LinksPlatform's Platform.Numbers Class Library

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

```csharp
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

```csharp
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

```csharp
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()
        {
```

```csharp
24              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
25              {
26                  Ensure.Always.IsNumeric<T>();
27                  emiter.LoadArguments(0, 1);
28                  emiter.Add();
29                  emiter.Return();
30              });
31          }
32
33          [MethodImpl(MethodImplOptions.AggressiveInlining)]
34          private static Func<T, T, T> CompileSubtractDelegate()
35          {
36              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
37              {
38                  Ensure.Always.IsNumeric<T>();
39                  emiter.LoadArguments(0, 1);
40                  emiter.Subtract();
41                  emiter.Return();
42              });
43          }
44
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          private static Func<T, T, T> CompileMultiplyDelegate()
47          {
48              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
49              {
50                  Ensure.Always.IsNumeric<T>();
51                  emiter.LoadArguments(0, 1);
52                  emiter.Emit(OpCodes.Mul);
53                  emiter.Return();
54              });
55          }
56
57          [MethodImpl(MethodImplOptions.AggressiveInlining)]
58          private static Func<T, T, T> CompileDivideDelegate()
59          {
60              return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
61              {
62                  Ensure.Always.IsNumeric<T>();
63                  emiter.LoadArguments(0, 1);
64                  if(NumericType<T>.IsSigned)
65                  {
66                      emiter.Emit(OpCodes.Div);
67                  }
68                  else
69                  {
70                      emiter.Emit(OpCodes.Div_Un);
71                  }
72                  emiter.Return();
73              });
74          }
75
76          [MethodImpl(MethodImplOptions.AggressiveInlining)]
77          private static Func<T, T> CompileIncrementDelegate()
78          {
79              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
80              {
81                  Ensure.Always.IsNumeric<T>();
82                  emiter.LoadArgument(0);
83                  emiter.Increment<T>();
84                  emiter.Return();
85              });
86          }
87
88          [MethodImpl(MethodImplOptions.AggressiveInlining)]
89          private static Func<T, T> CompileDecrementDelegate()
90          {
91              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
92              {
93                  Ensure.Always.IsNumeric<T>();
94                  emiter.LoadArgument(0);
95                  emiter.Decrement<T>();
96                  emiter.Return();
97              });
98          }
99      }
100  }
```

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

```csharp
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

```csharp
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)]
```

```csharp
                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

```csharp
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 =>
```

```csharp
        {
            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;
```

```csharp
                        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

```csharp
using System.Runtime.CompilerServices;

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

namespace Platform.Numbers
{
    /// <remarks>
```

```csharp
        /// 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 double[] _factorials =
        {
            1D, 1D, 2D, 6D, 24D, 120D, 720D, 5040D, 40320D, 362880D, 3628800D, 39916800D,
            479001600, 6227020800, 87178291200, 1307674368000D, 20922789888000D,
            355687428096000D, 6402373705728000D, 121645100408832000D, 2432902008176640000D,
            51090942171709440000D, 1124000727777607680000D
        };

        /// <remarks>
        /// <para>Source: https://oeis.org/A000108/list </para>
        /// <para>Источник: https://oeis.org/A000108/list </para>
        /// </remarks>
        private static readonly double[] _catalans =
        {
            1D,   1D,   2D,   5D,   14D,   42D,   132D,   429D,   1430D,   4862D,   16796D,   58786D,
            ↪   208012D,
            742900D,   2674440D,   9694845D,   35357670D,   129644790D,   477638700D,   1767263190D,
            6564120420D,   24466267020D,   91482563640D,   343059613650D,   1289904147324D,
            ↪   4861946401452D,
            18367353072152D,   69533550916004D,   263747951750360D,   1002242216651368D,
            ↪   3814986502092304D
        };

        /// <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 double Factorial(double n)
        {
            if (n <= 1)
            {
                return 1;
            }


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

            return n * Factorial(n - 1);
        }


        /// <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 double Catalan(double n)
        {


