

LinksPlatform's Platform.Numbers Class Library

./Platform.Numbers/Arithmetic.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      public static class Arithmetic
6      {
7          public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
8          public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
9          public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
10         public static T Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);
11         public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
12     }
13 }
```

./Platform.Numbers/ArithmeticExtensions.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      public static class ArithmeticExtensions
6      {
7          public static T Decrement<T>(this ref T x) where T : struct => x =
8              ↪ Arithmetic<T>.Decrement(x);
9          public static T Increment<T>(this ref T x) where T : struct => x =
10             ↪ Arithmetic<T>.Increment(x);
11     }
12 }
```

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

```
1  using System;
2  using Platform.Exceptions;
3  using Platform.Reflection;
4
5  // ReSharper disable StaticFieldInGenericType
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Numbers
9  {
10     public static class Arithmetic<T>
11     {
12         public static readonly Func<T, T, T> Add;
13         public static readonly Func<T, T, T> Subtract;
14         public static readonly Func<T, T> Increment;
15         public static readonly Func<T, T> Decrement;
16
17         static Arithmetic()
18         {
19             Add = CompileAddDelegate();
20             Subtract = CompileSubtractDelegate();
21             Increment = CompileIncrementDelegate();
22             Decrement = CompileDecrementDelegate();
23         }
24
25         private static Func<T, T, T> CompileAddDelegate()
26         {
27             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
28             {
29                 Ensure.Always.IsNumeric<T>();
30                 emitter.LoadArguments(0, 1);
31                 emitter.Add();
32                 emitter.Return();
33             });
34         }
35
36         private static Func<T, T, T> CompileSubtractDelegate()
37         {
38             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
39             {
40                 Ensure.Always.IsNumeric<T>();
41                 emitter.LoadArguments(0, 1);
42                 emitter.Subtract();
43                 emitter.Return();
44             });
45         }
46
47         private static Func<T, T> CompileIncrementDelegate()
48         {
```

```

49         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
50         {
51             Ensure.Always.IsNumeric<T>();
52             emitter.LoadArgument(0);
53             emitter.Increment<T>();
54             emitter.Return();
55         });
56     }
57
58     private static Func<T, T> CompileDecrementDelegate()
59     {
60         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
61         {
62             Ensure.Always.IsNumeric<T>();
63             emitter.LoadArgument(0);
64             emitter.Decrement<T>();
65             emitter.Return();
66         });
67     }
68 }
69 }

```

./Platform.Numbers/Bit.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      public static class Bit
6      {
7          public static long Count(long x)
8          {
9              long n = 0;
10             while (x != 0)
11             {
12                 n++;
13                 x &= x - 1;
14             }
15             return n;
16         }
17
18         public static int GetLowestPosition(ulong value)
19         {
20             if (value == 0)
21             {
22                 return -1;
23             }
24             var position = 0;
25             while ((value & 1UL) == 0)
26             {
27                 value >>= 1;
28                 ++position;
29             }
30             return position;
31         }
32
33         public static T And<T>(T x, T y) => Bit<T>.And(x, y);
34
35         public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
36             ↪ Bit<T>.PartialWrite(target, source, shift, limit);
37
38         public static T PartialRead<T>(T target, int shift, int limit) =>
39             ↪ Bit<T>.PartialRead(target, shift, limit);
40     }
41 }

```

./Platform.Numbers/BitExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      public static class BitwiseExtensions
6      {
7          public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
8              ↪ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
9          public static T PartialRead<T>(this T target, int shift, int limit) =>
10             ↪ Bit<T>.PartialRead(target, shift, limit);
11     }
12 }

