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

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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>
```

```
77          /// <para>Произведение x и y.</para>
78          /// </returns>
79          [MethodImpl(MethodImplOptions.AggressiveInlining)]
80          public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);
81
82          /// <summary>
83          /// <para>Performs dividing x by y.</para>
84          /// <para>Выполняет деление x на y.</para>
85          /// </summary>
86          /// <typeparam name="T">
87          /// <para>The numbers type.</para>
88          /// <para>Тип чисел.</para>
89          /// </typeparam>
90          /// <param name="x">
91          /// <para>Dividend.</para>
92          /// <para>Делимое.</para>
93          /// </param>
94          /// <param name="y">
95          /// <para>Divider.</para>
96          /// <para>Делитель.</para>
97          /// </param>
98          /// <returns>
99          /// <para>Quoitent of x and y.</para>
100         /// <para>Частное x и y.</para>
101         /// </returns>
102         [MethodImpl(MethodImplOptions.AggressiveInlining)]
103         public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);
104
105         /// <summary>
106         /// <para>Increasing the number by one.</para>
107         /// <para>Увеличивает число на единицу.</para>
108         /// </summary>
109         /// <typeparam name="T">
110         /// <para>The number type.</para>
111         /// <para>Тип числа.</para>
112         /// </typeparam>
113         /// <param name="x">
114         /// <para>The number to increase.</para>
115         /// <para>Число для увеличения.</para>
116         /// </param>
117         /// <returns>
118         /// <para>Increase by one number.</para>
119         /// <para>Увеличенное на единицу число.</para>
120         /// </returns>
121         [MethodImpl(MethodImplOptions.AggressiveInlining)]
122         public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
123
124         /// <summary>
125         /// <para>Increases the value of argument by one.</para>
126         /// <para>Увеличивает значение аргумента на единицу.</para>
127         /// </summary>
128         /// <typeparam name="T">
129         /// <para>The number type.</para>
130         /// <para>Тип числа.</para>
131         /// </typeparam>
132         /// <param name="x">
133         /// <para>The argument to increase.</para>
134         /// <para>Аргумент для увеличения.</para>
135         /// </param>
136         /// <returns>
137         /// <para>Increased argument value.</para>
138         /// <para>Увеличенное значение аргумента.</para>
139         /// </returns>
140         [MethodImpl(MethodImplOptions.AggressiveInlining)]
141         public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);
142
143         /// <summary>
144         /// <para>Decreases number by one.</para>
145         /// <para>Уменьшение числа на единицу.</para>
146         /// </summary>
147         /// <typeparam name="T">
148         /// <para>The number type.</para>
149         /// <para>Тип числа.</para>
150         /// </typeparam>
151         /// <param name="x">
152         /// <para>The number to reduce.</para>
153         /// <para>Число для уменьшения.</para>
154         /// </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 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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>.</para>
    /// <para>.</para>
    /// </summary>
    public static class ArithmeticExtensions
    {
        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Decrement(x);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
        public static T Increment<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Increment(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
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>.</para>
    /// <para>.</para>
    /// </summary>
    public static class Arithmetic<T>
    {
        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T, T> Add = CompileAddDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T, T> Divide = CompileDivideDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T> Increment = CompileIncrementDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T> Decrement = CompileDecrementDelegate();

        [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();
            });
        }

        [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();
            });
        }

        [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();
            });
        }

        [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();
            });
        }

        [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();
            });
        }

        [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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 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">
```

