LinksPlatform's Platform.Numbers Class Library

1.1 ./csharp/Platform.Numbers/Arithmetic.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of arithmetic methods.</para>
    /// <para>Представляет набор арифметических методов.</para>
    /// </summary>
    public static class Arithmetic
    {
        /// <summary>
        /// <para>Performing adding the x and y arguments.</para>
        /// <para>Выполняет сложение аргументов x и y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The first term.</para>
        /// <para>Первое слагаемое.</para>
        /// </param>
        /// <param name="y">
        /// <para>The second term.</para>
        /// <para>Второе слагаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Sum of x and y.</para>
        /// <para>Сумма x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);

        /// <summary>
        /// <para>Performs subtracting y from x.</para>
        /// <para>Выполняет вычитание y из x.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>Minuend.</para>
        /// <para>Уменьшаемое.</para>
        /// </param>
        /// <param name="y">
        /// <para>Subtrahend.</para>
        /// <para>Вычитаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Difference between x and y.</para>
        /// <para>Разность между x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);

        /// <summary>
        /// <para>Performs multiplication x by y.</para>
        /// <para>Выполняет умножение x на y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>First multiplier.</para>
        /// <para>Первый множитель.</para>
        /// </param>
        /// <param name="y">
        /// <para>Second multiplier.</para>
        /// <para>Второй множитель.</para>
        /// </param>
        /// <returns>
        /// <para>Product of x and y.</para>
        /// <para>Произведение x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);

        /// <summary>
        /// <para>Performs dividing x by y.</para>
        /// <para>Выполняет деление x на y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>Dividend.</para>
        /// <para>Делимое.</para>
        /// </param>
        /// <param name="y">
        /// <para>Divider.</para>
        /// <para>Делитель.</para>
        /// </param>
        /// <returns>
        /// <para>Quoitent of x and y.</para>
        /// <para>Частное x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);

        /// <summary>
        /// <para>Increasing the number by one.</para>
        /// <para>Увеличивает число на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to increase.</para>
        /// <para>Число для увеличения.</para>
        /// </param>
        /// <returns>
        /// <para>Increase by one number.</para>
        /// <para>Увеличенное на единицу число.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para>Increases the value of argument by one.</para>
        /// <para>Увеличивает значение аргумента на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The argument to increase.</para>
        /// <para>Аргумент для увеличения.</para>
        /// </param>
        /// <returns>
        /// <para>Increased argument value.</para>
        /// <para>Увеличенное значение аргумента.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para>Decreases number by one.</para>
        /// <para>Уменьшение числа на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to reduce.</para>
        /// <para>Число для уменьшения.</para>
        /// </param>
        /// <returns>
        /// <para>Decreased by one number.</para>
        /// <para>Уменьшенное на единицу число.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);

        /// <summary>
        /// <para>Decreases the value of the argument by one.</para>
        /// <para>Уменьшает значение аргумента на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The argument to reduce.</para>
        /// <para>Аргумент для уменьшения.</para>
        /// </param>
        /// <returns>
        /// <para>Decreased argument value.</para>
        /// <para>Уменьшеное значение аргумента.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(ref T x) => x = Arithmetic<T>.Decrement(x);
    }
}
```

1.2 ./csharp/Platform.Numbers/ArithmeticExtensions.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of extension methods that perform arithmetic operations on
     ↪  arbitrary object types.</para>
    /// <para>Представляет набор методов расширения выполняющих арифметические операции для
     ↪  объектов произвольного типа.</para>
    /// </summary>
    public static class ArithmeticExtensions
    {
        /// <summary>
        /// <para>Increments the variable passed as an argument by one.</para>
        /// <para>Увеличивает переданную в качестве аргумента переменную на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The reference to the incremented variable.</para>
        /// <para>Ссылка на увеличиваемую переменную.</para>
        /// </param>
        /// <returns>
        /// <para>The value of the argument incremented by one.</para>
        /// <para>Увеличенное значение аргумента на единицу.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(this ref T x) where T : struct => x =
         ↪  Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para>Decrements the variable passed as an argument by one.</para>
        /// <para>Уменьшает переданную в качестве аргумента переменную на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The reference to the decremented variable.</para>
        /// <para>Ссылка на уменьшаемую переменную.</para>
        /// </param>
        /// <returns>
        /// <para>The value of the argument decremented by one.</para>
        /// <para>Уменьшеное значение аргумента на единицу.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(this ref T x) where T : struct => x =
         ↪  Arithmetic<T>.Decrement(x);
    }
}
```

