

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, T> And;
14         public static readonly Func<T, T, int, int, T> PartialWrite;
15         public static readonly Func<T, int, int, T> PartialRead;
16
17         static Bit()
18         {
19             And = CompileAndDelegate();
20             PartialWrite = CompilePartialWriteDelegate();
21             PartialRead = CompilePartialReadDelegate();
22         }
23
24         private static Func<T, T, T> CompileAndDelegate()
25         {
26             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
27             {
28                 Ensure.Always.IsNumeric<T>();
29                 emitter.LoadArguments(0, 1);
30                 emitter.And();
31                 emitter.Return();
32             });
33         }
34
35         private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
36         {
37             return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
38             {
39                 Ensure.Always.IsNumeric<T>();
40                 var constants = GetConstants();
41                 var bitsNumber = constants.Item1;
42                 var numberFilledWithOnes = constants.Item2;
43                 ushort shiftArgument = 2;
44                 ushort limitArgument = 3;
45                 var checkLimit = emitter.DefineLabel();
46                 var calculateSourceMask = emitter.DefineLabel();
47                 // Check shift
48                 emitter.LoadArgument(shiftArgument);
49                 emitter.LoadConstant(0);
50                 emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
51                 // Fix shift
52                 emitter.LoadConstant(bitsNumber);
53                 emitter.LoadArgument(shiftArgument);
54                 emitter.Add();
55                 emitter.StoreArgument(shiftArgument);
56                 emitter.MarkLabel(checkLimit);
57                 // Check limit
58                 emitter.LoadArgument(limitArgument);
59                 emitter.LoadConstant(0);
60                 emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
61                 // Fix limit
62                 emitter.LoadConstant(bitsNumber);
63                 emitter.LoadArgument(limitArgument);
64                 emitter.Add();
65                 emitter.StoreArgument(limitArgument);
66                 emitter.MarkLabel(calculateSourceMask);
67                 var sourceMask = emitter.DeclareLocal<T>();
68                 var targetMask = emitter.DeclareLocal<T>();
69                 //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
70                 LoadMaxValueConstant(emitter);
71                 emitter.LoadArgument(limitArgument);
72                 emitter.ShiftLeft();
73                 emitter.Not();
74                 //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
75                 LoadMaxValueConstant(emitter);
76                 emitter.And();
77                 emitter.StoreLocal(sourceMask);
78                 emitter.LoadLocal(sourceMask);
```

```

79         emitter.LoadArgument(shiftArgument);
80         emitter.ShiftLeft();
81         emitter.Not();
82         emitter.StoreLocal(targetMask);
83         emitter.LoadArgument(0); // target
84         emitter.LoadLocal(targetMask);
85         emitter.And();
86         emitter.LoadArgument(1); // source
87         emitter.LoadLocal(sourceMask);
88         emitter.And();
89         emitter.LoadArgument(shiftArgument);
90         emitter.ShiftLeft();
91         emitter.Or();
92         emitter.Return();
93     });
94 }
95
96 private static Func<T, int, int, T> CompilePartialReadDelegate()
97 {
98     return DelegateHelpers.Compile<Func<T, int, int, T>>(emitter =>
99     {
100         Ensure.Always.IsNumeric<T>();
101         var constants = GetConstants();
102         var bitsNumber = constants.Item1;
103         var numberFilledWithOnes = constants.Item2;
104         ushort shiftArgument = 1;
105         ushort limitArgument = 2;
106         var checkLimit = emitter.DefineLabel();
107         var calculateSourceMask = emitter.DefineLabel();
108         // Check shift
109         emitter.LoadArgument(shiftArgument);
110         emitter.LoadConstant(0);
111         emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
112         // Fix shift
113         emitter.LoadConstant(bitsNumber);
114         emitter.LoadArgument(shiftArgument);
115         emitter.Add();
116         emitter.StoreArgument(shiftArgument);
117         emitter.MarkLabel(checkLimit);
118         // Check limit
119         emitter.LoadArgument(limitArgument);
120         emitter.LoadConstant(0);
121         emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
122         // Fix limit
123         emitter.LoadConstant(bitsNumber);
124         emitter.LoadArgument(limitArgument);
125         emitter.Add();
126         emitter.StoreArgument(limitArgument);
127         emitter.MarkLabel(calculateSourceMask);
128         var sourceMask = emitter.DeclareLocal<T>();
129         var targetMask = emitter.DeclareLocal<T>();
130         //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
131         LoadMaxValueConstant(emitter);
132         emitter.LoadArgument(limitArgument); // limit
133         emitter.ShiftLeft();
134         emitter.Not();
135         //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
136         LoadMaxValueConstant(emitter);
137         emitter.And();
138         emitter.StoreLocal(sourceMask);
139         emitter.LoadLocal(sourceMask);
140         emitter.LoadArgument(shiftArgument);
141         emitter.ShiftLeft();
142         emitter.StoreLocal(targetMask);
143         emitter.LoadArgument(0); // target
144         emitter.LoadLocal(targetMask);
145         emitter.And();
146         emitter.LoadArgument(shiftArgument);
147         emitter.ShiftRight();
148         emitter.Return();
149     });
150 }
151
152 private static void LoadMaxValueConstant(ILGenerator emitter)
153 {
154     var type = typeof(T);
155     if (type == typeof(ulong))
156     {

```

