

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 Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
9          public static T Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);
10         public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
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 Not<T>(T x) => Bit<T>.Not(x);
34
35         public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
36
37         public static T And<T>(T x, T y) => Bit<T>.And(x, y);
38
39         public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
40
41         public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
42
43         public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
44             ↪ Bit<T>.PartialWrite(target, source, shift, limit);
45
46         public static T PartialRead<T>(T target, int shift, int limit) =>
47             ↪ Bit<T>.PartialRead(target, shift, limit);
48     }
49 }

```

./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 Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
8         public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
9             ↪ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
10        public static T PartialRead<T>(this T target, int shift, int limit) =>
11            ↪ Bit<T>.PartialRead(target, shift, limit);
12    }
13 }

```

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

```

1  using System;
2  using System.Reflection.Emit;
3  using Platform.Exceptions;
4  using Platform.Reflection;
5
6  // ReSharper disable StaticFieldInGenericType
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Numbers
10 {
11     public static class Bit<T>
12     {
13         public static readonly Func<T, T> Not;
14         public static readonly Func<T, T, T> Or;
15         public static readonly Func<T, T, T> And;
16         public static readonly Func<T, int, T> ShiftLeft;
17         public static readonly Func<T, int, T> ShiftRight;
18         public static readonly Func<T, T, int, int, T> PartialWrite;
19         public static readonly Func<T, int, int, T> PartialRead;
20
21         static Bit()
22         {
23             Not = CompileNotDelegate();
24             Or = CompileOrDelegate();
25             And = CompileAndDelegate();
26             ShiftLeft = CompileShiftLeftDelegate();
27             ShiftRight = CompileShiftRightDelegate();
28             PartialWrite = CompilePartialWriteDelegate();
29             PartialRead = CompilePartialReadDelegate();
30         }
31
32         private static Func<T, T> CompileNotDelegate()
33         {
34             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
35             {
36                 Ensure.Always.IsNumeric<T>();
37                 emitter.LoadArguments(0);
38                 emitter.Not();
39                 emitter.Return();
40             });
41         }
42
43         private static Func<T, T, T> CompileOrDelegate()
44         {
45             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
46             {
47                 Ensure.Always.IsNumeric<T>();
48                 emitter.LoadArguments(0, 1);
49                 emitter.Or();
50                 emitter.Return();
51             });
52         }
53
54         private static Func<T, T, T> CompileAndDelegate()
55         {
56             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
57             {
58                 Ensure.Always.IsNumeric<T>();
59                 emitter.LoadArguments(0, 1);
60                 emitter.And();
61                 emitter.Return();
62             });
63         }
64
65         private static Func<T, int, T> CompileShiftLeftDelegate()
66         {
67             return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
68             {
69                 Ensure.Always.IsNumeric<T>();

```

```

70         emitter.LoadArguments(0, 1);
71         emitter.ShiftLeft();
72         emitter.Return();
73     });
74 }
75
76 private static Func<T, int, T> CompileShiftRightDelegate()
77 {
78     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
79     {
80         Ensure.Always.IsNumeric<T>();
81         emitter.LoadArguments(0, 1);
82         emitter.ShiftRight();
83         emitter.Return();
84     });
85 }
86
87 private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
88 {
89     return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
90     {
91         Ensure.Always.IsNumeric<T>();
92         var constants = GetConstants();
93         var bitsNumber = constants.Item1;
94         var numberFilledWithOnes = constants.Item2;
95         ushort shiftArgument = 2;
96         ushort limitArgument = 3;
97         var checkLimit = emitter.DefineLabel();
98         var calculateSourceMask = emitter.DefineLabel();
99         // Check shift
100        emitter.LoadArgument(shiftArgument);
101        emitter.LoadConstant(0);
102        emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
103        // Fix shift
104        if (typeof(T) == typeof(byte))
105        {
106            emitter.Emit(OpCodes.Ldc_I4_8);
107        }
108        else
109        {
110            emitter.LoadConstant(bitsNumber);
111        }
112        emitter.LoadArgument(shiftArgument);
113        emitter.Add();
114        emitter.StoreArgument(shiftArgument);
115        emitter.MarkLabel(checkLimit);
116        // Check limit
117        emitter.LoadArgument(limitArgument);
118        emitter.LoadConstant(0);
119        emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
120        // Fix limit
121        if (typeof(T) == typeof(byte))
122        {
123            emitter.Emit(OpCodes.Ldc_I4_8);
124        }
125        else
126        {
127            emitter.LoadConstant(bitsNumber);
128        }
129        emitter.LoadArgument(limitArgument);
130        emitter.Add();
131        emitter.StoreArgument(limitArgument);
132        emitter.MarkLabel(calculateSourceMask);
133        var sourceMask = emitter.DeclareLocal<T>();
134        var targetMask = emitter.DeclareLocal<T>();
135        //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
136        LoadMaxValueConstant(emitter);
137        emitter.LoadArgument(limitArgument);
138        emitter.ShiftLeft();
139        emitter.Not();
140        //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
141        LoadMaxValueConstant(emitter);
142        emitter.And();
143        emitter.StoreLocal(sourceMask);
144        emitter.LoadLocal(sourceMask);
145        emitter.LoadArgument(shiftArgument);
146        emitter.ShiftLeft();
147        emitter.Not();