```

./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>>(emitter =>
35             {
36                 Ensure.Always.IsNumeric<T>();
37                 emitter.LoadArguments(0);
38                 emitter.Not();
39                 emitter.Return();
40             });
41         }
42
43         private static Func<T, T, T> CompileOrDelegate()
44         {
45             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
46             {
47                 Ensure.Always.IsNumeric<T>();
48                 emitter.LoadArguments(0, 1);
49                 emitter.Or();
50                 emitter.Return();
51             });
52         }
53
54         private static Func<T, T, T> CompileAndDelegate()
55         {
56             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
57             {
58                 Ensure.Always.IsNumeric<T>();
59                 emitter.LoadArguments(0, 1);
60                 emitter.And();
61                 emitter.Return();
62             });
63         }
64
65         private static Func<T, int, T> CompileShiftLeftDelegate()
66         {
67             return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
68             {
69                 Ensure.Always.IsNumeric<T>();
70                 emitter.LoadArguments(0, 1);
71                 emitter.ShiftLeft();
72                 emitter.Return();
73             });
74         }
75
76         private static Func<T, int, T> CompileShiftRightDelegate()
77         {
78             return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
```

```

79     {
80         Ensure.Always.IsNumeric<T>();
81         emitter.LoadArguments(0, 1);
82         emitter.ShiftRight();
83         emitter.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>>(emitter =>
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 = emitter.DefineLabel();
98         var calculateSourceMask = emitter.DefineLabel();
99         // Check shift
100        emitter.LoadArgument(shiftArgument);
101        emitter.LoadConstant(0);
102        emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
103        // Fix shift
104        emitter.LoadConstant(bitsNumber);
105        emitter.LoadArgument(shiftArgument);
106        emitter.Add();
107        emitter.StoreArgument(shiftArgument);
108        emitter.MarkLabel(checkLimit);
109        // Check limit
110        emitter.LoadArgument(limitArgument);
111        emitter.LoadConstant(0);
112        emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
113        // Fix limit
114        emitter.LoadConstant(bitsNumber);
115        emitter.LoadArgument(limitArgument);
116        emitter.Add();
117        emitter.StoreArgument(limitArgument);
118        emitter.MarkLabel(calculateSourceMask);
119        var sourceMask = emitter.DeclareLocal<T>();
120        var targetMask = emitter.DeclareLocal<T>();
121        //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
122        LoadMaxValueConstant(emitter);
123        emitter.LoadArgument(limitArgument);
124        emitter.ShiftLeft();
125        emitter.Not();
126        //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
127        LoadMaxValueConstant(emitter);
128        emitter.And();
129        emitter.StoreLocal(sourceMask);
130        emitter.LoadLocal(sourceMask);
131        emitter.LoadArgument(shiftArgument);
132        emitter.ShiftLeft();
133        emitter.Not();
134        emitter.StoreLocal(targetMask);
135        emitter.LoadArgument(0); // target
136        emitter.LoadLocal(targetMask);
137        emitter.And();
138        emitter.LoadArgument(1); // source
139        emitter.LoadLocal(sourceMask);
140        emitter.And();
141        emitter.LoadArgument(shiftArgument);
142        emitter.ShiftLeft();
143        emitter.Or();
144        emitter.Return();
145    });
146 }
147
148 private static Func<T, int, int, T> CompilePartialReadDelegate()
149 {
150     return DelegateHelpers.Compile<Func<T, int, int, T>>(emitter =>
151     {
152         Ensure.Always.IsNumeric<T>();
153         var constants = GetConstants();
154         var bitsNumber = constants.Item1;
155         var numberFilledWithOnes = constants.Item2;
156         ushort shiftArgument = 1;

```

