

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  using Platform.Reflection.Sigil;
5
6  // ReSharper disable StaticFieldInGenericType
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Numbers
10 {
11     public static class Arithmetic<T>
12     {
13         public static readonly Func<T, T, T> Add;
14         public static readonly Func<T, T, T> And;
15         public static readonly Func<T, T> Increment;
16         public static readonly Func<T, T, T> Subtract;
17         public static readonly Func<T, T> Decrement;
18
19         static Arithmetic()
20         {
21             Add = DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
22             {
23                 Ensure.Always.IsNumeric<T>();
24                 emitter.LoadArguments(0, 1);
25                 emitter.Add();
26                 emitter.Return();
27             });
28             And = DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
29             {
30                 Ensure.Always.IsNumeric<T>();
31                 emitter.LoadArguments(0, 1);
32                 emitter.And();
33                 emitter.Return();
34             });
35             Increment = DelegateHelpers.Compile<Func<T, T>>(emitter =>
36             {
37                 Ensure.Always.IsNumeric<T>();
38                 emitter.LoadArgument(0);
39                 emitter.Increment(typeof(T));
40                 emitter.Return();
41             });
42             Subtract = DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
43             {
44                 Ensure.Always.IsNumeric<T>();
45                 emitter.LoadArguments(0, 1);
46                 emitter.Subtract();
47                 emitter.Return();
48             });
49         }
50     }
51 }
```

```

48     });
49     Decrement = DelegateHelpers.Compile<Func<T, T>>(emitter =>
50     {
51         Ensure.Always.IsNumeric<T>();
52         emitter.LoadArgument(0);
53         emitter.Decrement(typeof(T));
54         emitter.Return();
55     });
56 }
57 }
58 }

```

./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 Platform.Exceptions;
3  using Platform.Reflection;
4  using Platform.Reflection.Sigil;
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 = DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
19         {
20             Ensure.Always.IsNumeric<T>();
21             var constants = GetConstants();
22             var bitsNumber = constants.Item1;
23             var numberFilledWithOnes = constants.Item2;
24             ushort shiftArgument = 2;
25             ushort limitArgument = 3;
26             var checkLimit = emitter.DefineLabel();
27             var calculateSourceMask = emitter.DefineLabel();
28             // Check shift
29             emitter.LoadArgument(shiftArgument);
30             emitter.LoadConstant(0);
31             emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
32             // Fix shift
33             emitter.LoadConstant(bitsNumber);
34             emitter.LoadArgument(shiftArgument);
35             emitter.Add();
36             emitter.StoreArgument(shiftArgument);
37             emitter.MarkLabel(checkLimit);
38             // Check limit
39             emitter.LoadArgument(limitArgument);
40             emitter.LoadConstant(0);
41             emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
42             // Fix limit
43             emitter.LoadConstant(bitsNumber);
44             emitter.LoadArgument(limitArgument);
45             emitter.Add();
46             emitter.StoreArgument(limitArgument);
47             emitter.MarkLabel(calculateSourceMask);
48             using (var sourceMask = emitter.DeclareLocal<T>())
49             using (var targetMask = emitter.DeclareLocal<T>())
50             {
51                 emitter.LoadConstant(typeof(T), numberFilledWithOnes);
52                 emitter.LoadArgument(limitArgument);
53                 emitter.ShiftLeft();
54                 emitter.Not();
55                 emitter.LoadConstant(typeof(T), numberFilledWithOnes);
56                 emitter.And();
57                 emitter.StoreLocal(sourceMask);
58                 emitter.LoadLocal(sourceMask);
59                 emitter.LoadArgument(shiftArgument);
60                 emitter.ShiftLeft();
61                 emitter.Not();
62                 emitter.StoreLocal(targetMask);
63                 emitter.LoadArgument(0); // target
64                 emitter.LoadLocal(targetMask);
65                 emitter.And();
66                 emitter.LoadArgument(1); // source
67                 emitter.LoadLocal(sourceMask);
68                 emitter.And();
69                 emitter.LoadArgument(shiftArgument);
70                 emitter.ShiftLeft();
71                 emitter.Or();
72             }
73             emitter.Return();
74         });
75         PartialRead = DelegateHelpers.Compile<Func<T, int, int, T>>(emitter =>
76         {
77             Ensure.Always.IsNumeric<T>();
78             var constants = GetConstants();
79             var bitsNumber = constants.Item1;
80             var numberFilledWithOnes = constants.Item2;
81             ushort shiftArgument = 1;
82             ushort limitArgument = 2;
83             var checkLimit = emitter.DefineLabel();
84             var calculateSourceMask = emitter.DefineLabel();
85             // Check shift
86             emitter.LoadArgument(shiftArgument);
87             emitter.LoadConstant(0);
88             emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
89             // Fix shift