1.3 ./csharp/Platform.Numbers/Arithmetic[T].cs

```
using System;
using System.Reflection.Emit;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType
namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of compiled arithmetic operations delegates.</para>
    /// <para>Представляет набор скомпилированных делегатов арифметических операций.</para>
    /// </summary>
    public static class Arithmetic<T>
    {
        /// <summary>
        /// <para>A read-only field that represents a addition function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции сложения.</para>
        /// </summary>
        public static readonly Func<T, T, T> Add = CompileAddDelegate();

        /// <summary>
        /// <para>A read-only field that represents a subtraction function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции вычитания.</para>
        /// </summary>
        public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();

        /// <summary>
        /// <para>A read-only field that represents a multiplication function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции умножения.</para>
        /// </summary>
        public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();

        /// <summary>
        /// <para>A read-only field that represents a division function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции деления.</para>
        /// </summary>
        public static readonly Func<T, T, T> Divide = CompileDivideDelegate();

        /// <summary>
        /// <para>A read-only field that represents a increment function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции инкремента.</para>
        /// </summary>
        public static readonly Func<T, T> Increment = CompileIncrementDelegate();

        /// <summary>
        /// <para>A read-only field that represents a decrement function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции декремента.</para>
        /// </summary>
        public static readonly Func<T, T> Decrement = CompileDecrementDelegate();

        /// <summary>
        /// <para>
        /// Compiles the add delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileAddDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Add();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the subtract delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileSubtractDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Subtract();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the multiply delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileMultiplyDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Emit(OpCodes.Mul);
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the divide delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileDivideDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                if(NumericType<T>.IsSigned)
                {
                    emiter.Emit(OpCodes.Div);
                }
                else
                {
                    emiter.Emit(OpCodes.Div_Un);
                }
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the increment delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileIncrementDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                emiter.Increment<T>();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the decrement delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileDecrementDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                emiter.Decrement<T>();
                emiter.Return();
            });
        }
    }
}
```

1.4 ./csharp/Platform.Numbers/Bit.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>A set of operations on the set bits of a number.</para>
    /// <para>Набор операций над установленными битами числа.</para>
    /// </summary>
    public static class Bit
    {
        /// <summary>
        /// <para>Counts the number of bits set in a number.</para>
        /// <para>Подсчитывает количество установленных бит в числе.</para>
        /// </summary>
        /// <param name="x">
        /// <para>Bitwise number.</para>
        /// <para>Число в битовом представлении.</para>
        /// </param>
        /// <returns>
        /// <para>Number of bits set in a number.</para>
        /// <para>Количество установленных бит в числе.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Count(long x)
        {
            long n = 0;
            while (x != 0)
            {
                n++;
                x &= x - 1;
            }
            return n;
        }

        /// <summary>
        /// <para>Searches for the first bit set in a number.</para>
        /// <para>Ищет первый установленный бит в числе.</para>
        /// </summary>
        /// <param name="value">
        /// <para>Bitwise number.</para>
        /// <para>Число в битовом представлении.</para>
        /// </param>
        /// <returns>
        /// <para>First bit set.</para>
        /// <para>Первый установленный бит.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static int GetLowestPosition(ulong value)
        {
            if (value == 0)
            {
                return -1;
            }
            var position = 0;
            while ((value & 1UL) == 0)
            {
                value >>= 1;
                ++position;
            }
            return position;
        }

        /// <summary>
        /// <para>Performing bitwise inversion of a number.</para>
        /// <para>Выполняет побитовую инверсию числа.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>Number to invert.</para>
        /// <para>Число для инверсии.</para>
        /// </param>
        /// <returns>
        /// <para>Inverse value of the number.</para>
        /// <para>Обратное значение числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(T x) => Bit<T>.Not(x);

        /// <summary>
        /// <para>Performing bitwise numbers addition.</para>
        /// <para>Выполняет побитовое сложение чисел.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>First term.</para>
        /// <para>Первое слагаемое.</para>
        /// </param>
        /// <param name="y">
        /// <para>Second term.</para>
        /// <para>Второе слагаемое.</para>
        /// </param>
        /// <returns>
        /// <para>The logical sum of numbers</para>
        /// <para>Логическая сумма чисел.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);

