Bits

Sonntag, 19. Juni 2022 20:09

Author: Damir Ljubic
email: damirlj@yahoo.com

Code: https://godbolt.org/z/6qGPsTT3h
(Compiler Explorer)

The alternative implementation in C++17 for the some important bitwise operation (in terms of the memory alignment), that are first included in std library with the C++20

- **std::popcount**: counting 1 bits (https://en.cppreference.com/w/cpp/numeric/popcount)
- **std::countl_zero**: counting leading zeros (https://en.cppreference.com/w/cpp/numeric/countl_zero)
- **std::bit_width** $(1 + \log_2(x))$: bits required for the number binary representation (https://en.cppreference.com/w/cpp/numeric/bit_width)

The entire code, from the link above

```cpp
#include <bit>
#include <bitset>
#include <iostream>
#include <type_traits>


// There is a matching concept in C++20
template <typename T>
static constexpr bool is_unsigned_integral_v = std::is_integral_v<T> && !std::is_signed_v<T>;

/*
 *   The number a is power of two ( a = 2^n == 1 << n ), if
 *   it has only one bit (n^th) set to 1.
 *   The most convenient way to test it is: "a & (a-1) == 0"
*/
template <typename T>
constexpr bool is_power_of_2(const T n) noexcept
{
    static_assert(is_unsigned_integral_v<T>, "<Error> Works only with unsigned integral types!");

    if (n < 1) return false;
    return (n & (n-1)) == 0;
}


template <typename T>
constexpr std::size_t log2(const T n) noexcept; // forward declaration

template <typename T>
constexpr void printIsPowerOf2(const T n) noexcept
{
   const auto is_power_of_2_v = is_power_of_2(n);
   std::cout << std::boolalpha << "is_power_of_2(" << n <<"): " << is_power_of_2_v << '\n';
   if (is_power_of_2_v)
   {
       std::cout << "log2(" << n << ")=" << log2(n) << '\n';
   }
}


/*
* Count the number of '1' bits in binary representation of
* an integral number
* @see C++20 std::popcount
* https://en.cppreference.com/w/cpp/numeric/popcount
*/
template <typename T>
constexpr std::size_t count_ones(T n) noexcept
{
    static_assert(std::is_integral_v<T>, "<Error> Works only with integral types!");

    if (n == 1 || n == 2 ) return 1;

    std::size_t count = 0;
    while (n)
    {
        n = n & (n-1); // iterate until we reach the last "power of two" element: n & (n-1) == 0
        ++count;
    }

    return count;
}

template <typename T>
constexpr void printCountOnes(const T n) noexcept
{
```

```
    constexpr auto bits = 8 * sizeof(T);
    const auto ones = count_ones(n);


    // Test against the C++20 standard function
    #if __cplusplus >= 202002L

        assert(std::popcount(n) == static_cast<int>(ones));
    #endif

    if constexpr(std::is_same_v<T, std::uint8_t>)
    {
         std::cout << "n=" << static_cast<uint16_t>(n) << "(" << std::bitset<bits>(n) << "), ones: " << ones << '\n';
    }
    else
    {
         std::cout << "n=" << n << "(" << std::bitset<bits>(n) << "), ones: " << ones << '\n';
    }
}


using bit_t = enum class bit: std::uint8_t
{
    zero = 0,
    one = 1
};

/*
* Counts leading 0/1 in bits representation of a number
* Replacement for the C++20: std::countl_zero
* https://en.cppreference.com/w/cpp/numeric/countl_zero
*/
template <typename T>
constexpr std::size_t count_leading_x(const T n, const bit_t x) noexcept
{
   static_assert(std::is_integral_v<T>, "<Error> Works only with integral types!");

   constexpr auto bits = 8 * sizeof(T);
   const auto c_x = std::underlying_type_t<bit_t>(x);
   const auto n_bits = std::bitset<bits>(n).to_ullong(); // the biggest representation, for working with negative numbers as well
   auto position = bits - 1; // bits weight

   std::size_t count = 0;

   for (;;)
   {
       if (const auto mask = 1 << position; (mask & n_bits) >> position != c_x) break;
       ++count;
       if (position == 0) break;
       --position;
   }

   return count;
}

/*
* Calculates log2(x), where x must be power of two.
* It's replacement for C++20 std::bit_width (1 + log₂(x))
* https://en.cppreference.com/w/cpp/numeric/bit_width
*/
template <typename T>
constexpr std::size_t log2(const T n) noexcept
{
    if (n < 2) return 0;
    if (n == 2) return 1;

    constexpr auto digits = 8 * sizeof(T);
    return (digits - count_leading_x(n, bit_t::zero)) - 1u;
}

constexpr void test()
{
    constexpr std::uint32_t numbers[] = {0, 1, 2, 23, 64, 128};
    for (const auto n : numbers)
    {
        printCountOnes(n);
        printIsPowerOf2(n);
        std::cout << "Leading 0: " << count_leading_x(n, bit_t::zero) << '\n';
        std::cout << "Leading 1: " << count_leading_x(n, bit_t::one) << '\n';
    }
}

int main()
{
    test();
}
```