```
97          /// <para>Second term.</para>
98          /// <para>Второе слагаемое.</para>
99          /// </param>
100         /// <returns>
101         /// <para>The logical sum of numbers</para>
102         /// <para>Логическая сумма чисел.</para>
103         /// </returns>
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
106
107         /// <summary>
108         /// <para>Performs bitwise numbers multiplication.</para>
109         /// <para>Выполняет побитовое умножение чисел.</para>
110         /// <typeparam name="T">
111         /// <para>The numbers type.</para>
112         /// <para>Тип чисел.</para>
113         /// </typeparam>
114         /// </summary>
115         /// <param name="x">
116         /// <para>First multiplier.</para>
117         /// <para>Первый множитель.</para>
118         /// </param>
119         /// <param name="y">
120         /// <para>Second multiplier.</para>
121         /// <para>Второй множитель.</para>
122         /// </param>
123         /// <returns>
124         /// <para>Logical product of numbers.</para>
125         /// <para>Логическое произведение чисел.</para>
126         /// </returns>
127         [MethodImpl(MethodImplOptions.AggressiveInlining)]
128         public static T And<T>(T x, T y) => Bit<T>.And(x, y);
129
130         /// <summary>
131         /// <para>Performs a bitwise shift of a number to the left by the specified number of
            ↪ bits.</para>
132         /// <para>Выполняет побитовый свиг числа влево на указанное количество бит.</para>
133         /// </summary>
134         /// <typeparam name="T">
135         /// <para>The number type.</para>
136         /// <para>Тип числа.</para>
137         /// </typeparam>
138         /// <param name="x">
139         /// <para>The number on which the left bitwise shift operation will be performed.</para>
140         /// <para>Число над которым будет производиться операция пиботового смещения
            ↪ влево.</para>
141         /// </param>
142         /// <param name="y">
143         /// <para>The number of bits to shift.</para>
144         /// <para>Количество бит на которые выполнить смещение.</para>
145         /// </param>
146         /// <returns>
147         /// <para>The value with discarded high-order bits that are outside the range of the
            ↪ number type and set low-order empty bit positions to zero.</para>
148         /// <para>Значение с отброшенными старшими битами, которые находятся за пределами
            ↪ диапазона типа числа и устанавливленными пустыми битовыми позициями младших разрядов
            ↪ в ноль.</para>
149         /// </returns>
150         [MethodImpl(MethodImplOptions.AggressiveInlining)]
151         public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
152
153         /// <summary>
154         /// <para>Performs a bitwise shift of a number to the right by the specified number of
            ↪ bits.</para>
155         /// <para>Выполняет побитовый свиг числа вправо на указанное количество бит.</para>
156         /// </summary>
157         /// <typeparam name="T">
158         /// <para>The number type.</para>
159         /// <para>Тип числа.</para>
160         /// </typeparam>
161         /// <param name="x">
162         /// <para>The number on which the right bitwise shift operation will be performed.</para>
163         /// <para>Число над которым будет производиться операция побитового смещения
            ↪ вправо.</para>
164         /// </param>
165         /// <param name="y">
166         /// <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 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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```
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>.</para>
        /// <para>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <param name="source">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</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>.</para>
        /// <para>.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>.</para>
        /// <para>.</para>
        /// </param>
        /// <returns>
        /// <para>.</para>
        /// <para>.</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;
```

```
5
6   // ReSharper disable StaticFieldInGenericType
7   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9   namespace Platform.Numbers
10  {
11      /// <summary>
12      /// <para>.</para>
13      /// <para>.</para>
14      /// </summary>
15      public static class Bit<T>
16      {
17          /// <summary>
18          /// <para>.</para>
19          /// <para>.</para>
20          /// </summary>
21          public static readonly Func<T, T> Not = CompileNotDelegate();
22
23          /// <summary>
24          /// <para>.</para>
25          /// <para>.</para>
26          /// </summary>
27          public static readonly Func<T, T, T> Or = CompileOrDelegate();
28
29          /// <summary>
30          /// <para>.</para>
31          /// <para>.</para>
32          /// </summary>
33          public static readonly Func<T, T, T> And = CompileAndDelegate();
34
35          /// <summary>
36          /// <para>.</para>
37          /// <para>.</para>
38          /// </summary>
39          public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
40
41          /// <summary>
42          /// <para>.</para>
43          /// <para>.</para>
44          /// </summary>
45          public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
46
47          /// <summary>
48          /// <para>.</para>
49          /// <para>.</para>
50          /// </summary>
51          public static readonly Func<T, T, int, int, T> PartialWrite =
            ↪  CompilePartialWriteDelegate();
52
53          /// <summary>
54          /// <para>.</para>
55          /// <para>.</para>
56          /// </summary>
57          public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
58
59          [MethodImpl(MethodImplOptions.AggressiveInlining)]
60          private static Func<T, T> CompileNotDelegate()
61          {
62              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
63              {
64                  Ensure.Always.IsNumeric<T>();
65                  emiter.LoadArguments(0);
66                  emiter.Not();
67                  emiter.Return();
68              });
69          }
70
71          [MethodImpl(MethodImplOptions.AggressiveInlining)]
72          private static Func<T, T, T> CompileOrDelegate()
73          {
74              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
75              {
76                  Ensure.Always.IsNumeric<T>();
77                  emiter.LoadArguments(0, 1);
78                  emiter.Or();
79                  emiter.Return();
80              });
81          }
82
```

