

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 And<T>(T x, T y) => Arithmetic<T>.And(x, y);
9          public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
10         public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
11         public static T Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);
12         public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
13     }
14 }
```

./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> And;
14         public static readonly Func<T, T, T> Subtract;
15         public static readonly Func<T, T> Increment;
16         public static readonly Func<T, T> Decrement;
17
18         static Arithmetic()
19         {
20             Add = CompileAddDelegate();
21             And = CompileAndDelegate();
22             Subtract = CompileSubtractDelegate();
23             Increment = CompileIncrementDelegate();
24             Decrement = CompileDecrementDelegate();
25         }
26
27         private static Func<T, T, T> CompileAddDelegate()
28         {
29             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
30             {
31                 Ensure.Always.IsNumeric<T>();
32                 emitter.LoadArguments(0, 1);
33                 emitter.Add();
34                 emitter.Return();
35             });
36         }
37
38         private static Func<T, T, T> CompileAndDelegate()
39         {
40             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
41             {
42                 Ensure.Always.IsNumeric<T>();
43                 emitter.LoadArguments(0, 1);
44                 emitter.And();
45                 emitter.Return();
46             });
47         }
48     }
49 }
```

```

48
49 private static Func<T, T, T> CompileSubtractDelegate()
50 {
51     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
52     {
53         Ensure.Always.IsNumeric<T>();
54         emitter.LoadArguments(0, 1);
55         emitter.Subtract();
56         emitter.Return();
57     });
58 }
59
60 private static Func<T, T> CompileIncrementDelegate()
61 {
62     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
63     {
64         Ensure.Always.IsNumeric<T>();
65         emitter.LoadArgument(0);
66         emitter.Increment<T>();
67         emitter.Return();
68     });
69 }
70
71 private static Func<T, T> CompileDecrementDelegate()
72 {
73     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
74     {
75         Ensure.Always.IsNumeric<T>();
76         emitter.LoadArgument(0);
77         emitter.Decrement<T>();
78         emitter.Return();
79     });
80 }
81 }
82 }

```

./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 PartialWrite<T>(T target, T source, int shift, int limit) =>
34             ⇨ Bit<T>.PartialWrite(target, source, shift, limit);
35
36         public static T PartialRead<T>(T target, int shift, int limit) =>
37             ⇨ Bit<T>.PartialRead(target, shift, limit);
38     }
39 }

```

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

```

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

```

```

143     {
144         emitter.Emit(OpCodes.Ldc_I8, unchecked((long)ulong.MaxValue));
145     }
146     else if (type == typeof(uint))
147     {
148         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)uint.MaxValue));
149     }
150     else if (type == typeof(ushort))
151     {
152         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)ushort.MaxValue));
153     }
154     else if (type == typeof(byte))
155     {
156         emitter.Emit(OpCodes.Ldc_I4_S, unchecked((sbyte)byte.MaxValue));
157     }
158     else
159     {
160         throw new NotSupportedException();
161     }
162 }

```

```

164 private static Tuple<int, T> GetConstants()
165 {
166     var type = typeof(T);
167     if (type == typeof(ulong))
168     {
169         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
170     }
171     if (type == typeof(uint))
172     {
173         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
174     }
175     if (type == typeof(ushort))
176     {
177         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
178     }
179     if (type == typeof(byte))
180     {
181         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
182     }
183     throw new NotSupportedException();
184 }
185 }
186 }

```

./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
65
66
67
68
69
70
71
72
73
74
75
76
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     }
37 }

```



```

33     });
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 }

```

```

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

```

```

174     private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
        ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
        ↪ masksAndShift.Item2, masksAndShift.Item3);
175
176     private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
        ↪ targetMask) >> shift;
177
178     private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
        ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
179 }
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 & 0xFFFFFFF0) >> 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, 3
- ./Platform.Numbers/Integer.cs, 5
- ./Platform.Numbers/Integer[T].cs, 6
- ./Platform.Numbers/Math.cs, 7
- ./Platform.Numbers/MathExtensions.cs, 8
- ./Platform.Numbers/Math[T].cs, 8