LinksPlatform's Platform.Numbers Class Library

1.1 ./Platform.Numbers/Arithmetic.cs

```csharp
1  using System;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Numbers
7  {
8      public static class Arithmetic
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         [Obsolete]
18         public static T Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
25     }
26 }
```

1.2 ./Platform.Numbers/ArithmeticExtensions.cs

```csharp
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 ArithmeticExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static T Decrement<T>(this ref T x) where T : struct => x =
           ↪   Arithmetic<T>.Decrement(x);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T Increment<T>(this ref T x) where T : struct => x =
           ↪   Arithmetic<T>.Increment(x);
14     }
15 }
```

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

```csharp
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 Arithmetic<T>
12     {
13         public static readonly Func<T, T, T> Add = CompileAddDelegate();
14         public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
15         public static readonly Func<T, T, T> Increment = CompileIncrementDelegate();
16         public static readonly Func<T, T, T> Decrement = CompileDecrementDelegate();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         private static Func<T, T, T> CompileAddDelegate()
20         {
21             return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
22             {
23                 Ensure.Always.IsNumeric<T>();
24                 emiter.LoadArguments(0, 1);
25                 emiter.Add();
26                 emiter.Return();
27             });
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
31        private static Func<T, T, T> CompileSubtractDelegate()
32        {
33            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
34            {
35                Ensure.Always.IsNumeric<T>();
36                emiter.LoadArguments(0, 1);
37                emiter.Subtract();
38                emiter.Return();
39            });
40        }
41
42        [MethodImpl(MethodImplOptions.AggressiveInlining)]
43        private static Func<T, T> CompileIncrementDelegate()
44        {
45            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
46            {
47                Ensure.Always.IsNumeric<T>();
48                emiter.LoadArgument(0);
49                emiter.Increment<T>();
50                emiter.Return();
51            });
52        }
53
54        [MethodImpl(MethodImplOptions.AggressiveInlining)]
55        private static Func<T, T> CompileDecrementDelegate()
56        {
57            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
58            {
59                Ensure.Always.IsNumeric<T>();
60                emiter.LoadArgument(0);
61                emiter.Decrement<T>();
62                emiter.Return();
63            });
64        }
65    }
66 }
```

1.4 ./Platform.Numbers/Bit.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 Bit
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static long Count(long x)
11        {
12            long n = 0;
13            while (x != 0)
14            {
15                n++;
16                x &= x - 1;
17            }
18            return n;
19        }
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        public static int GetLowestPosition(ulong value)
23        {
24            if (value == 0)
25            {
26                return -1;
27            }
28            var position = 0;
29            while ((value & 1UL) == 0)
30            {
31                value >>= 1;
32                ++position;
33            }
34            return position;
35        }
36
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        public static T Not<T>(T x) => Bit<T>.Not(x);
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
```

```
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public static T And<T>(T x, T y) => Bit<T>.And(x, y);
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
   ↪    Bit<T>.PartialWrite(target, source, shift, limit);
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public static T PartialRead<T>(T target, int shift, int limit) =>
   ↪    Bit<T>.PartialRead(target, shift, limit);
57     }
58 }
```

1.5   ./Platform.Numbers/BitExtensions.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 BitwiseExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         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);
14
15         [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   ./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         {
```

```csharp
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);
```

```csharp
                    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();
```

```
192              emiter.Return();
193          });
194      }

196      [MethodImpl(MethodImplOptions.AggressiveInlining)]
197      private static Tuple<int, T> GetConstants()
198      {
199          var type = typeof(T);
200          if (type == typeof(ulong))
201          {
202              return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
203          }
204          if (type == typeof(uint))
205          {
206              return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
207          }
208          if (type == typeof(ushort))
209          {
210              return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
211          }
212          if (type == typeof(byte))
213          {
214              return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
215          }
216          throw new NotSupportedException();
217      }
218  }
219 }
```