```

157     ushort limitArgument = 2;
158     var checkLimit = emitter.DefineLabel();
159     var calculateSourceMask = emitter.DefineLabel();
160     // Check shift
161     emitter.LoadArgument(shiftArgument);
162     emitter.LoadConstant(0);
163     emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
164     // Fix shift
165     emitter.LoadConstant(bitsNumber);
166     emitter.LoadArgument(shiftArgument);
167     emitter.Add();
168     emitter.StoreArgument(shiftArgument);
169     emitter.MarkLabel(checkLimit);
170     // Check limit
171     emitter.LoadArgument(limitArgument);
172     emitter.LoadConstant(0);
173     emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
174     // Fix limit
175     emitter.LoadConstant(bitsNumber);
176     emitter.LoadArgument(limitArgument);
177     emitter.Add();
178     emitter.StoreArgument(limitArgument);
179     emitter.MarkLabel(calculateSourceMask);
180     var sourceMask = emitter.DeclareLocal<T>();
181     var targetMask = emitter.DeclareLocal<T>();
182     //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
183     LoadMaxValueConstant(emitter);
184     emitter.LoadArgument(limitArgument); // limit
185     emitter.ShiftLeft();
186     emitter.Not();
187     //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
188     LoadMaxValueConstant(emitter);
189     emitter.And();
190     emitter.StoreLocal(sourceMask);
191     emitter.LoadLocal(sourceMask);
192     emitter.LoadArgument(shiftArgument);
193     emitter.ShiftLeft();
194     emitter.StoreLocal(targetMask);
195     emitter.LoadArgument(0); // target
196     emitter.LoadLocal(targetMask);
197     emitter.And();
198     emitter.LoadArgument(shiftArgument);
199     emitter.ShiftRight();
200     emitter.Return();
201 });
202 }
203
204 private static void LoadMaxValueConstant(ILGenerator emitter)
205 {
206     var type = typeof(T);
207     if (type == typeof(ulong))
208     {
209         emitter.Emit(OpCodes.Ldc_I8, unchecked((long)ulong.MaxValue));
210     }
211     else if (type == typeof(uint))
212     {
213         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)uint.MaxValue));
214     }
215     else if (type == typeof(ushort))
216     {
217         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)ushort.MaxValue));
218     }
219     else if (type == typeof(byte))
220     {
221         emitter.Emit(OpCodes.Ldc_I4_S, unchecked((sbyte)byte.MaxValue));
222     }
223     else
224     {
225         throw new NotSupportedException();
226     }
227 }
228
229 private static Tuple<int, T> GetConstants()
230 {
231     var type = typeof(T);
232     if (type == typeof(ulong))
233     {
234         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);

```

```

235     }
236     if (type == typeof(uint))
237     {
238         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
239     }
240     if (type == typeof(ushort))
241     {
242         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
243     }
244     if (type == typeof(byte))
245     {
246         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
247     }
248     throw new NotSupportedException();
249 }
250 }
251 }

```

./Platform.Numbers/Integer.cs

```

1  using System;
2  using Platform.Converters;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Numbers
7  {
8      public struct Integer : IEquatable<Integer>
9      {
10         public readonly ulong Value;
11         public Integer(ulong value) => Value = value;
12         public static implicit operator Integer(ulong integer) => new Integer(integer);
13         public static implicit operator Integer(long integer) => To.UInt64(integer);
14         public static implicit operator Integer(uint integer) => new Integer(integer);
15         public static implicit operator Integer(int integer) => To.UInt64(integer);
16         public static implicit operator Integer(ushort integer) => new Integer(integer);
17         public static implicit operator Integer(short integer) => To.UInt64(integer);
18         public static implicit operator Integer(byte integer) => new Integer(integer);
19         public static implicit operator Integer(sbyte integer) => To.UInt64(integer);
20         public static implicit operator Integer(bool integer) => To.UInt64(integer);
21         public static implicit operator ulong(Integer integer) => integer.Value;
22         public static implicit operator long(Integer integer) => To.Int64(integer.Value);
23         public static implicit operator uint(Integer integer) => To.UInt32(integer.Value);
24         public static implicit operator int(Integer integer) => To.Int32(integer.Value);
25         public static implicit operator ushort(Integer integer) => To.UInt16(integer.Value);
26         public static implicit operator short(Integer integer) => To.Int16(integer.Value);
27         public static implicit operator byte(Integer integer) => To.Byte(integer.Value);
28         public static implicit operator sbyte(Integer integer) => To.SByte(integer.Value);
29         public static implicit operator bool(Integer integer) => To.Boolean(integer.Value);
30         public bool Equals(Integer other) => Value == other.Value;
31         public override string ToString() => Value.ToString();
32     }
33 }

```

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

```

1  using System;
2  using System.Reflection;
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
11  namespace Platform.Numbers
12  {
13      public struct Integer<T> : IEquatable<Integer<T>>
14      {
15         private static readonly EqualityComparer<T> _equalityComparer =
16             ↪ 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         {

```