```

```

90     emitter.LoadConstant(bitsNumber);
91     emitter.LoadArgument(shiftArgument);
92     emitter.Add();
93     emitter.StoreArgument(shiftArgument);
94     emitter.MarkLabel(checkLimit);
95     // Check limit
96     emitter.LoadArgument(limitArgument);
97     emitter.LoadConstant(0);
98     emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
99     // Fix limit
100    emitter.LoadConstant(bitsNumber);
101    emitter.LoadArgument(limitArgument);
102    emitter.Add();
103    emitter.StoreArgument(limitArgument);
104    emitter.MarkLabel(calculateSourceMask);
105    using (var sourceMask = emitter.DeclareLocal<T>())
106    using (var targetMask = emitter.DeclareLocal<T>())
107    {
108        emitter.LoadConstant(typeof(T), numberFilledWithOnes);
109        emitter.LoadArgument(limitArgument); // limit
110        emitter.ShiftLeft();
111        emitter.Not();
112        emitter.LoadConstant(typeof(T), numberFilledWithOnes);
113        emitter.And();
114        emitter.StoreLocal(sourceMask);
115        emitter.LoadLocal(sourceMask);
116        emitter.LoadArgument(shiftArgument);
117        emitter.ShiftLeft();
118        emitter.StoreLocal(targetMask);
119        emitter.LoadArgument(0); // target
120        emitter.LoadLocal(targetMask);
121        emitter.And();
122        emitter.LoadArgument(shiftArgument);
123        emitter.ShiftRight();
124    }
125    emitter.Return();
126    });
127 }
128
129 private static Tuple<int, T> GetConstants()
130 {
131     var type = typeof(T);
132     if (type == typeof(ulong))
133     {
134         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
135     }
136     if (type == typeof(uint))
137     {
138         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
139     }
140     if (type == typeof(ushort))
141     {
142         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
143     }
144     if (type == typeof(byte))
145     {
146         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
147     }
148     throw new NotSupportedException();
149 }
150 }
151 }

```

./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.Linq;
3  using System.Reflection;
4  using System.Collections.Generic;
5  using Platform.Exceptions;
6  using Platform.Reflection;
7  using Platform.Reflection.Sigil;
8  using Platform.Converters;
9
10 // ReSharper disable StaticFieldInGenericType
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Numbers
14 {
15     public struct Integer<T> : IEquatable<Integer<T>>
16     {
17         private static readonly EqualityComparer<T> _equalityComparer =
18             ↪ EqualityComparer<T>.Default;
19         private static readonly Func<ulong, Integer<T>> _create;
20
21         public static readonly T Zero;
22         public static readonly T One;
23         public static readonly T Two;
24
25         public readonly T Value;
26
27         static Integer()
28         {
29             _create = DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emitter =>
30             {
31                 if (typeof(T) != typeof(Integer))
32                 {
33                     Ensure.Always.CanBeNumeric<T>();
34                 }
35                 emitter.LoadArgument(0);
36                 if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))
37                 {
38                     emitter.Call(typeof(To).GetTypeInfo().GetMethod(typeof(T).Name,
39                     ↪ Types<ulong>.Array.ToArray()));
40                 }
41                 if (Type<T>.IsNullable)
42                 {
43                     emitter.NewObject(typeof(T), Type<T>.UnderlyingType);
44                 }
45                 if (typeof(T) == typeof(Integer))
46                 {
47                     emitter.NewObject(typeof(Integer), typeof(ulong));
48                 }
49                 emitter.NewObject(typeof(Integer<T>), typeof(T));
50                 emitter.Return();
51             });
52             try
53             {
54                 Zero = default;
55                 One = Arithmetic.Increment(Zero);
56                 Two = Arithmetic.Increment(One);
57             }
58             catch (Exception exception)
59             {
60             }
61         }
62     }
63 }

```

