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> Each function performs the simplest arithmetic operation on the type specified as a parameter </para>
    /// <para> Каждая функция выполняет простейшую арифметическую операцию над типом, указанным в качестве параметра </para>
    /// </summary>
    public static class Arithmetic
    {

        /// <summary>
        /// <para> Adding the x and y arguments </para>
        /// <para> Сложение аргументов x и y </para>
        /// </summary>
        /// <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> Subtracting x from y </para>
        /// <para> Вычитание x из y </para>
        /// </summary>
        /// <returns>
        /// <para> Difference of x and y </para>
        /// <para> Разность x и y </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);

        /// <summary>
        /// <para> Multiplication the x and y </para>
        /// <para> Умножение x и y </para>
        /// </summary>
        /// <returns>
        /// <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> Dividing x by y </para>
        /// <para> Деление x на y </para>
        /// </summary>
        /// <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 parameter x by one </para>
        /// <para> Увеличение параметра x на единицу </para>
        /// </summary>
        /// <returns>
        /// <para> Increase by one x </para>
        /// <para> Увеличенное на единицу x </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para> Increase the parameter x passed by reference by one </para>
        /// <para> Увеличение переданного по ссылке параметра x на единицу </para>
        /// </summary>
        /// <returns>
```

```
        /// <para> Increase by one x with returning the value to the original variable </para>
        /// <para> Увеличенное на единицу x с возвратом значения в исходную переменную </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para> Decrease parameter x by one </para>
        /// <para> Уменьшение параметра x на единицу </para>
        /// </summary>
        /// <returns>
        /// <para> Reduced by one x </para>
        /// <para> Уменьшенное на единицу x </para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);

        /// <summary>
        /// <para> Decreasing the parameter x passed by reference by one </para>
        /// <para> Уменьшение переданного по ссылке параметра x на единицу </para>
        /// </summary>
        /// <returns>
        /// <para> Reduced by one x with returning the value to the original variable </para>
        /// <para> Уменьшенное на единицу x с воз вратом значения в исходную переменную </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
{
    public static class ArithmeticExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Decrement<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Decrement(x);

        [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
{
    public static class Arithmetic<T>
    {
        public static readonly Func<T, T, T> Add = CompileAddDelegate();
        public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
        public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
        public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
        public static readonly Func<T, T> Increment = CompileIncrementDelegate();
        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> Various operations on the set number bits </para>
    /// <para> Различные операции над установленными ьитами числа </para>
    /// </summary>
    /// <returns>
    /// <para> Data on the established beta numbers </para>
    /// <para> Данные об установленных бетах числа </para>

    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>
        /// <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>
        /// <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>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </para>
        ///</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(T x) => Bit<T>.Not(x);

        ///<summary>
        ///<para>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </para>
        ///</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);

        ///<summary>
        ///<para>  </para>
```

```
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </para>
        ///</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T And<T>(T x, T y) => Bit<T>.And(x, y);

        ///<summary>
        ///<para>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </para>
        ///</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);

        ///<summary>
        ///<para>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </para>
        ///</returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);

        ///<summary>
        ///<para>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </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>  </para>
        ///<para>  </para>
        ///</summary>
        ///<returns>
        ///<para>  </para>
        ///<para>  </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>
    /// <returns> bit operations </returns>
    /// </summary>
    /* bit operations */
    public static class BitwiseExtensions

    {

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);

        [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);

```

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

namespace Platform.Numbers
{
    public static class Bit<T>
    {
        public static readonly Func<T, T> Not = CompileNotDelegate();
        public static readonly Func<T, T, T> Or = CompileOrDelegate();
        public static readonly Func<T, T, T> And = CompileAndDelegate();
        public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
        public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
        public static readonly Func<T, T, int, int, T> PartialWrite =
        ↪  CompilePartialWriteDelegate();
        public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();

        [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();
            });
        }

        [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();
            });
        }

        [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
```

```
{
    /// <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
        };

        public static readonly ulong MaximumFactorialNumber = 20;

        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 произведение 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 Каталановое число с номером указанным в качестве аргумента.</para>
        /// </summary>
        /// <param name="n"><para>The number of Catalan number.</para><para>Номер Каталанового
        ↪  числа.</para></param>
        /// <returns><para>The Catalan Number with the number specified as an
        ↪  argument.</para><para>Каталановое число 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.");
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Abs<T>(T x) => Math<T>.Abs(x);

        [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> Additional mathematical transformations </para>
    /// <para> Дополнительные математические преобразования </para>
    /// </summary>
    /// <returns>
    /// <para> The converted value in the original variable </para>
    /// <para> Преобразованное значение в исходной переменной </para>
    /// </returns>
    public static class MathExtensions
    {
        /// <summary>
        /// <para> Takes a module from a number </para>
        /// <para> Берёт модуль от числа </para>
        /// </summary>
        /// <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 negatory </para>
        /// <para> Делает число отрицательным </para>
        /// </summary>
        /// <returns>
        /// <para> Negatory 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
{
    public static class Math<T>
    {
        public static readonly Func<T, T> Abs = CompileAbsDelegate();
        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;
```

```
3
4  namespace Platform.Numbers.Tests
5  {
6      public static class BitTests
7      {
8          [Theory]
9          [InlineData(00, -1)] // 0000 0000 (none, -1)
10         [InlineData(01, 00)] // 0000 0001 (first, 0)
11         [InlineData(08, 03)] // 0000 1000 (forth, 3)
12         [InlineData(88, 03)] // 0101 1000 (forth, 3)
13         public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
14         {
15             Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
16         }
17
18         [Fact]
19         public static void ByteBitwiseOperationsTest()
20         {
21             Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
22             Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
23             Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
24             Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
25             Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
26         }
27
28         [Fact]
29         public static void UInt16BitwiseOperationsTest()
30         {
31             Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
32             Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
33             Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
34             Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
35             Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
36         }
37
38         [Fact]
39         public static void UInt32BitwiseOperationsTest()
40         {
41             Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
42             Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
43             Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
44             Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
45             Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
46         }
47
48         [Fact]
49         public static void UInt64BitwiseOperationsTest()
50         {
51             Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
52             Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
53             Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
54             Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
55             Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
56         }
57
58         [Fact]
59         public static void PartialReadWriteTest()
60         {
61             {
62                 uint firstValue = 1;
63                 uint secondValue = 1543;
64
65                 // Pack (join) two values at the same time
66                 uint value = secondValue << 1 | firstValue;
67
68                 uint unpackagedFirstValue = value & 1;
69                 uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
70
71                 Assert.True(firstValue == unpackagedFirstValue);
72                 Assert.True(secondValue == unpackagedSecondValue);
73
74                 // Using universal functions:
75                 Assert.True(PartialRead(value, 0, 1) == firstValue);
76                 Assert.True(PartialRead(value, 1, -1) == secondValue);
77
78                 firstValue = 0;
79                 secondValue = 6892;
80
81                 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);

        [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;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }
    }
}
```

Index