        /// <summary>
        /// <para>Performs bitwise numbers multiplication.</para>
        /// <para>Выполняет побитовое умножение чисел.</para>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// </summary>
        /// <param name="x">
        /// <para>First multiplier.</para>
        /// <para>Первый множитель.</para>
        /// </param>
        /// <param name="y">
        /// <para>Second multiplier.</para>
        /// <para>Второй множитель.</para>
        /// </param>
        /// <returns>
        /// <para>Logical product of numbers.</para>
        /// <para>Логическое произведение чисел.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T And<T>(T x, T y) => Bit<T>.And(x, y);

        /// <summary>
        /// <para>Performs a bitwise shift of a number to the left by the specified number of bits.</para>
        /// <para>Выполняет побитовый свиг числа влево на указанное количество бит.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number on which the left bitwise shift operation will be performed.</para>
        /// <para>Число над которым будет производиться операция пиботового смещения влево.</para>
        /// </param>
        /// <param name="y">
        /// <para>The number of bits to shift.</para>
        /// <para>Количество бит на которые выполнить смещение.</para>
        /// </param>
        /// <returns>
        /// <para>The value with discarded high-order bits that are outside the range of the number's type and set low-order empty bit positions to zero.</para>
        /// <para>Значение с отброшенными старшими битами, которые находятся за пределами диапазона типа числа и устанавливленными пустыми битовыми позициями младших разрядов в ноль.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);

        /// <summary>
        /// <para>Performs a bitwise shift of a number to the right by the specified number of bits.</para>
        /// <para>Выполняет побитовый свиг числа вправо на указанное количество бит.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number on which the right bitwise shift operation will be performed.</para>
        /// <para>Число над которым будет производиться операция побитового смещения вправо.</para>
        /// </param>
        /// <param name="y">
        /// <para>The number of bits to shift.</para>
        /// <para>Количество бит на которые выполнить смещение.</para>
        /// </param>
        /// <returns>
        /// <para>The value with discarded low-order bits.</para>
        /// <para>Значение с отброшенными младшими битами.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);

        /// <summary>
        /// <para>Performs a partial write of a specified number of bits from source number to target number.</para>
        /// <para>Выполняет частичную запись определенного количества бит исходного числа в целевое число.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The value to which the partial write will be performed.</para>
        /// <para>Значение в которое будет выполнена частичная запись.</para>
        /// </param>
        /// <param name="source">
        /// <para>Data source for recording.</para>
        /// <para>Источник данных для записи.</para>
        /// </param>
        /// <param name="shift">
        /// <para>The start position to read from.</para>
        /// <para>Стартовая позиция чтения.</para>
        /// </param>
        /// <param name="limit">
        /// <para>The number of bits to write from source to target.</para>
        /// <para>Количество бит, которые нужно записать из source в target.</para>
        /// </param>
        /// <returns>
        /// <para>The target number updated with bits from source number.</para>
        /// <para>Целевое число с обновленными битами из исходного числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
         ↪  Bit<T>.PartialWrite(target, source, shift, limit);

        /// <summary>
        /// <para>Reads a specified number of bits from the number at specified position.</para>
        /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The number from which the partial read will be performed.</para>
        /// <para>Число из которого будет выполнено частичное чтение.</para>
        /// </param>
        /// <param name="shift">
        /// <para>The start position to read from.</para>
        /// <para>Стартовая позиция чтения.</para>
        /// </param>
        /// <param name="limit">
        /// <para>The number of bits to read.</para>
        /// <para>Количество бит, которые нужно считать.</para>
        /// </param>
        /// <returns>
        /// <para>The number consisting of bits read from the source number.</para>
        /// <para>Число состоящее из считанных из исходного числа бит.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialRead<T>(T target, int shift, int limit) =>
         ↪  Bit<T>.PartialRead(target, shift, limit);
    }
}
```

1.5 ./csharp/Platform.Numbers/BitExtensions.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of bitwise operation.</para>
    /// <para>Представляет набор битовых операций.</para>
    /// </summary>
    public static class BitwiseExtensions
    {
        /// <summary>
        /// <para>Performs bitwise inversion of a number.</para>
        /// <para>Выполняет побитовую инверсию числа.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The number to invert.</para>
        /// <para>Число для инверсии.</para>
        /// </param>
        /// <returns>
        /// <para>An inverted value of the number.</para>
        /// <para>Обратное значение числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);

        /// <summary>
        /// <para>Performs a partial write of a specified number of bits from source number to target number.</para>
        /// <para>Выполняет частичную запись определенного количества бит исходного числа в целевое число.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The value to which the partial write will be performed.</para>
        /// <para>Значение в которое будет выполнена частичная запись.</para>
        /// </param>
        /// <param name="source">
        /// <para>Data source for writing.</para>
        /// <para>Источник данных для записи.</para>
        /// </param>
        /// <param name="shift">
        /// <para>The start position to read from.</para>
        /// <para>Стартовая позиция чтения.</para>
        /// </param>
        /// <param name="limit">
        /// <para>The number of bits to write from source to target.</para>
        /// <para>Количество бит, которые нужно записать из source в target.</para>
        /// </param>
        /// <returns>
        /// <para>The target number updated with bits from source number.</para>
        /// <para>Целевое число с обновленными битами из исходного числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);

        /// <summary>
        /// <para>Reads a specified number of bits from the number at specified position.</para>
        /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The number from which the partial read will be performed.</para>
        /// <para>Число из которого будет выполнено частичное чтение.</para>
        /// </param>
        /// <param name="shift">
        /// <para>The start position to read from.</para>
        /// <para>Стартовая позиция чтения.</para>
        /// </param>
        /// <param name="limit">
        /// <para>The number of bits to read.</para>
        /// <para>Количество бит, которые нужно считать.</para>
        /// </param>
        /// <returns>
        /// <para>The number consisting of bits read from the source number.</para>
        /// <para>Число состоящее из считанных из исходного числа бит.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialRead<T>(this T target, int shift, int limit) => Bit<T>.PartialRead(target, shift, limit);
    }
}
```