```

58         exception.Ignore();
59     }
60 }
61
62 public Integer(T value) => Value = value;
63
64 public static implicit operator Integer(Integer<T> integer)
65 {
66     if (typeof(T) == typeof(Integer))
67     {
68         return (Integer)(object)integer.Value;
69     }
70     return Convert.ToUInt64(integer.Value);
71 }
72
73 public static implicit operator ulong(Integer<T> integer) => ((Integer)integer).Value;
74
75 public static implicit operator T(Integer<T> integer) => integer.Value;
76
77 public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);
78
79 public static implicit operator Integer<T>(ulong integer) => _create(integer);
80
81 public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);
82
83 public static implicit operator Integer<T>(long integer) => To.UInt64(integer);
84
85 public static implicit operator Integer<T>(uint integer) => new Integer(integer);
86
87 public static implicit operator Integer<T>(int integer) => To.UInt64(integer);
88
89 public static implicit operator Integer<T>(ushort integer) => new Integer(integer);
90
91 public static implicit operator Integer<T>(short integer) => To.UInt64(integer);
92
93 public static implicit operator Integer<T>(byte integer) => new Integer(integer);
94
95 public static implicit operator Integer<T>(sbyte integer) => To.UInt64(integer);
96
97 public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);
98
99 public static implicit operator long(Integer<T> integer) => To.Int64(integer);
100
101 public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);
102
103 public static implicit operator int(Integer<T> integer) => To.Int32(integer);
104
105 public static implicit operator ushort(Integer<T> integer) => To.UInt16(integer);
106
107 public static implicit operator short(Integer<T> integer) => To.Int16(integer);
108
109 public static implicit operator byte(Integer<T> integer) => To.Byte(integer);
110
111 public static implicit operator sbyte(Integer<T> integer) => To.SByte(integer);
112
113 public static implicit operator bool(Integer<T> integer) => To.Boolean(integer);
114
115 public bool Equals(Integer<T> other) => _equalityComparer.Equals(Value, other.Value);
116
117 public override string ToString() => Value.ToString();
118 }
119 }

```

./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  using Platform.Reflection.Sigil;
6
7  // ReSharper disable StaticFieldInGenericType
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Numbers
11 {
12     public static class Math<T>
13     {
14         public static readonly Func<T, T> Abs;
15         public static readonly Func<T, T> Negate;
16
17         static Math()
18         {
19             Abs = DelegateHelpers.Compile<Func<T, T>>(emitter =>

```

```

20     {
21         Ensure.Always.IsNumeric<T>();
22         emitter.LoadArgument(0);
23         if (Type<T>.IsSigned)
24         {
25             emitter.Call(typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
26                 ↳ typeof(T) }));
27             emitter.Return();
28         });
29         Negate = DelegateHelpers.Compile<Func<T, T>>(emitter =>
30         {
31             Ensure.Always.IsSigned<T>();
32             emitter.LoadArgument(0);
33             emitter.Negate();
34             emitter.Return();
35         });
36     }
37 }
38 }

```

./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
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     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);
177
178     private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
179     ↪ targetMask) >> shift;
180
181     private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
182     ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
183 }

```

./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, 8
- ./Platform.Numbers.Tests/ArithmeticTests.cs, 8
- ./Platform.Numbers.Tests/BitTests.cs, 8
- ./Platform.Numbers.Tests/MathExtensionsTests.cs, 11
- ./Platform.Numbers.Tests/MathTests.cs, 11
- ./Platform.Numbers.Tests/SystemTests.cs, 11
- ./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, 4
- ./Platform.Numbers/Integer[T].cs, 5
- ./Platform.Numbers/Math.cs, 6
- ./Platform.Numbers/MathExtensions.cs, 7
- ./Platform.Numbers/Math[T].cs, 7