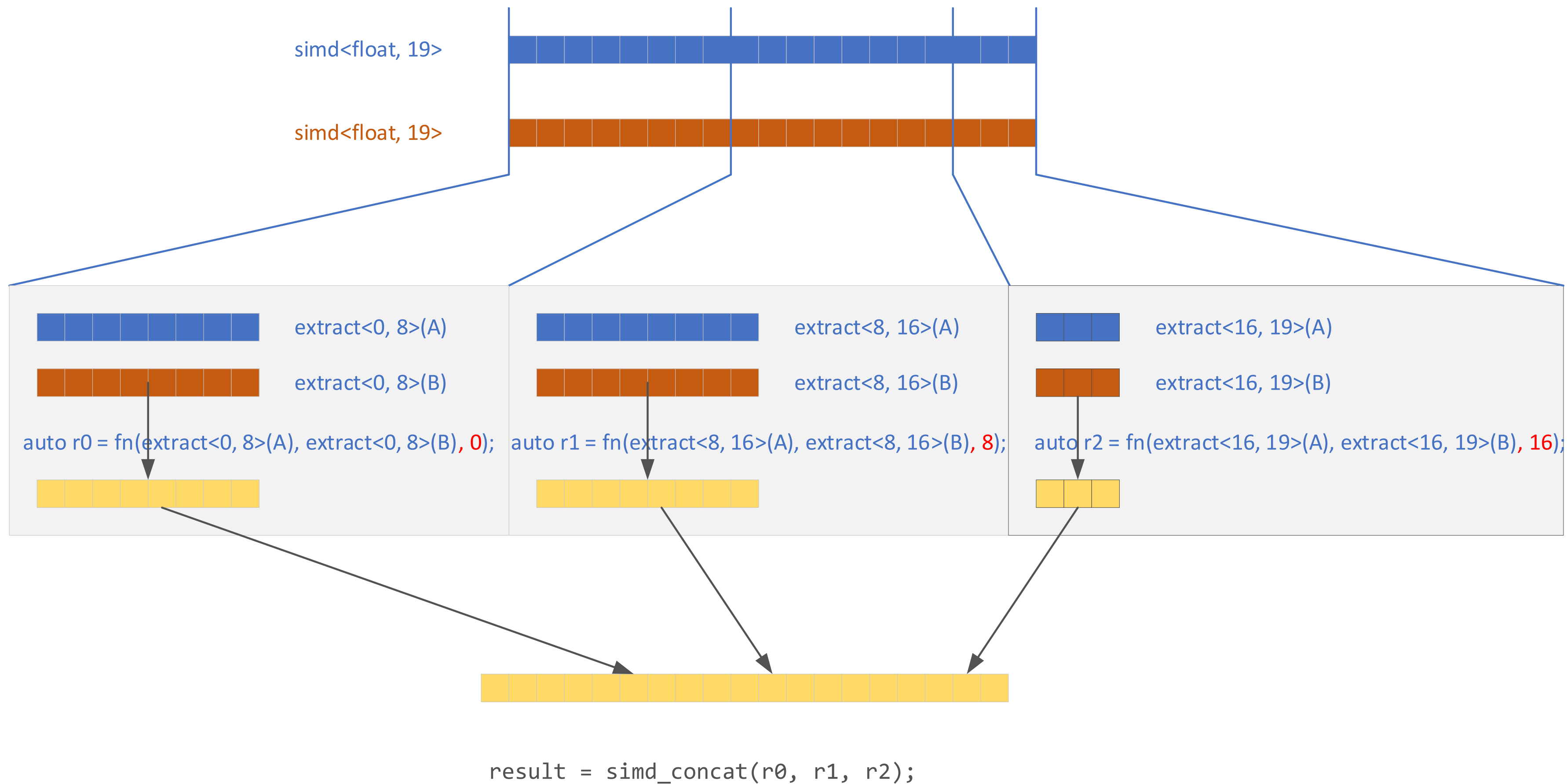
Worked example

```cpp
auto addsub(const simd<float, 19>& x, const simd<float, 19>& y)
{
    auto native = [](auto a, auto b) {
        if constexpr (a.size <= 4)
            return simd<float, a.size>(_mm_addsub_ps(static_cast<__m128>(a),
                                                     static_cast<__m128>(b)));
        else if constexpr (a.size <= 8)
            return simd<float, a.size>(_mm256_addsub_ps(static_cast<__m256>(a),
                                                        static_cast<__m256>(b)));
    };

    return simd_invoke(native, x, y);
}
```

```asm
addsub(xvec::basic_simd<float, xvec::simd_fixed_size_abi<19> > const&, x
        mov     rax, rdi
        vmovups ymm0, ymmword ptr [rsi]           # AlignMOV convert to Un
        vaddsubps       ymm0, ymm0, ymmword ptr [rdx]
        vmovups ymm1, ymmword ptr [rsi + 32]      # AlignMOV convert to Un
        vaddsubps       ymm1, ymm1, ymmword ptr [rdx + 32]
        vmovups xmm2, xmmword ptr [rsi + 64]      # AlignMOV convert to Un
        vaddsubps       xmm2, xmm2, xmmword ptr [rdx + 64]
        vmovups ymmword ptr [rdi], ymm0           # AlignMOV convert to Un
        vmovups ymmword ptr [rdi + 32], ymm1      # AlignMOV convert to Un
        vmovlps qword ptr [rdi + 64], xmm2
```