1.6 ./csharp/Platform.Numbers/Bit[T].cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of compiled bit operations delegates.</para>
```

```csharp
    /// <para>Представляет набор скомпилированных делегатов битовых операций.</para>
    /// </summary>
    public static class Bit<T>
    {
        /// <summary>
        /// <para>A read-only field that represents a bitwise inversion function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции побитовой инверсии числа.</para>
        /// </summary>
        public static readonly Func<T, T> Not = CompileNotDelegate();

        /// <summary>
        /// <para>A read-only field that represents a logic addition function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции логического сложения.</para>
        /// </summary>
        public static readonly Func<T, T, T> Or = CompileOrDelegate();

        /// <summary>
        /// <para>A read-only field that represents a logic multiplication function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции логического умножения.</para>
        /// </summary>
        public static readonly Func<T, T, T> And = CompileAndDelegate();

        /// <summary>
        /// <para>A read-only field that represents a bitwise left shift function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции побитового сдвига числа влево.</para>
        /// </summary>
        public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();

        /// <summary>
        /// <para>A read-only field that represents a bitwise right shift function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции побитового сдвига числа вправо.</para>
        /// </summary>
        public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();

        /// <summary>
        /// <para>A read-only field that represents a bitwise number representation partial rewrite function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции частичной побитовой перезаписи представления числа.</para>
        /// </summary>
        public static readonly Func<T, T, int, int, T> PartialWrite = CompilePartialWriteDelegate();

        /// <summary>
        /// <para>A read-only field that represents a bitwise number representation partial read function delegate.</para>
        /// <para>Поле только для чтения, представляющее делегат функции частичного побитового считывания числа.</para>
        /// </summary>
        public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();

