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
{
    public static class Arithmetic
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);

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

        [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
{
    public static class Bit
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long Count(long x)
        {
            long n = 0;
            while (x != 0)
            {
                n++;
                x &= x - 1;
            }
            return n;
        }

        [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;
        }

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

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

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

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

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

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
        ↪  Bit<T>.PartialWrite(target, source, shift, limit);

        [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
{
    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)]
```

```
16          public static T PartialRead<T>(this T target, int shift, int limit) =>
            ↪  Bit<T>.PartialRead(target, shift, limit);
17      }
18  }
```

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

```
1   using System;
2   using System.Runtime.CompilerServices;
3   using Platform.Exceptions;
4   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      public static class Bit<T>
12      {
13          public static readonly Func<T, T> Not = CompileNotDelegate();
14          public static readonly Func<T, T, T> Or = CompileOrDelegate();
15          public static readonly Func<T, T, T> And = CompileAndDelegate();
16          public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
17          public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
18          public static readonly Func<T, T, int, int, T> PartialWrite =
            ↪  CompilePartialWriteDelegate();
19          public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
20
21          [MethodImpl(MethodImplOptions.AggressiveInlining)]
22          private static Func<T, T> CompileNotDelegate()
23          {
24              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
25              {
26                  Ensure.Always.IsNumeric<T>();
27                  emiter.LoadArguments(0);
28                  emiter.Not();
29                  emiter.Return();
30              });
31          }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          private static Func<T, T, T> CompileOrDelegate()
35          {
36              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
37              {
38                  Ensure.Always.IsNumeric<T>();
39                  emiter.LoadArguments(0, 1);
40                  emiter.Or();
41                  emiter.Return();
42              });
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          private static Func<T, T, T> CompileAndDelegate()
47          {
48              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
49              {
50                  Ensure.Always.IsNumeric<T>();
51                  emiter.LoadArguments(0, 1);
52                  emiter.And();
53                  emiter.Return();
54              });
55          }
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          private static Func<T, int, T> CompileShiftLeftDelegate()
59          {
60              return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
61              {
62                  Ensure.Always.IsNumeric<T>();
63                  emiter.LoadArguments(0, 1);
64                  emiter.ShiftLeft();
65                  emiter.Return();
66              });
67          }
68
69          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70          private static Func<T, int, T> CompileShiftRightDelegate()
71          {
72              return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
```

```
73              {
74                  Ensure.Always.IsNumeric<T>();
75                  emiter.LoadArguments(0, 1);
76                  emiter.ShiftRight();
77                  emiter.Return();
78              });
79          }
80
81          [MethodImpl(MethodImplOptions.AggressiveInlining)]
82          private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
83          {
84              return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emiter =>
85              {
86                  Ensure.Always.IsNumeric<T>();
87                  var constants = GetConstants();
88                  var bitsNumber = constants.Item1;
89                  var numberFilledWithOnes = constants.Item2;
90                  ushort shiftArgument = 2;
91                  ushort limitArgument = 3;
92                  var checkLimit = emiter.DefineLabel();
93                  var calculateSourceMask = emiter.DefineLabel();
94                  // Check shift
95                  emiter.LoadArgument(shiftArgument);
96                  emiter.LoadConstant(0);
97                  emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
98                  // Fix shift
99                  emiter.LoadConstant(bitsNumber);
100                 emiter.LoadArgument(shiftArgument);
101                 emiter.Add();
102                 emiter.StoreArgument(shiftArgument);
103                 emiter.MarkLabel(checkLimit);
104                 // Check limit
105                 emiter.LoadArgument(limitArgument);
106                 emiter.LoadConstant(0);
107                 emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
108                 // Fix limit
109                 emiter.LoadConstant(bitsNumber);
110                 emiter.LoadArgument(limitArgument);
111                 emiter.Add();
112                 emiter.StoreArgument(limitArgument);
113                 emiter.MarkLabel(calculateSourceMask);
114                 var sourceMask = emiter.DeclareLocal<T>();
115                 var targetMask = emiter.DeclareLocal<T>();
116                 emiter.LoadConstant(typeof(T), numberFilledWithOnes);
117                 emiter.LoadArgument(limitArgument);
118                 emiter.ShiftLeft();
119                 emiter.Not();
120                 emiter.LoadConstant(typeof(T), numberFilledWithOnes);
121                 emiter.And();
122                 emiter.StoreLocal(sourceMask);
123                 emiter.LoadLocal(sourceMask);
124                 emiter.LoadArgument(shiftArgument);
125                 emiter.ShiftLeft();
126                 emiter.Not();
127                 emiter.StoreLocal(targetMask);
128                 emiter.LoadArgument(0); // target
129                 emiter.LoadLocal(targetMask);
130                 emiter.And();
131                 emiter.LoadArgument(1); // source
132                 emiter.LoadLocal(sourceMask);
133                 emiter.And();
134                 emiter.LoadArgument(shiftArgument);
135                 emiter.ShiftLeft();
136                 emiter.Or();
137                 emiter.Return();
138             });
139         }
140
141         [MethodImpl(MethodImplOptions.AggressiveInlining)]
142         private static Func<T, int, int, T> CompilePartialReadDelegate()
143         {
144             return DelegateHelpers.Compile<Func<T, int, int, T>>(emiter =>
145             {
146                 Ensure.Always.IsNumeric<T>();
147                 var constants = GetConstants();
148                 var bitsNumber = constants.Item1;
149                 var numberFilledWithOnes = constants.Item2;
150                 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.Collections.Generic;
using System.Linq;
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 long[] _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 long[] _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
        };