```

157         emitter.Emit(OpCodes.Ldc_I8, unchecked((long)ulong.MaxValue));
158     }
159     else if (type == typeof(uint))
160     {
161         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)uint.MaxValue));
162     }
163     else if (type == typeof(ushort))
164     {
165         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)ushort.MaxValue));
166     }
167     else if (type == typeof(byte))
168     {
169         emitter.Emit(OpCodes.Ldc_I4_S, unchecked((sbyte)byte.MaxValue));
170     }
171     else
172     {
173         throw new NotSupportedException();
174     }
175 }
176
177 private static Tuple<int, T> GetConstants()
178 {
179     var type = typeof(T);
180     if (type == typeof(ulong))
181     {
182         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
183     }
184     if (type == typeof(uint))
185     {
186         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
187     }
188     if (type == typeof(ushort))
189     {
190         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
191     }
192     if (type == typeof(byte))
193     {
194         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
195     }
196     throw new NotSupportedException();
197 }
198 }
199 }

```

./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         {
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             public Integer(T value) => Value = value;
40
41             public static implicit operator 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             public static implicit operator T(Integer<T> integer) => integer.Value;
52             public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);
53             public static implicit operator Integer<T>(ulong integer) => _create(integer);
54             public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);
55             public static implicit operator Integer<T>(long integer) => To.UInt64(integer);
56             public static implicit operator Integer<T>(uint integer) => new Integer(integer);
57             public static implicit operator Integer<T>(int integer) => To.UInt64(integer);
58             public static implicit operator Integer<T>(ushort integer) => new Integer(integer);
59             public static implicit operator Integer<T>(short integer) => To.UInt64(integer);
60             public static implicit operator Integer<T>(byte integer) => new Integer(integer);
61             public static implicit operator Integer<T>(sbyte integer) => To.UInt64(integer);
62             public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);
63             public static implicit operator long(Integer<T> integer) => To.Int64(integer);
64             public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);
65
66
67
68
69
70
71
72
73
74
75
76
77
78
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     }
37 }