        /// <summary>
        /// <para>
        /// Compiles the not delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileNotDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0);
                emiter.Not();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the or delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileOrDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Or();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the and delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, T> CompileAndDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.And();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the shift left delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and int and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, T> CompileShiftLeftDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.ShiftLeft();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the shift right delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and int and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, T> CompileShiftRightDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.ShiftRight<T>();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the partial write delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t and int and int and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                var constants = GetConstants();
                var bitsNumber = constants.Item1;
                var numberFilledWithOnes = constants.Item2;
                ushort shiftArgument = 2;
                ushort limitArgument = 3;
                var checkLimit = emiter.DefineLabel();
                var calculateSourceMask = emiter.DefineLabel();
                // Check shift
                emiter.LoadArgument(shiftArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
                // Fix shift
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(shiftArgument);
                emiter.Add();
                emiter.StoreArgument(shiftArgument);
                emiter.MarkLabel(checkLimit);
                // Check limit
                emiter.LoadArgument(limitArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                // Fix limit
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(limitArgument);
                emiter.Add();
                emiter.StoreArgument(limitArgument);
                emiter.MarkLabel(calculateSourceMask);
                var sourceMask = emiter.DeclareLocal<T>();
                var targetMask = emiter.DeclareLocal<T>();
                emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                emiter.LoadArgument(limitArgument);
                emiter.ShiftLeft();
                emiter.Not();
                emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                emiter.And();
                emiter.StoreLocal(sourceMask);
                emiter.LoadLocal(sourceMask);
                emiter.LoadArgument(shiftArgument);
                emiter.ShiftLeft();
                emiter.Not();
                emiter.StoreLocal(targetMask);
                emiter.LoadArgument(0); // target
                emiter.LoadLocal(targetMask);
                emiter.And();
                emiter.LoadArgument(1); // source
                emiter.LoadLocal(sourceMask);
                emiter.And();
                emiter.LoadArgument(shiftArgument);
                emiter.ShiftLeft();
                emiter.Or();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the partial read delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and int and int and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, int, T> CompilePartialReadDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                var constants = GetConstants();
                var bitsNumber = constants.Item1;
                var numberFilledWithOnes = constants.Item2;
                ushort shiftArgument = 1;
                ushort limitArgument = 2;
                var checkLimit = emiter.DefineLabel();
                var calculateSourceMask = emiter.DefineLabel();
                // Check shift
                emiter.LoadArgument(shiftArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
                // Fix shift
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(shiftArgument);
                emiter.Add();
                emiter.StoreArgument(shiftArgument);
                emiter.MarkLabel(checkLimit);
                // Check limit
                emiter.LoadArgument(limitArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                // Fix limit
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(limitArgument);
                emiter.Add();
                emiter.StoreArgument(limitArgument);
                emiter.MarkLabel(calculateSourceMask);
                var sourceMask = emiter.DeclareLocal<T>();
                var targetMask = emiter.DeclareLocal<T>();
                emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                emiter.LoadArgument(limitArgument); // limit
                emiter.ShiftLeft();
                emiter.Not();
                emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                emiter.And();
                emiter.StoreLocal(sourceMask);
                emiter.LoadLocal(sourceMask);
                emiter.LoadArgument(shiftArgument);
                emiter.ShiftLeft();
                emiter.StoreLocal(targetMask);
                emiter.LoadArgument(0); // target
                emiter.LoadLocal(targetMask);
                emiter.And();
                emiter.LoadArgument(shiftArgument);
                emiter.ShiftRight<T>();
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Gets the constants.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="NotSupportedException">
        /// <para></para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>A tuple of int and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Tuple<int, T> GetConstants()
        {
            var type = typeof(T);
            if (type == typeof(ulong))
            {
                return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
            }
            if (type == typeof(uint))
            {
                return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
            }
            if (type == typeof(ushort))
            {
                return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
            }
            if (type == typeof(byte))
            {
                return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
            }
            throw new NotSupportedException();
        }
    }
}
```

1.7 ./csharp/Platform.Numbers/Math.cs

```
using System;
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of math methods.</para>
    /// <para>Представляет набор математических методов.</para>
    /// </summary>
    /// <remarks>Resizable array (FileMappedMemory) for values cache may be used. or cached
     ↪  oeis.org</remarks>
    public static class Math
    {
        private static readonly ulong[] _factorials =
        {
            1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
            479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
            355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
        };

        private static readonly ulong[] _catalans =
        {
            1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
            742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
            6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
            18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304,
            14544636039226909, 55534064877048198, 212336130412243110, 812944042149730764,
             ↪  3116285494907301262, 11959798385860453492
        };

        /// <summary>
        /// <para>Represents the limit for calculating the catanal number, supported by the <see
         ↪  cref="ulong"/> type.</para>
        /// <para>Представляет предел расчёта катаналового числа, поддерживаемый <see
         ↪  cref="ulong"/> типом.</para>
        /// </summary>
        public static readonly ulong MaximumFactorialNumber = 20;

        /// <summary>
        /// <para>Represents the limit for calculating the factorial number, supported by the
         ↪  <see cref="ulong"/> type.</para>
        /// <para>Представляет предел расчёта факториала числа, поддерживаемый <see
         ↪  cref="ulong"/> типом.</para>
        /// </summary>
        public static readonly ulong MaximumCatalanIndex = 36;