1.7 ./Platform.Numbers/Integer.cs

```
1  using System;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Numbers
7  {
8      [Obsolete]
9      public struct Integer : IEquatable<Integer>
10     {
11         public readonly ulong Value;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public Integer(ulong value) => Value = value;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static implicit operator Integer(ulong integer) => new Integer(integer);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static implicit operator Integer(long integer) => unchecked((ulong)integer);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public static implicit operator Integer(uint integer) => new Integer(integer);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static implicit operator Integer(int integer) => unchecked((ulong)integer);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public static implicit operator Integer(ushort integer) => new Integer(integer);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public static implicit operator Integer(short integer) => unchecked((ulong)integer);
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public static implicit operator Integer(byte integer) => new Integer(integer);
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public static implicit operator Integer(sbyte integer) => unchecked((ulong)integer);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static implicit operator Integer(bool integer) => integer ? 1UL : 0UL;
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public static implicit operator ulong(Integer integer) => integer.Value;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public static implicit operator long(Integer integer) => unchecked((long)integer.Value);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static implicit operator uint(Integer integer) => unchecked((uint)integer.Value);
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator int(Integer integer) => unchecked((int)integer.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator ushort(Integer integer) =>
        ↪   unchecked((ushort)integer.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator short(Integer integer) =>
        ↪   unchecked((short)integer.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator byte(Integer integer) => unchecked((byte)integer.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator sbyte(Integer integer) =>
        ↪   unchecked((sbyte)integer.Value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator bool(Integer integer) => integer.Value != 0UL;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(Integer other) => Value == other.Value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => Value.ToString();
    }
}
```

1.8 ./Platform.Numbers/Integer[T].cs

```csharp
using System;
using System.Runtime.CompilerServices;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
#pragma warning disable RECS0108 // Warns about static fields in generic types

namespace Platform.Numbers
{
    [Obsolete]
    public struct Integer<T> : IEquatable<Integer<T>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
        ↪   EqualityComparer<T>.Default;
        private static readonly Func<ulong, Integer<T>> _create;

        public static readonly T Zero;
        public static readonly T One;
        public static readonly T Two;

        public readonly T Value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        static Integer()
        {
            _create = CompileCreateDelegate();
            try
            {
                Zero = default;
                One = Arithmetic.Increment(Zero);
                Two = Arithmetic.Increment(One);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public Integer(T value) => Value = value;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator Integer(Integer<T> integer)
        {
            if (typeof(T) == typeof(Integer))
            {
```

```csharp
                    return (Integer)(object)integer.Value;
                }
                return Convert.ToUInt64(integer.Value);
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator ulong(Integer<T> integer) => ((Integer)integer).Value;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator T(Integer<T> integer) => integer.Value;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(ulong integer) => _create(integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(long integer) => unchecked((ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(uint integer) => new Integer(integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(int integer) => unchecked((ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(ushort integer) => new Integer(integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(short integer) => unchecked((ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(byte integer) => new Integer(integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(sbyte integer) => unchecked((ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator Integer<T>(bool integer) => integer ? 1UL : 0UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator long(Integer<T> integer) =>
                unchecked((long)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator uint(Integer<T> integer) =>
                unchecked((uint)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator int(Integer<T> integer) =>
                unchecked((int)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator ushort(Integer<T> integer) =>
                unchecked((ushort)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator short(Integer<T> integer) =>
                unchecked((short)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator byte(Integer<T> integer) =>
                unchecked((byte)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator sbyte(Integer<T> integer) =>
                unchecked((sbyte)(ulong)integer);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static implicit operator bool(Integer<T> integer) => integer != 0UL;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool Equals(Integer<T> other) => _equalityComparer.Equals(Value, other.Value);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
121         public override string ToString() => Value.ToString();
122
123         [MethodImpl(MethodImplOptions.AggressiveInlining)]
124         private static Func<ulong, Integer<T>> CompileCreateDelegate()
125         {
126             return DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emiter =>
127             {
128                 if (typeof(T) != typeof(Integer))
129                 {
130                     Ensure.Always.CanBeNumeric<T>();
131                 }
132                 emiter.LoadArgument(0);
133                 if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))
134                 {
135                     emiter.UncheckedConvert<ulong, T>();
136                     //emiter.Call(typeof(To).GetMethod(typeof(T).Name, Types<ulong>.Array));
137                 }
138                 if (NumericType<T>.IsNullable)
139                 {
140                     emiter.NewObject(typeof(T), NumericType<T>.UnderlyingType);
141                 }
142                 if (typeof(T) == typeof(Integer))
143                 {
144                     emiter.NewObject(typeof(Integer), typeof(ulong));
145                 }
146                 emiter.NewObject(typeof(Integer<T>), typeof(T));
147                 emiter.Return();
148             });
149         }
150     }
151 }
```