Indexing – pass in the index of each piece too

```
simd<float, 19> A;
simd<float, 19> B;
auto r = simd_invoke_indexed<8>(fn, A, B);
```




```
result = simd_concat(r0, r1, r2);
```

Indexed example

```
auto special_memory_store(simd<float, 32> x, float* ptr)
{
    auto do_native =
      [=]<typename T, typename ABI>(basic_simd<T, ABI> data, auto idx) {
        (_mm256_store_ps(ptr + idx, static_cast<__m256>(data)));
    };

    simd_invoke(do_native, x);
}
```

```
special_memory_store(xvec::basic_simd<float, xvec::simd_fixed_size_abi<32> >, flo
        vmovups ymm0, ymmword ptr [rdi]          # AlignMOV convert to UnAlignMOV
        vmovups ymm1, ymmword ptr [rdi + 32]     # AlignMOV convert to UnAlignMOV
        vmovups ymm2, ymmword ptr [rdi + 64]     # AlignMOV convert to UnAlignMOV
        vmovups ymm3, ymmword ptr [rdi + 96]     # AlignMOV convert to UnAlignMOV
        vmovups ymmword ptr [rsi], ymm0          # AlignMOV convert to UnAlignMOV
        vmovups ymmword ptr [rsi + 32], ymm1     # AlignMOV convert to UnAlignMOV
        vmovups ymmword ptr [rsi + 64], ymm2     # AlignMOV convert to UnAlignMOV
        vmovups ymmword ptr [rsi + 96], ymm3     # AlignMOV convert to UnAlignMOV
```

simd_invoke size detection

simd_invoke detects the block size from the native size.

Given an argument of type simd<float>, it will choose simd<float>::size as the block size (like simd_split).

```
simd<float, 19> on AVX2
->        simd<float,8>, simd<float,8>, simd<float, 3>
```

But **simd_invoke<simd<float, 19>, simd<char, 19>>**

simd<float>::size != simd<char>::size

Explicitly pass in the block size: simd_invoke<8>(floats, chars);

```cpp
template<std::floating_point _Tp, typename _Abi>
constexpr basic_simd<_Tp, _Abi> fma(const basic_simd<_Tp, _Abi>& x, const basic_simd<_Tp, _Abi>& y,
{
  target_overloads impl {
    // 128-bit
    [=](xmm_simd_impl<float> auto x, auto y, auto acc)
      { return _mm_fmadd_ps(x.to_register(), y.to_register(), acc.to_register()); },
    [=](xmm_simd_impl<double> auto x, auto y, auto acc)
      { return _mm_fmadd_pd(x.to_register(), y.to_register(), acc.to_register()); },

    // 256-bit
    [=](ymm_simd_impl<float> auto x, auto y, auto acc)
      { return _mm256_fmadd_ps(x.to_register(), y.to_register(), acc.to_register()); },
    [=](ymm_simd_impl<double> auto x, auto y, auto acc)
      { return _mm256_fmadd_pd(x.to_register(), y.to_register(), acc.to_register()); },

    // 512-bit
    [=](zmm_simd_impl<float> auto x, auto y, auto acc)
      { return _mm512_fmadd_ps(x.to_register(), y.to_register(), acc.to_register()); },
    [=](zmm_simd_impl<double> auto x, auto y, auto acc)
      { return _mm512_fmadd_pd(x.to_register(), y.to_register(), acc.to_register()); },
  };

  auto wrapper = [=](auto a, auto b, auto c) { return decltype(a)(impl(a, b, c)); }; // Return a si
  return simd_invoke(wrapper, x, y, acc);
}
```