        /// <summary>
        /// <para>Returns the product of all positive integers less than or equal to the number specified as an argument.</para>
        /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному в качестве аргумента числу.</para>
        /// </summary>
        /// <param name="n">
        /// <para>The maximum positive number that will participate in factorial's product.</para>
        /// <para>Максимальное положительное число, которое будет участвовать в произведении факториала.</para>
        /// </param>
        /// <returns>
        /// <para>The product of all positive integers less than or equal to the number specified as an argument.</para>
        /// <para>Произведение всех положительных чисел меньше или равных указанному, в качестве аргумента, числу.</para>
        /// </returns>
        public static ulong Factorial(ulong n)
        {
            if (n >= 0 && n <= MaximumFactorialNumber)
            {
                return _factorials[n];
            }
            else
            {
                throw new ArgumentOutOfRangeException($"Only numbers from 0 to {MaximumFactorialNumber} are supported by unsigned integer with 64 bits length.");
            }
        }

        /// <summary>
        /// <para>Returns the Catalan Number with the number specified as an argument.</para>
        /// <para>Возвращает Число Катанала с номером, указанным в качестве аргумента.</para>
        /// </summary>
        /// <param name="n">
        /// <para>The number of the Catalan number.</para>
        /// <para>Номер Числа Катанала.</para>
        /// </param>
        /// <returns>
        /// <para>The Catalan Number with the number specified as an argument.</para>
        /// <para>Число Катанала с номером, указанным в качестве аргумента.</para>
        /// </returns>
        public static ulong Catalan(ulong n)
        {
            if (n >= 0 && n <= MaximumCatalanIndex)
            {
                return _catalans[n];
            }
            else
            {
                throw new ArgumentOutOfRangeException($"Only numbers from 0 to {MaximumCatalanIndex} are supported by unsigned integer with 64 bits length.");
            }
        }

        /// <summary>
        /// <para>Checks if a number is a power of two.</para>
        /// <para>Проверяет, является ли число степенью двойки.</para>
        /// </summary>
        /// <param name="x">
        /// <para>The number to check.</para>
        /// <para>Число для проверки.</para>
        /// </param>
        /// <returns>
        /// <para>True if the number is a power of two otherwise false.</para>
        /// <para>True, если число является степенью двойки, иначе - false.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;

        /// <summary>
        /// <para>Takes a module from a number.</para>
        /// <para>Берёт модуль от числа.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number from which to take the absolute value.</para>
        /// <para>Число, от которого необходимо взять абсолютное значение.</para>
        /// </param>
        /// <returns>
        /// <para>The absolute value of the number.</para>
        /// <para>Абсолютное значение числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Abs<T>(T x) => Math<T>.Abs(x);

        /// <summary>
        /// <para>Makes a number negative.</para>
        /// <para>Делает число отрицательным.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to be made negative.</para>
        /// <para>Число которое нужно сделать отрицательным.</para>
        /// </param>
        /// <returns>
        /// <para>A negative number.</para>
        /// <para>Отрицательное число.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Negate<T>(T x) => Math<T>.Negate(x);
    }
}
```

1.8 ./csharp/Platform.Numbers/MathExtensions.cs

```
using System.Runtime.CompilerServices;

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of extension methods that perform mathematical operations on arbitrary object types.</para>
    /// <para>Представляет набор методов расширения выполняющих математические операции для объектов произвольного типа.</para>
    /// </summary>
    public static class MathExtensions
    {
        /// <summary>
        /// <para>Takes a module from a number.</para>
        /// <para>Берёт модуль от числа.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number from which to take the absolute value.</para>
        /// <para>Число от которого необходимо взять абсолютное значение.</para>
        /// </param>
        /// <returns>
        /// <para>The absolute value of a number.</para>
        /// <para>Абсолютное значение числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);

        /// <summary>
        /// <para>Makes a number negative.</para>
        /// <para>Делает число отрицательным.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to be made negative.</para>
        /// <para>Число которое нужно сделать отрицательным.</para>
        /// </param>
        /// <returns>
        /// <para>Negative number.</para>
        /// <para>Отрицательное число.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
    }
}
```

1.9 ./csharp/Platform.Numbers/Math[T].cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType
namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of compiled math operations delegates.</para>
    /// <para>Представляет набор скомпилированных делегатов математических операций.</para>
    /// </summary>
    public static class Math<T>
    {
        /// <summary>
        /// <para>A read-only field that represents a number modulus calculation function
        ↪  delegate.</para>
        /// <para>Поле только для чтения, которое представляет делегат функции вычисления модуля
        ↪  числа.</para>
        /// </summary>
        public static readonly Func<T, T> Abs = CompileAbsDelegate();

        /// <summary>
        /// <para>A read-only field that represents a number negation function delegate.</para>
        /// <para>Поле только для чтения, которое представляет делегат функции отрицания
        ↪  числа.</para>
        /// </summary>
        public static readonly Func<T, T> Negate = CompileNegateDelegate();

        /// <summary>
        /// <para>
        /// Compiles the abs delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileAbsDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                if (NumericType<T>.IsSigned)
                {
                    emiter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
                }
                emiter.Return();
            });
        }

        /// <summary>
        /// <para>
        /// Compiles the negate delegate.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>A func of t and t</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileNegateDelegate()
```

```csharp
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                emiter.LoadArgument(0);
                emiter.Negate();
                emiter.Return();
            });
        }
    }
}
```

