LinksPlatform's Platform.Numbers Class Library

./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 Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);

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

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

./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);
    }
}
```

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

```csharp
using System;
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;
        public static readonly Func<T, T, T> Subtract;
        public static readonly Func<T, T> Increment;
        public static readonly Func<T, T> Decrement;

        static Arithmetic()
        {
            Add = CompileAddDelegate();
            Subtract = CompileSubtractDelegate();
            Increment = CompileIncrementDelegate();
            Decrement = CompileDecrementDelegate();
        }

        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();
```

```
33                });
34            }
35
36            private static Func<T, T, T> CompileSubtractDelegate()
37            {
38                return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
39                {
40                    Ensure.Always.IsNumeric<T>();
41                    emiter.LoadArguments(0, 1);
42                    emiter.Subtract();
43                    emiter.Return();
44                });
45            }
46
47            private static Func<T, T> CompileIncrementDelegate()
48            {
49                return DelegateHelpers.Compile<Func<T, T>>(emiter =>
50                {
51                    Ensure.Always.IsNumeric<T>();
52                    emiter.LoadArgument(0);
53                    emiter.Increment<T>();
54                    emiter.Return();
55                });
56            }
57
58            private static Func<T, T> CompileDecrementDelegate()
59            {
60                return DelegateHelpers.Compile<Func<T, T>>(emiter =>
61                {
62                    Ensure.Always.IsNumeric<T>();
63                    emiter.LoadArgument(0);
64                    emiter.Decrement<T>();
65                    emiter.Return();
66                });
67            }
68        }
69    }
```

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

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

./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  }
```