```
        [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();
            });
        }

        [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();
            });
        }

        [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();
                emiter.Return();
            });
        }

        [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();
            });
        }

        [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();
                emiter.Return();
            });
        }

        [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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a collection of algebraic methods.</para>
    /// <para>Представляет набор алгебраических методов.</para>
    /// </summary>
    /// <remarks>Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org</remarks>
    public static class Math
    {
        /// <remarks>
        /// <para>Source: https://oeis.org/A000142/list </para>
        /// <para>Источник: https://oeis.org/A000142/list </para>
        /// </remarks>
        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
        };

        /// <remarks>
        /// <para>Source: https://oeis.org/A000108/list </para>
        /// <para>Источник: https://oeis.org/A000108/list </para>
        /// </remarks>
        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>
```

```
52          /// <para>Returns the product of all positive integers less than or equal to the number
            ↪  specified as an argument.</para>
53          /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
            ↪  в качестве аргумента числу.</para>
54          /// </summary>
55          /// <param name="n">
56          /// <para>The maximum positive number that will participate in factorial's
            ↪  product.</para>
57          /// <para>Максимальное положительное число, которое будет участвовать в произведении
            ↪  факториала.</para>
58          /// </param>
59          /// <returns>
60          /// <para>The product of all positive integers less than or equal to the number
            ↪  specified as an argument.</para>
61          /// <para>Произведение всех положительных чисел меньше или равных указанному, в качестве
            ↪  аргумента, числу.</para>
62          /// </returns>
63          public static ulong Factorial(ulong n)
64          {
65              if (n >= 0 && n <= MaximumFactorialNumber)
66              {
67                  return _factorials[n];
68              }
69              else
70              {
71                  throw new ArgumentOutOfRangeException($"Only numbers from 0 to
                    ↪  {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
                    ↪  length.");
72              }
73          }
74
75          /// <summary>
76          /// <para>Returns the Catalan Number with the number specified as an argument.</para>
77          /// <para>Возвращает Число Катанала с номером, указанным в качестве аргумента.</para>
78          /// </summary>
79          /// <param name="n">
80          /// <para>The number of the Catalan number.</para>
81          /// <para>Номер Числа Катанала.</para>
82          /// </param>
83          /// <returns>
84          /// <para>The Catalan Number with the number specified as an argument.</para>
85          /// <para>Число Катанала с номером, указанным в качестве аргумента.</para>
86          /// </returns>
87          public static ulong Catalan(ulong n)
88          {
89              if (n >= 0 && n <= MaximumCatalanIndex)
90              {
91                  return _catalans[n];
92              }
93              else
94              {
95                  throw new ArgumentOutOfRangeException($"Only numbers from 0 to
                    ↪  {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
                    ↪  length.");
96              }
97          }
98
99          /// <summary>
100         /// <para>Checks if a number is a power of two.</para>
101         /// <para>Проверяет, является ли число степенью двойки.</para>
102         /// </summary>
103         /// <param name="x">
104         /// <para>The number to check.</para>
105         /// <para>Число для проверки.</para>
106         /// </param>
107         /// <returns>
108         /// <para>True if the number is a power of two otherwise false.</para>
109         /// <para>True, если число является степенью двойки, иначе - false.</para>
110         /// </returns>
111         [MethodImpl(MethodImplOptions.AggressiveInlining)]
112         public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
113
114         /// <summary>
115         /// <para>Takes a module from a number.</para>
116         /// <para>Берёт модуль от числа.</para>
117         /// </summary>
118         /// <typeparam name="T">
```