1.10 ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```csharp
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the arithmetic extensions tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ArithmeticExtensionsTests
    {
        /// <summary>
        /// <para>
        /// Tests that increment test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void IncrementTest()
        {
            var number = 0UL;
            var returnValue = number.Increment();
            Assert.Equal(1UL, returnValue);
            Assert.Equal(1UL, number);
        }

        /// <summary>
        /// <para>
        /// Tests that decrement test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void DecrementTest()
        {
            var number = 1UL;
            var returnValue = number.Decrement();
            Assert.Equal(0UL, returnValue);
            Assert.Equal(0UL, number);
        }
    }
}
```

1.11 ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs

```csharp
using System;
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the arithmetic tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class ArithmeticTests
    {
        /// <summary>
        /// <para>
        /// Tests that compiled operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CompiledOperationsTest()
        {
            Assert.Equal(3, Arithmetic.Add(1, 2));
            Assert.Equal(1, Arithmetic.Subtract(2, 1));
            Assert.Equal(8, Arithmetic.Multiply(2, 4));
            Assert.Equal(4, Arithmetic.Divide(8, 2));
            Assert.Equal(2, Arithmetic.Increment(1));
            Assert.Equal(1UL, Arithmetic.Decrement(2UL));
            Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
        }
    }
}
```

1.12 ./csharp/Platform.Numbers.Tests/BitTests.cs

```
using System;
using Platform.Reflection;
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the bit tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class BitTests
    {
        /// <summary>
        /// <para>
        /// Tests that get lowest bit position test.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="value">
        /// <para>The value.</para>
        /// <para></para>
        /// </param>
        /// <param name="expectedPosition">
        /// <para>The expected position.</para>
        /// <para></para>
        /// </param>
        [Theory]
        [InlineData(00, -1)] // 0000 0000 (none, -1)
        [InlineData(01, 00)] // 0000 0001 (first, 0)
        [InlineData(08, 03)] // 0000 1000 (forth, 3)
        [InlineData(88, 03)] // 0101 1000 (forth, 3)
        public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
        {
            Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
        }

        /// <summary>
        /// <para>
        /// Tests that byte bitwise operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void ByteBitwiseOperationsTest()
        {
            Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
            Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
            Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
            Assert.Equal(NumericType<byte>.MaxValue >> 1,
             ↪  Bit<byte>.ShiftRight(NumericType<byte>.MaxValue, 1));
        }

        /// <summary>
        /// <para>
        /// Tests that u int 16 bitwise operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void UInt16BitwiseOperationsTest()
        {
            Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
            Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
            Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
            Assert.Equal(NumericType<ushort>.MaxValue >> 1,
             ↪  Bit<ushort>.ShiftRight(NumericType<ushort>.MaxValue, 1));
        }

        /// <summary>
        /// <para>
        /// Tests that u int 32 bitwise operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void UInt32BitwiseOperationsTest()
        {
            Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
            Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
            Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
            Assert.Equal(NumericType<uint>.MaxValue >> 1,
             ↪  Bit<uint>.ShiftRight(NumericType<uint>.MaxValue, 1));
        }

        /// <summary>
        /// <para>
        /// Tests that u int 64 bitwise operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void UInt64BitwiseOperationsTest()
        {
            Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
            Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
            Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
            Assert.Equal(NumericType<ulong>.MaxValue >> 1,
             ↪  Bit<ulong>.ShiftRight(NumericType<ulong>.MaxValue, 1));
        }

        /// <summary>
        /// <para>
        /// Tests that partial read write test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void PartialReadWriteTest()
        {
            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
                uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                Assert.True(firstValue == unpackagedFirstValue);
                Assert.True(secondValue == unpackagedSecondValue);

                // Using universal functions:
                Assert.True(PartialRead(value, 0, 1) == firstValue);
                Assert.True(PartialRead(value, 1, -1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = PartialWrite(value, firstValue, 0, 1);
                value = PartialWrite(value, secondValue, 1, -1);