./Platform.Numbers/Bit[T].cs
```
1   using System;
2   using System.Reflection.Emit;
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;
14          public static readonly Func<T, T, T> Or;
15          public static readonly Func<T, T, T> And;
16          public static readonly Func<T, int, T> ShiftLeft;
17          public static readonly Func<T, int, T> ShiftRight;
18          public static readonly Func<T, T, int, int, T> PartialWrite;
19          public static readonly Func<T, int, int, T> PartialRead;
20
21          static Bit()
22          {
23              Not = CompileNotDelegate();
24              Or = CompileOrDelegate();
25              And = CompileAndDelegate();
26              ShiftLeft = CompileShiftLeftDelegate();
27              ShiftRight = CompileShiftRightDelegate();
28              PartialWrite = CompilePartialWriteDelegate();
29              PartialRead = CompilePartialReadDelegate();
30          }
31
32          private static Func<T, T> CompileNotDelegate()
33          {
34              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
35              {
36                  Ensure.Always.IsNumeric<T>();
37                  emiter.LoadArguments(0);
```

```csharp
38              emiter.Not();
39              emiter.Return();
40          });
41      }
42
43      private static Func<T, T, T> CompileOrDelegate()
44      {
45          return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
46          {
47              Ensure.Always.IsNumeric<T>();
48              emiter.LoadArguments(0, 1);
49              emiter.Or();
50              emiter.Return();
51          });
52      }
53
54      private static Func<T, T, T> CompileAndDelegate()
55      {
56          return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
57          {
58              Ensure.Always.IsNumeric<T>();
59              emiter.LoadArguments(0, 1);
60              emiter.And();
61              emiter.Return();
62          });
63      }
64
65      private static Func<T, int, T> CompileShiftLeftDelegate()
66      {
67          return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
68          {
69              Ensure.Always.IsNumeric<T>();
70              emiter.LoadArguments(0, 1);
71              emiter.ShiftLeft();
72              emiter.Return();
73          });
74      }
75
76      private static Func<T, int, T> CompileShiftRightDelegate()
77      {
78          return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
79          {
80              Ensure.Always.IsNumeric<T>();
81              emiter.LoadArguments(0, 1);
82              emiter.ShiftRight();
83              emiter.Return();
84          });
85      }
86
87      private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
88      {
89          return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emiter =>
90          {
91              Ensure.Always.IsNumeric<T>();
92              var constants = GetConstants();
93              var bitsNumber = constants.Item1;
94              var numberFilledWithOnes = constants.Item2;
95              ushort shiftArgument = 2;
96              ushort limitArgument = 3;
97              var checkLimit = emiter.DefineLabel();
98              var calculateSourceMask = emiter.DefineLabel();
99              // Check shift
100             emiter.LoadArgument(shiftArgument);
101             emiter.LoadConstant(0);
102             emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
103             // Fix shift
104             emiter.LoadConstant(bitsNumber);
105             emiter.LoadArgument(shiftArgument);
106             emiter.Add();
107             emiter.StoreArgument(shiftArgument);
108             emiter.MarkLabel(checkLimit);
109             // Check limit
110             emiter.LoadArgument(limitArgument);
111             emiter.LoadConstant(0);
112             emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
113             // Fix limit
114             emiter.LoadConstant(bitsNumber);
115             emiter.LoadArgument(limitArgument);
```

```csharp
116                     emiter.Add();
117                     emiter.StoreArgument(limitArgument);
118                     emiter.MarkLabel(calculateSourceMask);
119                     var sourceMask = emiter.DeclareLocal<T>();
120                     var targetMask = emiter.DeclareLocal<T>();
121                     emiter.LoadConstant(typeof(T), numberFilledWithOnes);
122                     emiter.LoadArgument(limitArgument);
123                     emiter.ShiftLeft();
124                     emiter.Not();
125                     emiter.LoadConstant(typeof(T), numberFilledWithOnes);
126                     emiter.And();
127                     emiter.StoreLocal(sourceMask);
128                     emiter.LoadLocal(sourceMask);
129                     emiter.LoadArgument(shiftArgument);
130                     emiter.ShiftLeft();
131                     emiter.Not();
132                     emiter.StoreLocal(targetMask);
133                     emiter.LoadArgument(0); // target
134                     emiter.LoadLocal(targetMask);
135                     emiter.And();
136                     emiter.LoadArgument(1); // source
137                     emiter.LoadLocal(sourceMask);
138                     emiter.And();
139                     emiter.LoadArgument(shiftArgument);
140                     emiter.ShiftLeft();
141                     emiter.Or();
142                     emiter.Return();
143                 });
144         }
145
146         private static Func<T, int, int, T> CompilePartialReadDelegate()
147         {
148             return DelegateHelpers.Compile<Func<T, int, int, T>>(emiter =>
149             {
150                 Ensure.Always.IsNumeric<T>();
151                 var constants = GetConstants();
152                 var bitsNumber = constants.Item1;
153                 var numberFilledWithOnes = constants.Item2;
154                 ushort shiftArgument = 1;
155                 ushort limitArgument = 2;
156                 var checkLimit = emiter.DefineLabel();
157                 var calculateSourceMask = emiter.DefineLabel();
158                 // Check shift
159                 emiter.LoadArgument(shiftArgument);
160                 emiter.LoadConstant(0);
161                 emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
162                 // Fix shift
163                 emiter.LoadConstant(bitsNumber);
164                 emiter.LoadArgument(shiftArgument);
165                 emiter.Add();
166                 emiter.StoreArgument(shiftArgument);
167                 emiter.MarkLabel(checkLimit);
168                 // Check limit
169                 emiter.LoadArgument(limitArgument);
170                 emiter.LoadConstant(0);
171                 emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
172                 // Fix limit
173                 emiter.LoadConstant(bitsNumber);
174                 emiter.LoadArgument(limitArgument);
175                 emiter.Add();
176                 emiter.StoreArgument(limitArgument);
177                 emiter.MarkLabel(calculateSourceMask);
178                 var sourceMask = emiter.DeclareLocal<T>();
179                 var targetMask = emiter.DeclareLocal<T>();
180                 emiter.LoadConstant(typeof(T), numberFilledWithOnes);
181                 emiter.LoadArgument(limitArgument); // limit
182                 emiter.ShiftLeft();
183                 emiter.Not();
184                 emiter.LoadConstant(typeof(T), numberFilledWithOnes);
185                 emiter.And();
186                 emiter.StoreLocal(sourceMask);
187                 emiter.LoadLocal(sourceMask);
188                 emiter.LoadArgument(shiftArgument);
189                 emiter.ShiftLeft();
190                 emiter.StoreLocal(targetMask);
191                 emiter.LoadArgument(0); // target
192                 emiter.LoadLocal(targetMask);
193                 emiter.And();
```

```csharp
194                    emiter.LoadArgument(shiftArgument);
195                    emiter.ShiftRight();
196                    emiter.Return();
197                });
198            }
199
200        private static Tuple<int, T> GetConstants()
201        {
202            var type = typeof(T);
203            if (type == typeof(ulong))
204            {
205                return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
206            }
207            if (type == typeof(uint))
208            {
209                return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
210            }
211            if (type == typeof(ushort))
212            {
213                return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
214            }
215            if (type == typeof(byte))
216            {
217                return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
218            }
219            throw new NotSupportedException();
220        }
221    }
222 }
```