```

26     _create = CompileCreateDelegate();
27     try
28     {
29         Zero = default;
30         One = Arithmetic.Increment(Zero);
31         Two = Arithmetic.Increment(One);
32     }
33     catch (Exception exception)
34     {
35         exception.Ignore();
36     }
37 }
38
39 public Integer(T value) => Value = value;
40
41 public static implicit operator Integer(Integer<T> integer)
42 {
43     if (typeof(T) == typeof(Integer))
44     {
45         return (Integer)(object)integer.Value;
46     }
47     return Convert.ToUInt64(integer.Value);
48 }
49
50 public static implicit operator ulong(Integer<T> integer) => ((Integer)integer).Value;
51
52 public static implicit operator T(Integer<T> integer) => integer.Value;
53
54 public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);
55
56 public static implicit operator Integer<T>(ulong integer) => _create(integer);
57
58 public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);
59
60 public static implicit operator Integer<T>(long integer) => To.UInt64(integer);
61
62 public static implicit operator Integer<T>(uint integer) => new Integer(integer);
63
64 public static implicit operator Integer<T>(int integer) => To.UInt64(integer);
65
66 public static implicit operator Integer<T>(ushort integer) => new Integer(integer);
67
68 public static implicit operator Integer<T>(short integer) => To.UInt64(integer);
69
70 public static implicit operator Integer<T>(byte integer) => new Integer(integer);
71
72 public static implicit operator Integer<T>(sbyte integer) => To.UInt64(integer);
73
74 public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);
75
76 public static implicit operator long(Integer<T> integer) => To.Int64(integer);
77
78 public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);
79
80 public static implicit operator int(Integer<T> integer) => To.Int32(integer);
81
82 public static implicit operator ushort(Integer<T> integer) => To.UInt16(integer);
83
84 public static implicit operator short(Integer<T> integer) => To.Int16(integer);
85
86 public static implicit operator byte(Integer<T> integer) => To.Byte(integer);
87
88 public static implicit operator sbyte(Integer<T> integer) => To.SByte(integer);
89
90 public static implicit operator bool(Integer<T> integer) => To.Boolean(integer);
91
92 public bool Equals(Integer<T> other) => _equalityComparer.Equals(Value, other.Value);
93
94 public override string ToString() => Value.ToString();
95
96 private static Func<ulong, Integer<T>> CompileCreateDelegate()
97 {
98     return DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emitter =>
99     {
100         if (typeof(T) != typeof(Integer))
101         {
102             Ensure.Always.CanBeNumeric<T>();
103         }
104         emitter.LoadArgument(0);
105         if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))

```

```

106     {
107         emitter.Call(typeof(To).GetTypeInfo().GetMethod(typeof(T).Name,
108             ↪ Types<ulong>.Array));
109     }
110     if (NumericType<T>.IsNullable)
111     {
112         emitter.NewObject(typeof(T), NumericType<T>.UnderlyingType);
113     }
114     if (typeof(T) == typeof(Integer))
115     {
116         emitter.NewObject(typeof(Integer), typeof(ulong));
117     }
118     emitter.NewObject(typeof(Integer<T>), typeof(T));
119     emitter.Return();
120 });
121 }
122 }

```

./Platform.Numbers/Math.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      /// <remarks>
6      /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
7      /// </remarks>
8      public static class Math
9      {
10         /// <remarks>
11         /// Source: https://oeis.org/A000142/list
12         /// </remarks>
13         private static readonly ulong[] _factorials =
14         {
15             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
16             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
17             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
18         };
19
20         /// <remarks>
21         /// Source: https://oeis.org/A000108/list
22         /// </remarks>
23         private static readonly ulong[] _catalans =
24         {
25             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
26             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
27             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
28             18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304
29         };
30
31         public static double Factorial(double n)
32         {
33             if (n <= 1)
34             {
35                 return 1;
36             }
37             if (n < _factorials.Length)
38             {
39                 return _factorials[(int)n];
40             }
41             return n * Factorial(n - 1);
42         }
43
44         public static double Catalan(double n)
45         {
46             if (n <= 1)
47             {
48                 return 1;
49             }
50             if (n < _catalans.Length)
51             {
52                 return _catalans[(int)n];
53             }
54             return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
55         }
56
57         public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
58
59         public static T Abs<T>(T x) => Math<T>.Abs(x);