                Assert.True(PartialRead(value, 0, 1) == firstValue);
                Assert.True(PartialRead(value, 1, -1) == secondValue);
            }

            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
                uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                Assert.True(firstValue == unpackagedFirstValue);
                Assert.True(secondValue == unpackagedSecondValue);

                // Using universal functions:
                Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
                Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = Bit.PartialWrite(value, firstValue, 0, 1);
                value = Bit.PartialWrite(value, secondValue, 1, -1);

                Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
                Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
            }

            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
                uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                Assert.True(firstValue == unpackagedFirstValue);
                Assert.True(secondValue == unpackagedSecondValue);

                // Using universal functions:
                var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
                var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
                var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
                var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);

                Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
                value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);

                Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
            }
        }

        // TODO: Can be optimized using precalculation of TargetMask and SourceMask
        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return target & targetMask | (source & sourceMask) << shift;
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }

        private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
        }

        private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return new Tuple<uint, int>(targetMask, shift);
        }

        private static uint PartialWrite(uint target, uint targetMask, uint source, uint sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;

        private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int> masksAndShift) => PartialWrite(target, masksAndShift.Item1, source, masksAndShift.Item2, masksAndShift.Item3);

        private static uint PartialRead(uint target, uint targetMask, int shift) => (target & targetMask) >> shift;

        private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) => PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);

        /// <summary>
        /// <para>
        /// Tests that bug with loading constant of 8 test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void BugWithLoadingConstantOf8Test()
        {
            Bit<byte>.PartialWrite(0, 1, 5, -5);
        }
    }
}
```

1.13 ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs

```
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the math extensions tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class MathExtensionsTests
    {
        /// <summary>
        /// <para>
        /// Tests that abs test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void AbsTest()
        {
            var number = -1L;
            var returnValue = number.Abs();
            Assert.Equal(1L, returnValue);
            Assert.Equal(1L, number);
        }

        /// <summary>
        /// <para>
        /// Tests that negate test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void NegateTest()
        {
            var number = 2L;
            var returnValue = number.Negate();
            Assert.Equal(-2L, returnValue);
            Assert.Equal(-2L, number);
        }

        /// <summary>
        /// <para>
        /// Tests that unsigned negate test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void UnsignedNegateTest()
        {
            var number = 2UL;
            var returnValue = number.Negate();
            Assert.Equal(18446744073709551614, returnValue);
            Assert.Equal(18446744073709551614, number);
        }
    }
}
```

1.14 ./csharp/Platform.Numbers.Tests/MathTests.cs

```
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the math tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class MathTests
    {
        /// <summary>
        /// <para>
        /// Tests that compiled operations test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void CompiledOperationsTest()
        {
            Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
        }
    }
}
```

1.15 ./csharp/Platform.Numbers.Tests/SystemTests.cs

```
using Xunit;

namespace Platform.Numbers.Tests
{
    /// <summary>
    /// <para>
    /// Represents the system tests.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class SystemTests
    {
        /// <summary>
        /// <para>
        /// Tests that possible pack two values into one test.
        /// </para>
        /// <para></para>
        /// </summary>
        [Fact]
        public static void PossiblePackTwoValuesIntoOneTest()
        {
            uint value = 0;

            // Set one to first bit
            value |= 1;

            Assert.True(value == 1);

            // Set zero to first bit
            value &= 0xFFFFFFFE;

            // Get first bit
            uint read = value & 1;

            Assert.True(read == 0);

            uint firstValue = 1;
            uint secondValue = 1543;

            // Pack (join) two values at the same time
            value = (secondValue << 1) | firstValue;

            uint unpackagedFirstValue = value & 1;
            uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

            Assert.True(firstValue == unpackagedFirstValue);
            Assert.True(secondValue == unpackagedSecondValue);

            // Using universal functions:

            Assert.True(PartialRead(value, 0, 1) == firstValue);
            Assert.True(PartialRead(value, 1, -1) == secondValue);

            firstValue = 0;
            secondValue = 6892;

            value = PartialWrite(value, firstValue, 0, 1);
            value = PartialWrite(value, secondValue, 1, -1);

            Assert.True(PartialRead(value, 0, 1) == firstValue);
            Assert.True(PartialRead(value, 1, -1) == secondValue);
        }

        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return (target & targetMask) | ((source & sourceMask) << shift);
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }
    }
}
```

Index