./Platform.Numbers/Integer.cs

```csharp
1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Numbers
8  {
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) => To.UInt64(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) => To.UInt64(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) => To.UInt64(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) => To.UInt64(integer);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public static implicit operator Integer(bool integer) => To.UInt64(integer);
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) => To.Int64(integer.Value);
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
50          public static implicit operator uint(Integer integer) => To.UInt32(integer.Value);

51
52          [MethodImpl(MethodImplOptions.AggressiveInlining)]
53          public static implicit operator int(Integer integer) => To.Int32(integer.Value);

54
55          [MethodImpl(MethodImplOptions.AggressiveInlining)]
56          public static implicit operator ushort(Integer integer) => To.UInt16(integer.Value);

57
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public static implicit operator short(Integer integer) => To.Int16(integer.Value);

60
61          [MethodImpl(MethodImplOptions.AggressiveInlining)]
62          public static implicit operator byte(Integer integer) => To.Byte(integer.Value);

63
64          [MethodImpl(MethodImplOptions.AggressiveInlining)]
65          public static implicit operator sbyte(Integer integer) => To.SByte(integer.Value);

66
67          [MethodImpl(MethodImplOptions.AggressiveInlining)]
68          public static implicit operator bool(Integer integer) => To.Boolean(integer.Value);

69
70          [MethodImpl(MethodImplOptions.AggressiveInlining)]
71          public bool Equals(Integer other) => Value == other.Value;

72
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public override string ToString() => Value.ToString();
75      }
76  }
```

./Platform.Numbers/Integer[T].cs
```csharp
1   using System;
2   using System.Runtime.CompilerServices;
3   using System.Collections.Generic;
4   using Platform.Exceptions;
5   using Platform.Reflection;
6   using Platform.Converters;

7
8   // ReSharper disable StaticFieldInGenericType
9   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10  #pragma warning disable RECS0108 // Warns about static fields in generic types

11
12  namespace Platform.Numbers
13  {
14      public struct Integer<T> : IEquatable<Integer<T>>
15      {
16          private static readonly EqualityComparer<T> _equalityComparer =
            ↪  EqualityComparer<T>.Default;
17          private static readonly Func<ulong, Integer<T>> _create;

18
19          public static readonly T Zero;
20          public static readonly T One;
21          public static readonly T Two;

22
23          public readonly T Value;

24
25          static Integer()
26          {
27              _create = CompileCreateDelegate();
28              try
29              {
30                  Zero = default;
31                  One = Arithmetic.Increment(Zero);
32                  Two = Arithmetic.Increment(One);
33              }
34              catch (Exception exception)
35              {
36                  exception.Ignore();
37              }
38          }

39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public Integer(T value) => Value = value;

42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public static implicit operator Integer(Integer<T> integer)
45          {
46              if (typeof(T) == typeof(Integer))
47              {
48                  return (Integer)(object)integer.Value;
49              }
50              return Convert.ToUInt64(integer.Value);
```

```csharp
        }

        [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) => To.UInt64(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) => To.UInt64(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) => To.UInt64(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) => To.UInt64(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator long(Integer<T> integer) => To.Int64(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator int(Integer<T> integer) => To.Int32(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator ushort(Integer<T> integer) => To.UInt16(integer);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static implicit operator short(Integer<T> integer) => To.Int16(integer);

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

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

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

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

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

        private static Func<ulong, Integer<T>> CompileCreateDelegate()
        {
            return DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emiter =>
            {
                if (typeof(T) != typeof(Integer))
                {
                    Ensure.Always.CanBeNumeric<T>();
                }
                emiter.LoadArgument(0);
```

```
131             if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))
132             {
133                 emiter.Call(typeof(To).GetMethod(typeof(T).Name, Types<ulong>.Array));
134             }
135             if (NumericType<T>.IsNullable)
136             {
137                 emiter.NewObject(typeof(T), NumericType<T>.UnderlyingType);
138             }
139             if (typeof(T) == typeof(Integer))
140             {
141                 emiter.NewObject(typeof(Integer), typeof(ulong));
142             }
143             emiter.NewObject(typeof(Integer<T>), typeof(T));
144             emiter.Return();
145         });
146     }
147   }
148 }
```

./Platform.Numbers/Math.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     /// <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         /// Source: https://oeis.org/A000142/list
14         /// </remarks>
15         private static readonly ulong[] _factorials =
16         {
17             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
18             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
19             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
20         };
21
22         /// <remarks>
23         /// Source: https://oeis.org/A000108/list
24         /// </remarks>
25         private static readonly ulong[] _catalans =
26         {
27             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
28             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
29             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
30             18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304
31         };
32
33         public static double Factorial(double n)
34         {
35             if (n <= 1)
36             {
37                 return 1;
38             }
39             if (n < _factorials.Length)
40             {
41                 return _factorials[(int)n];
42             }
43             return n * Factorial(n - 1);
44         }
45
46         public static double Catalan(double n)
47         {
48             if (n <= 1)
49             {
50                 return 1;
51             }
52             if (n < _catalans.Length)
53             {
54                 return _catalans[(int)n];
55             }
56             return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
57         }
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
60    public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;