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

```
80        [MethodImpl(MethodImplOptions.AggressiveInlining)]
81        public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
82
83        [MethodImpl(MethodImplOptions.AggressiveInlining)]
84        public static T Abs<T>(T x) => Math<T>.Abs(x);
85
86        [MethodImpl(MethodImplOptions.AggressiveInlining)]
87        public static T Negate<T>(T x) => Math<T>.Negate(x);
88    }
89 }
```

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

```
1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Numbers
6  {
7      public static class MathExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
14     }
15 }
```

1.9 ./csharp/Platform.Numbers/Math[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 Math<T>
12     {
13         public static readonly Func<T, T> Abs = CompileAbsDelegate();
14         public static readonly Func<T, T> Negate = CompileNegateDelegate();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         private static Func<T, T> CompileAbsDelegate()
18         {
19             return DelegateHelpers.Compile<Func<T, T>>(emiter =>
20             {
21                 Ensure.Always.IsNumeric<T>();
22                 emiter.LoadArgument(0);
23                 if (NumericType<T>.IsSigned)
24                 {
25                     emiter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
26                 }
27                 emiter.Return();
28             });
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         private static Func<T, T> CompileNegateDelegate()
33         {
34             return DelegateHelpers.Compile<Func<T, T>>(emiter =>
35             {
36                 emiter.LoadArgument(0);
37                 emiter.Negate();
38                 emiter.Return();
39             });
40         }
41     }
42 }
```

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

```
1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class ArithmeticExtensionsTests
6      {
```

```
 7            [Fact]
 8            public static void IncrementTest()
 9            {
10                var number = 0UL;
11                var returnValue = number.Increment();
12                Assert.Equal(1UL, returnValue);
13                Assert.Equal(1UL, number);
14            }
15
16            [Fact]
17            public static void DecrementTest()
18            {
19                var number = 1UL;
20                var returnValue = number.Decrement();
21                Assert.Equal(0UL, returnValue);
22                Assert.Equal(0UL, number);
23            }
24        }
25    }
```

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

```
 1    using System;
 2    using Xunit;
 3
 4    namespace Platform.Numbers.Tests
 5    {
 6        public static class ArithmeticTests
 7        {
 8            [Fact]
 9            public static void CompiledOperationsTest()
10            {
11                Assert.Equal(3, Arithmetic.Add(1, 2));
12                Assert.Equal(1, Arithmetic.Subtract(2, 1));
13                Assert.Equal(8, Arithmetic.Multiply(2, 4));
14                Assert.Equal(4, Arithmetic.Divide(8, 2));
15                Assert.Equal(2, Arithmetic.Increment(1));
16                Assert.Equal(1UL, Arithmetic.Decrement(2UL));
17                Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
18            }
19        }
20    }
```

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

```
 1    using System;
 2    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            }
```

```csharp
        [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;
```

```csharp
117                    uint secondValue = 1543;
118
119                    // 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;
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
4  {
5      public static class MathExtensionsTests
6      {
7          [Fact]
8          public static void AbsTest()
9          {
10             var number = -1L;
11             var returnValue = number.Abs();
12             Assert.Equal(1L, returnValue);
13             Assert.Equal(1L, number);
14         }
15
16         [Fact]
17         public static void NegateTest()
18         {
19             var number = 2L;
20             var returnValue = number.Negate();
21             Assert.Equal(-2L, returnValue);
22             Assert.Equal(-2L, number);
23         }
24
25         [Fact]
26         public static void UnsignedNegateTest()
27         {
28             var number = 2UL;
29             var returnValue = number.Negate();
30             Assert.Equal(18446744073709551614, returnValue);
31             Assert.Equal(18446744073709551614, number);
32         }
33     }
34 }
```

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

```
1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class MathTests
6      {
7          [Fact]
8          public static void CompiledOperationsTest()
```

```csharp
        {
            Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
        }
    }
}
```

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

```csharp
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)
```

```
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