```



```

60
61     public static T Negate<T>(T x) => Math<T>.Negate(x);
62 }
63 }

```

./Platform.Numbers/MathExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Numbers
4  {
5      public static class MathExtensions
6      {
7          public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
8          public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
9      }
10 }

```

./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>>(emitter =>
25             {
26                 Ensure.Always.IsNumeric<T>();
27                 emitter.LoadArgument(0);
28                 if (NumericType<T>.IsSigned)
29                 {
30                     emitter.Call(typeof(System.Math).GetTypeInfo().GetMethod("Abs",
31                                     ↪ Types<T>.Array));
32                 }
33                 emitter.Return();
34             });
35
36             private static Func<T, T> CompileNegateDelegate()
37             {
38                 return DelegateHelpers.Compile<Func<T, T>>(emitter =>
39                 {
40                     Ensure.Always.IsSigned<T>();
41                     emitter.LoadArgument(0);
42                     emitter.Negate();
43                     emitter.Return();
44                 });
45             }
46         }
47     }

```

./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 }

```

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

```

./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         }
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 & 0xFFFFFFF0) >> 1;
70
71         Assert.True(firstValue == unpackagedFirstValue);
72         Assert.True(secondValue == unpackagedSecondValue);
73
74         // Using universal functions:
75
76         Assert.True(PartialRead(value, 0, 1) == firstValue);
77         Assert.True(PartialRead(value, 1, -1) == secondValue);
78
79         firstValue = 0;
80         secondValue = 6892;
81
82         value = PartialWrite(value, firstValue, 0, 1);
83         value = PartialWrite(value, secondValue, 1, -1);
84
85         Assert.True(PartialRead(value, 0, 1) == firstValue);
86         Assert.True(PartialRead(value, 1, -1) == secondValue);
87     }
88
89     {
90         uint firstValue = 1;
91         uint secondValue = 1543;
92
93         // Pack (join) two values at the same time
94         uint value = secondValue << 1 | firstValue;
95
96         uint unpackagedFirstValue = value & 1;
97         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
98
99         Assert.True(firstValue == unpackagedFirstValue);
100        Assert.True(secondValue == unpackagedSecondValue);
101
102        // Using universal functions:
103
104        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
105        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
106
107        firstValue = 0;
108        secondValue = 6892;
109
110        value = Bit.PartialWrite(value, firstValue, 0, 1);
111        value = Bit.PartialWrite(value, secondValue, 1, -1);
112
113        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
114        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
115    }
116
117    {
118        uint firstValue = 1;
119        uint secondValue = 1543;
120
121        // Pack (join) two values at the same time
122        uint value = secondValue << 1 | firstValue;
123
124        uint unpackagedFirstValue = value & 1;
125        uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;

```

```

126     Assert.True(firstValue == unpackagedFirstValue);
127     Assert.True(secondValue == unpackagedSecondValue);
128
129     // Using universal functions:
130
131     var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
132     var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
133     var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
134     var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
135
136     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
137     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
138
139     firstValue = 0;
140     secondValue = 6892;
141
142     value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
143     value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
144
145     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
146     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
147 }
148
149 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
150 private static uint PartialWrite(uint target, uint source, int shift, int limit)
151 {
152     if (shift < 0)
153     {
154         shift = 32 + shift;
155     }
156     if (limit < 0)
157     {
158         limit = 32 + limit;
159     }
160     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
161     var targetMask = ~(sourceMask << shift);
162     return target & targetMask | (source & sourceMask) << shift;
163 }
164
165 private static uint PartialRead(uint target, int shift, int limit)
166 {
167     if (shift < 0)
168     {
169         shift = 32 + shift;
170     }
171     if (limit < 0)
172     {
173         limit = 32 + limit;
174     }
175     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
176     var targetMask = sourceMask << shift;
177     return (target & targetMask) >> shift;
178 }
179
180 private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
181 {
182     if (shift < 0)
183     {
184         shift = 32 + shift;
185     }
186     if (limit < 0)
187     {
188         limit = 32 + limit;
189     }
190     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
191     var targetMask = ~(sourceMask << shift);
192     return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
193 }
194
195 private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
196 {
197     if (shift < 0)
198     {
199         shift = 32 + shift;
200     }
201     if (limit < 0)
202     {
203

```