```
        /// <para>The number 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 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;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Provides a set of extension methods that perform mathematical operations on
    ↪  arbitrary object types.</para>
    /// <para>Предоставляет 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 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 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
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>.</para>
    /// <para>.</para>
    /// </summary>
    public static class Math<T>
    {
        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T> Abs = CompileAbsDelegate();

        /// <summary>
        /// <para>.</para>
        /// <para>.</para>
        /// </summary>
        public static readonly Func<T, T> Negate = CompileNegateDelegate();

        [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();
            });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, T> CompileNegateDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                emiter.LoadArgument(0);
                emiter.Negate();
                emiter.Return();
            });
        }
    }
}
```

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

```
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class ArithmeticExtensionsTests
    {
        [Fact]
        public static void IncrementTest()
        {
```

```
            var number = 0UL;
            var returnValue = number.Increment();
            Assert.Equal(1UL, returnValue);
            Assert.Equal(1UL, number);
        }

        [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

```
using System;
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class ArithmeticTests
    {
        [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 Xunit;

namespace Platform.Numbers.Tests
{
    public static class BitTests
    {
        [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);
        }

        [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));
        }

        [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));
        }

        [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));
        }

        [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));
        }

        [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
```

```
120                 uint value = secondValue << 1 | firstValue;
121
122                 uint unpackagedFirstValue = value & 1;
123                 uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
124
125                 Assert.True(firstValue == unpackagedFirstValue);
126                 Assert.True(secondValue == unpackagedSecondValue);
127
128                 // Using universal functions:
129                 var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
130                 var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
131                 var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
132                 var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
133
134                 Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
135                 Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
136
137                 firstValue = 0;
138                 secondValue = 6892;
139
140                 value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
141                 value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
142
143                 Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
144                 Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
145             }
146         }
147
148         // TODO: Can be optimized using precalculation of TargetMask and SourceMask
149         private static uint PartialWrite(uint target, uint source, int shift, int limit)
150         {
151             if (shift < 0)
152             {
153                 shift = 32 + shift;
154             }
155             if (limit < 0)
156             {
157                 limit = 32 + limit;
158             }
159             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
160             var targetMask = ~(sourceMask << shift);
161             return target & targetMask | (source & sourceMask) << shift;
162         }
163
164         private static uint PartialRead(uint target, int shift, int limit)
165         {
166             if (shift < 0)
167             {
168                 shift = 32 + shift;
169             }
170             if (limit < 0)
171             {
172                 limit = 32 + limit;
173             }
174             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
175             var targetMask = sourceMask << shift;
176             return (target & targetMask) >> shift;
177         }
178
179         private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
180         {
181             if (shift < 0)
182             {
183                 shift = 32 + shift;
184             }
185             if (limit < 0)
186             {
187                 limit = 32 + limit;
188             }
189             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
190             var targetMask = ~(sourceMask << shift);
191             return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
192         }
193
194         private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
195         {
196             if (shift < 0)
197             {
198                 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);

        [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
{
    public static class MathExtensionsTests
    {
        [Fact]
        public static void AbsTest()
        {
            var number = -1L;
            var returnValue = number.Abs();
            Assert.Equal(1L, returnValue);
            Assert.Equal(1L, number);
        }

        [Fact]
        public static void NegateTest()
        {
            var number = 2L;
            var returnValue = number.Negate();
            Assert.Equal(-2L, returnValue);
            Assert.Equal(-2L, number);
        }

        [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
{
    public static class MathTests
    {
        [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
{
    public static class SystemTests
    {
        [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;
            }
```

```
77              var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
78              var targetMask = sourceMask << shift;
79              return (target & targetMask) >> shift;
80          }
81      }
82  }
```

Index