61
62    [MethodImpl(MethodImplOptions.AggressiveInlining)]
63    public static T Abs<T>(T x) => Math<T>.Abs(x);
64
65    [MethodImpl(MethodImplOptions.AggressiveInlining)]
66    public static T Negate<T>(T x) => Math<T>.Negate(x);
67    }
68  }
```

./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  }
```

./Platform.Numbers/Math[T].cs

```
1   using System;
2   using System.Reflection;
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;
14          public static readonly Func<T, T> Negate;
15
16          static Math()
17          {
18              Abs = CompileAbsDelegate();
19              Negate = CompileNegateDelegate();
20          }
21
22          private static Func<T, T> CompileAbsDelegate()
23          {
24              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
25              {
26                  Ensure.Always.IsNumeric<T>();
27                  emiter.LoadArgument(0);
28                  if (NumericType<T>.IsSigned)
29                  {
30                      emiter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
31                  }
32                  emiter.Return();
33              });
34          }
35
36          private static Func<T, T> CompileNegateDelegate()
37          {
38              return DelegateHelpers.Compile<Func<T, T>>(emiter =>
39              {
40                  Ensure.Always.IsSigned<T>();
41                  emiter.LoadArgument(0);
42                  emiter.Negate();
43                  emiter.Return();
44              });
45          }
46      }
47  }
```

./Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```
1   using Xunit;
2
```

```csharp
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);
        }
    }
}
```

./Platform.Numbers.Tests/ArithmeticTests.cs
```csharp
using System;
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class ArithmeticTests
    {
        [Fact]
        public static void CompiledOperationsTest()
        {
            Assert.True(Arithmetic<short>.Add(1, 2) == 3);
            Assert.True(Arithmetic<short>.Subtract(2, 1) == 1);
            Assert.True(Arithmetic<byte>.Increment(1) == 2);
            Assert.True(Arithmetic<ulong>.Decrement(2) == 1);
            Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
        }
    }
}
```

./Platform.Numbers.Tests/BitTests.cs
```csharp
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));
        }
    }
}
```

```csharp
35              Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
36          }
37
38          [Fact]
39          public static void UInt32BitwiseOperationsTest()
40          {
41              Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
42              Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
43              Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
44              Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
45              Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
46          }
47
48          [Fact]
49          public static void UInt64BitwiseOperationsTest()
50          {
51              Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
52              Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
53              Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
54              Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
55              Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
56          }
57
58          [Fact]
59          public static void PartialReadWriteTest()
60          {
61              {
62                  uint firstValue = 1;
63                  uint secondValue = 1543;
64
65                  // Pack (join) two values at the same time
66                  uint value = secondValue << 1 | firstValue;
67
68                  uint unpackagedFirstValue = value & 1;
69                  uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
70
71                  Assert.True(firstValue == unpackagedFirstValue);
72                  Assert.True(secondValue == unpackagedSecondValue);
73
74                  // Using universal functions:
75                  Assert.True(PartialRead(value, 0, 1) == firstValue);
76                  Assert.True(PartialRead(value, 1, -1) == secondValue);
77
78                  firstValue = 0;
79                  secondValue = 6892;
80
81                  value = PartialWrite(value, firstValue, 0, 1);
82                  value = PartialWrite(value, secondValue, 1, -1);
83
84                  Assert.True(PartialRead(value, 0, 1) == firstValue);
85                  Assert.True(PartialRead(value, 1, -1) == secondValue);
86              }
87
88              {
89                  uint firstValue = 1;
90                  uint secondValue = 1543;
91
92                  // Pack (join) two values at the same time
93                  uint value = secondValue << 1 | firstValue;
94
95                  uint unpackagedFirstValue = value & 1;
96                  uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
97
98                  Assert.True(firstValue == unpackagedFirstValue);
99                  Assert.True(secondValue == unpackagedSecondValue);
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
```

```csharp
                    {
                        uint firstValue = 1;
                        uint secondValue = 1543;

                        // Pack (join) two values at the same time
                        uint value = secondValue << 1 | firstValue;

                        uint unpackagedFirstValue = value & 1;
                        uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                        Assert.True(firstValue == unpackagedFirstValue);
                        Assert.True(secondValue == unpackagedSecondValue);

                        // Using universal functions:
                        var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
                        var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
                        var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
                        var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);

                        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);

                        firstValue = 0;
                        secondValue = 6892;

                        value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
                        value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);

                        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
                    }
            }

        // TODO: Can be optimized using precalculation of TargetMask and SourceMask
        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return target & targetMask | (source & sourceMask) << shift;
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
        }

        private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
        }

        private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
```

```csharp
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 }
```

./Platform.Numbers.Tests/MathExtensionsTests.cs
```csharp
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  }
```

./Platform.Numbers.Tests/MathTests.cs
```csharp
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  }
```

```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)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
            return (target & targetMask) >> shift;
```

```
80            }
81        }
82    }
```

Index