```

205         limit = 32 + limit;
206     }
207     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
208     var targetMask = sourceMask << shift;
209     return new Tuple<uint, int>(targetMask, shift);
210 }
211
212 private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
213
214 private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
215
216 private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
217
218 private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
219 }
220 }

```

./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 }

```

./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 }

```

./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

```

```

14     value |= 1;
15
16     Assert.True(value == 1);
17
18     // Set zero to first bit
19     value &= 0xFFFFFFFF;
20
21     // Get first bit
22     uint read = value & 1;
23
24     Assert.True(read == 0);
25
26     uint firstValue = 1;
27     uint secondValue = 1543;
28
29     // Pack (join) two values at the same time
30     value = (secondValue << 1) | firstValue;
31
32     uint unpackagedFirstValue = value & 1;
33     uint unpackagedSecondValue = (value & 0xFFFFFFFF) >> 1;
34
35     Assert.True(firstValue == unpackagedFirstValue);
36     Assert.True(secondValue == unpackagedSecondValue);
37
38     // Using universal functions:
39
40     Assert.True(PartialRead(value, 0, 1) == firstValue);
41     Assert.True(PartialRead(value, 1, -1) == secondValue);
42
43     firstValue = 0;
44     secondValue = 6892;
45
46     value = PartialWrite(value, firstValue, 0, 1);
47     value = PartialWrite(value, secondValue, 1, -1);
48
49     Assert.True(PartialRead(value, 0, 1) == firstValue);
50     Assert.True(PartialRead(value, 1, -1) == secondValue);
51 }
52
53 private static uint PartialWrite(uint target, uint source, int shift, int limit)
54 {
55     if (shift < 0)
56     {
57         shift = 32 + shift;
58     }
59     if (limit < 0)
60     {
61         limit = 32 + limit;
62     }
63     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
64     var targetMask = ~(sourceMask << shift);
65     return (target & targetMask) | ((source & sourceMask) << shift);
66 }
67
68 private static uint PartialRead(uint target, int shift, int limit)
69 {
70     if (shift < 0)
71     {
72         shift = 32 + shift;
73     }
74     if (limit < 0)
75     {
76         limit = 32 + limit;
77     }
78     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
79     var targetMask = sourceMask << shift;
80     return (target & targetMask) >> shift;
81 }
82 }
83 }

```

Index

- ./Platform.Numbers.Tests/ArithmeticExtensionsTests.cs, 9
- ./Platform.Numbers.Tests/ArithmeticTests.cs, 10
- ./Platform.Numbers.Tests/BitTests.cs, 10
- ./Platform.Numbers.Tests/MathExtensionsTests.cs, 13
- ./Platform.Numbers.Tests/MathTests.cs, 13
- ./Platform.Numbers.Tests/SystemTests.cs, 13
- ./Platform.Numbers/Arithmetic.cs, 1
- ./Platform.Numbers/ArithmeticExtensions.cs, 1
- ./Platform.Numbers/Arithmetic[T].cs, 1
- ./Platform.Numbers/Bit.cs, 2
- ./Platform.Numbers/BitExtensions.cs, 2
- ./Platform.Numbers/Bit[T].cs, 2
- ./Platform.Numbers/Integer.cs, 6
- ./Platform.Numbers/Integer[T].cs, 6
- ./Platform.Numbers/Math.cs, 8
- ./Platform.Numbers/MathExtensions.cs, 9
- ./Platform.Numbers/Math[T].cs, 9