        /// <summary>
        /// <para>Generate the factorial of the value "n".</para>
        /// <para>Генерация факториала из значения переменной "n".</para>
        /// </summary>
        /// <param name="n"><para>Factorial generation value.</para><para>Значение генерации
        ↪  факториала.</para></param>
        /// <returns><para>Result of factorial calculation.</para><para>Результат подсчета
        ↪  факториала</para></returns>
        public static long Factorial(long n)
        {
            if (n <= 1)
            {
                return 1;
            }


            if (n < _factorials.Length)
            {
                return _factorials[n];
            }

            return n * Factorial(n - 1);
        }


        public static long FactorialEratosthenesSieve(long n)
        {
            if (n < 0)
                return 0;
            if (n == 0)
                return 1;
            if (n < _factorials.Length)
                return _factorials[n];

            bool[] u = new bool[n + 1]; // маркеры для решета Эратосфена
            List<Tuple<long, long>> p = new List<Tuple<long, long>>(); // множители и их
            ↪  показатели степеней
            for (int i = 2; i <= n; ++i)
                if (!u[i]) // если i - очередное простое число
                {
                    // считаем показатель степени в разложении
                    long k = n / i;
                    long c = 0;
                    while (k > 0)
                    {
                        c += k;
                        k /= i;
```

```
                    }
                    // запоминаем множитель и его показатель степени
                    p.Add(new Tuple<long, long>(i, c));
                    // просеиваем составные числа через решето
                    int j = 2;
                    while (i * j <= n)
                    {
                        u[i * j] = true;
                        ++j;
                    }
                }
            // вычисляем факториал
            int r = 1;
            for (int i = p.Count() - 1; i >= 0; --i)
                r *= (int)System.Math.Pow(p[i].Item1, p[i].Item2);
            return r;
        }


        /// <summary>
        /// <para>Generating the Catalan Number of the value "n".</para>
        /// <para>Генерация числа Каталана из значения переменной "n".</para>
        /// </summary>
        /// <param name="n"><para>Catalan Number generation value.</para><para>Значение
        ↪  генерации Числа Каталана.</para></param>
        /// <returns><para>Result of Catalan Number calculation.</para><para>Результат подсчета
        ↪  Числа Каталана.</para></returns>
        public static long Catalan(int n)
        {


            if (n <= 1)
            {
                return 1;
            }
            if (n < _catalans.Length)
            {
                return _catalans[(int)n];
            }
            return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
        }

        [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
{
    public static class MathExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);

        [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;

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
            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;
        }
```

```
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;
199             }
200             if (limit < 0)
201             {
202                 limit = 32 + limit;
203             }
204             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
205             var targetMask = sourceMask << shift;
206             return new Tuple<uint, int>(targetMask, shift);
207         }
208 
209         private static uint PartialWrite(uint target, uint targetMask, uint source, uint
            ↪  sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
210 
211         private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
            ↪  masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
            ↪  masksAndShift.Item2, masksAndShift.Item3);
212 
213         private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
            ↪  targetMask) >> shift;
214 
215         private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
            ↪  PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
216 
217         [Fact]
218         public static void BugWithLoadingConstantOf8Test()
219         {
220             Bit<byte>.PartialWrite(0, 1, 5, -5);
221         }
222     }
223 }
```

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

```
1  using Xunit;
2 
3  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;
```

```
33
34              Assert.True(firstValue == unpackagedFirstValue);
35              Assert.True(secondValue == unpackagedSecondValue);
36
37              // Using universal functions:
38
39              Assert.True(PartialRead(value, 0, 1) == firstValue);
40              Assert.True(PartialRead(value, 1, -1) == secondValue);
41
42              firstValue = 0;
43              secondValue = 6892;
44
45              value = PartialWrite(value, firstValue, 0, 1);
46              value = PartialWrite(value, secondValue, 1, -1);
47
48              Assert.True(PartialRead(value, 0, 1) == firstValue);
49              Assert.True(PartialRead(value, 1, -1) == secondValue);
50          }
51
52          private static uint PartialWrite(uint target, uint source, int shift, int limit)
53          {
54              if (shift < 0)
55              {
56                  shift = 32 + shift;
57              }
58              if (limit < 0)
59              {
60                  limit = 32 + limit;
61              }
62              var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
63              var targetMask = ~(sourceMask << shift);
64              return (target & targetMask) | ((source & sourceMask) << shift);
65          }
66
67          private static uint PartialRead(uint target, int shift, int limit)
68          {
69              if (shift < 0)
70              {
71                  shift = 32 + shift;
72              }
73              if (limit < 0)
74              {
75                  limit = 32 + limit;
76              }
77              var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
78              var targetMask = sourceMask << shift;
79              return (target & targetMask) >> shift;
80          }
81      }
82  }
```

Index