1.9   ./Platform.Numbers/Math.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      /// <remarks>
8      /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
9      /// </remarks>
10     public static class Math
11     {
12         /// <remarks>
13         /// <para>Source: https://oeis.org/A000142/list </para>
14         /// <para>Источник: https://oeis.org/A000142/list </para>
15         /// </remarks>
16         private static readonly double[] _factorials =
17         {
18             1D, 1D, 2D, 6D, 24D, 120D, 720D, 5040D, 40320D, 362880D, 3628800D, 39916800D,
19             479001600, 6227020800, 87178291200, 1307674368000D, 20922789888000D,
20             355687428096000D, 6402373705728000D, 121645100408832000D, 2432902008176640000D,
21             51090942171709440000D, 1124000727777607680000D
22         };
23
24         /// <remarks>
25         /// <para>Source: https://oeis.org/A000108/list </para>
26         /// <para>Источник: https://oeis.org/A000108/list </para>
27         /// </remarks>
28         private static readonly double[] _catalans =
29         {
30             1D,  1D,  2D,  5D,  14D,  42D,  132D,  429D,  1430D,  4862D,  16796D,  58786D,
             ↪  208012D,
31             742900D,  2674440D,  9694845D,  35357670D,  129644790D,  477638700D,  1767263190D,
32             6564120420D,  24466267020D,  91482563640D,  343059613650D,  1289904147324D,
             ↪  4861946401452D,
33             18367353072152D,  69533550916004D,  26374795175036D,  1002242216651368D,
             ↪  3814986502092304D
34         };
35
36         public static double Factorial(double n)
37         {
38             if (n <= 1)
39             {
40                 return 1;
41             }
42             if (n < _factorials.Length)
43             {
```

```
44              return _factorials[(int)n];
45          }
46          return n * Factorial(n - 1);
47      }
48
49      public static double Catalan(double n)
50      {
51          if (n <= 1)
52          {
53              return 1;
54          }
55          if (n < _catalans.Length)
56          {
57              return _catalans[(int)n];
58          }
59          return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
60      }
61
62      [MethodImpl(MethodImplOptions.AggressiveInlining)]
63      public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
64
65      [MethodImpl(MethodImplOptions.AggressiveInlining)]
66      public static T Abs<T>(T x) => Math<T>.Abs(x);
67
68      [MethodImpl(MethodImplOptions.AggressiveInlining)]
69      public static T Negate<T>(T x) => Math<T>.Negate(x);
70   }
71 }
```

1.10   ./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 }
```

1.11   ./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  }
```

1.12 ./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  }
```

1.13 ./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.True(Arithmetic<short>.Add(1, 2) == 3);
12              Assert.True(Arithmetic<short>.Subtract(2, 1) == 1);
13              Assert.True(Arithmetic<byte>.Increment(1) == 2);
14              Assert.True(Arithmetic<ulong>.Decrement(2) == 1);
15              Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
16          }
17      }
18  }
```

1.14 ./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()
```

```csharp
    {
        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);
```

```csharp
100
101                 // Using universal functions:
102                 Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
103                 Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
104
105                 firstValue = 0;
106                 secondValue = 6892;
107
108                 value = Bit.PartialWrite(value, firstValue, 0, 1);
109                 value = Bit.PartialWrite(value, secondValue, 1, -1);
110
111                 Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
112                 Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
113             }
114
115             {
116                 uint firstValue = 1;
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    }
```

1.15 ./Platform.Numbers.Tests/IntegerTests.cs

```
1    using System;
2    using Xunit;
3
4    namespace Platform.Numbers.Tests
5    {
6        public static class IntegerTests
7        {
8            [Fact]
9            [Obsolete]
10            public static void SignedIntegersTest()
11            {
12                var integer = -10;
13                Assert.True(((Integer)integer) == -10L);
14            }
15        }
16    }
```

1.16 ./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 }
```

1.17 ./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()
9          {
10             Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
11         }
12     }
13 }
```

1.18 ./Platform.Numbers.Tests/SystemTests.cs

```
1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class SystemTests
6      {
7          [Fact]
8          public static void PossiblePackTwoValuesIntoOneTest()
9          {
10             uint value = 0;
11
12             // Set one to first bit
13             value |= 1;
14
15             Assert.True(value == 1);
16
17             // Set zero to first bit
18             value &= 0xFFFFFFFE;
19
20             // Get first bit
21             uint read = value & 1;
22
23             Assert.True(read == 0);
24
25             uint firstValue = 1;
26             uint secondValue = 1543;
27
28             // Pack (join) two values at the same time
29             value = (secondValue << 1) | firstValue;
30
31             uint unpackagedFirstValue = value & 1;
32             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);
```

```csharp
            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