```



```

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 PartialReadWriteTest()
20 {
21     {
22         uint firstValue = 1;
23         uint secondValue = 1543;
24
25         // Pack (join) two values at the same time
26         uint value = secondValue << 1 | firstValue;
27
28         uint unpackagedFirstValue = value & 1;
29         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
30
31         Assert.True(firstValue == unpackagedFirstValue);
32         Assert.True(secondValue == unpackagedSecondValue);
33
34         // Using universal functions:
35
36         Assert.True(PartialRead(value, 0, 1) == firstValue);
37         Assert.True(PartialRead(value, 1, -1) == secondValue);
38
39         firstValue = 0;
40         secondValue = 6892;
41
42         value = PartialWrite(value, firstValue, 0, 1);
43         value = PartialWrite(value, secondValue, 1, -1);
44
45         Assert.True(PartialRead(value, 0, 1) == firstValue);
46         Assert.True(PartialRead(value, 1, -1) == secondValue);
47     }
48
49     {
50         uint firstValue = 1;
51         uint secondValue = 1543;
52
53         // Pack (join) two values at the same time
54         uint value = secondValue << 1 | firstValue;
55
56         uint unpackagedFirstValue = value & 1;
57         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
58
59         Assert.True(firstValue == unpackagedFirstValue);
60         Assert.True(secondValue == unpackagedSecondValue);
61
62         // Using universal functions:
63
64         Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
65         Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
66
67         firstValue = 0;
68         secondValue = 6892;
69
70         value = Bit.PartialWrite(value, firstValue, 0, 1);
71         value = Bit.PartialWrite(value, secondValue, 1, -1);
72
73         Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
74         Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
75     }
76
77     {
78         uint firstValue = 1;
79         uint secondValue = 1543;
80
81         // Pack (join) two values at the same time
82         uint value = secondValue << 1 | firstValue;
83
84         uint unpackagedFirstValue = value & 1;
85         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
86
87         Assert.True(firstValue == unpackagedFirstValue);
88         Assert.True(secondValue == unpackagedSecondValue);
89
90         // Using universal functions:
91
92         var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
93         var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
94         var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
95         var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
96
97         Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
98         Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);

```

```

99         firstValue = 0;
100         secondValue = 6892;
101
102         value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
103         value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
104
105         Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
106         Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
107     }
108 }
109
110 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
111 private static uint PartialWrite(uint target, uint source, int shift, int limit)
112 {
113     if (shift < 0)
114     {
115         shift = 32 + shift;
116     }
117     if (limit < 0)
118     {
119         limit = 32 + limit;
120     }
121     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
122     var targetMask = ~(sourceMask << shift);
123     return target & targetMask | (source & sourceMask) << shift;
124 }
125
126 private static uint PartialRead(uint target, int shift, int limit)
127 {
128     if (shift < 0)
129     {
130         shift = 32 + shift;
131     }
132     if (limit < 0)
133     {
134         limit = 32 + limit;
135     }
136     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
137     var targetMask = sourceMask << shift;
138     return (target & targetMask) >> shift;
139 }
140
141 private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
142 {
143     if (shift < 0)
144     {
145         shift = 32 + shift;
146     }
147     if (limit < 0)
148     {
149         limit = 32 + limit;
150     }
151     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
152     var targetMask = ~(sourceMask << shift);
153     return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
154 }
155
156 private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
157 {
158     if (shift < 0)
159     {
160         shift = 32 + shift;
161     }
162     if (limit < 0)
163     {
164         limit = 32 + limit;
165     }
166     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
167     var targetMask = sourceMask << shift;
168     return new Tuple<uint, int>(targetMask, shift);
169 }
170
171 private static uint PartialWrite(uint target, uint targetMask, uint source, uint
172     ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
173
174 private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
175     ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
176     ↪ masksAndShift.Item2, masksAndShift.Item3);

```

```

175         private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
176             ↪ targetMask) >> shift;
177
178         private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
179             ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
180     }

```

./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, 9
- ./Platform.Numbers.Tests/BitTests.cs, 9
- ./Platform.Numbers.Tests/MathExtensionsTests.cs, 12
- ./Platform.Numbers.Tests/MathTests.cs, 12
- ./Platform.Numbers.Tests/SystemTests.cs, 12
- ./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, 5
- ./Platform.Numbers/Integer[T].cs, 5
- ./Platform.Numbers/Math.cs, 7
- ./Platform.Numbers/MathExtensions.cs, 8
- ./Platform.Numbers/Math[T].cs, 8