```

```

148         emitter.StoreLocal(targetMask);
149         emitter.LoadArgument(0); // target
150         emitter.LoadLocal(targetMask);
151         emitter.And();
152         emitter.LoadArgument(1); // source
153         emitter.LoadLocal(sourceMask);
154         emitter.And();
155         emitter.LoadArgument(shiftArgument);
156         emitter.ShiftLeft();
157         emitter.Or();
158         emitter.Return();
159     });
160 }
161
162 private static Func<T, int, int, T> CompilePartialReadDelegate()
163 {
164     return DelegateHelpers.Compile<Func<T, int, int, T>>(emitter =>
165     {
166         Ensure.Always.IsNumeric<T>();
167         var constants = GetConstants();
168         var bitsNumber = constants.Item1;
169         var numberFilledWithOnes = constants.Item2;
170         ushort shiftArgument = 1;
171         ushort limitArgument = 2;
172         var checkLimit = emitter.DefineLabel();
173         var calculateSourceMask = emitter.DefineLabel();
174         // Check shift
175         emitter.LoadArgument(shiftArgument);
176         emitter.LoadConstant(0);
177         emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
178         // Fix shift
179         emitter.LoadConstant(bitsNumber);
180         emitter.LoadArgument(shiftArgument);
181         emitter.Add();
182         emitter.StoreArgument(shiftArgument);
183         emitter.MarkLabel(checkLimit);
184         // Check limit
185         emitter.LoadArgument(limitArgument);
186         emitter.LoadConstant(0);
187         emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
188         // Fix limit
189         emitter.LoadConstant(bitsNumber);
190         emitter.LoadArgument(limitArgument);
191         emitter.Add();
192         emitter.StoreArgument(limitArgument);
193         emitter.MarkLabel(calculateSourceMask);
194         var sourceMask = emitter.DeclareLocal<T>();
195         var targetMask = emitter.DeclareLocal<T>();
196         //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
197         LoadMaxValueConstant(emitter);
198         emitter.LoadArgument(limitArgument); // limit
199         emitter.ShiftLeft();
200         emitter.Not();
201         //emitter.LoadConstant(typeof(T), numberFilledWithOnes);
202         LoadMaxValueConstant(emitter);
203         emitter.And();
204         emitter.StoreLocal(sourceMask);
205         emitter.LoadLocal(sourceMask);
206         emitter.LoadArgument(shiftArgument);
207         emitter.ShiftLeft();
208         emitter.StoreLocal(targetMask);
209         emitter.LoadArgument(0); // target
210         emitter.LoadLocal(targetMask);
211         emitter.And();
212         emitter.LoadArgument(shiftArgument);
213         emitter.ShiftRight();
214         emitter.Return();
215     });
216 }
217
218 private static void LoadMaxValueConstant(ILGenerator emitter)
219 {
220     var type = typeof(T);
221     if (type == typeof(ulong))
222     {
223         emitter.Emit(OpCodes.Ldc_I8, unchecked((long)ulong.MaxValue));
224     }
225     else if (type == typeof(uint))

```

```

226     {
227         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)uint.MaxValue));
228     }
229     else if (type == typeof(ushort))
230     {
231         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)ushort.MaxValue));
232     }
233     else if (type == typeof(byte))
234     {
235         emitter.Emit(OpCodes.Ldc_I4, unchecked((int)byte.MaxValue));
236     }
237     else
238     {
239         throw new NotSupportedException();
240     }
241 }
242
243 private static Tuple<int, T> GetConstants()
244 {
245     var type = typeof(T);
246     if (type == typeof(ulong))
247     {
248         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
249     }
250     if (type == typeof(uint))
251     {
252         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
253     }
254     if (type == typeof(ushort))
255     {
256         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
257     }
258     if (type == typeof(byte))
259     {
260         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
261     }
262     throw new NotSupportedException();
263 }
264 }
265 }

```

./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 Integer(ulong integer) => integer.Value;
22         public static implicit operator Integer(long integer) => To.Int64(integer.Value);
23         public static implicit operator Integer(uint integer) => To.UInt32(integer.Value);
24         public static implicit operator Integer(int integer) => To.Int32(integer.Value);
25         public static implicit operator Integer(ushort integer) => To.UInt16(integer.Value);
26         public static implicit operator Integer(short integer) => To.Int16(integer.Value);
27         public static implicit operator Integer(byte integer) => To.Byte(integer.Value);
28         public static implicit operator Integer(sbyte integer) => To.SByte(integer.Value);
29         public static implicit operator Integer(bool 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
40         public Integer(T value) => Value = value;
41
42         public static implicit operator Integer<T> (integer)
43         {
44             if (typeof(T) == typeof(Integer))
45             {
46                 return (Integer)(object)integer.Value;
47             }
48             return Convert.ToUInt64(integer.Value);
49         }
50
51         public static implicit operator ulong(Integer<T> integer) => ((Integer)integer).Value;
52         public static implicit operator T(Integer<T> integer) => integer.Value;
53
54         public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);
55         public static implicit operator Integer<T>(ulong integer) => _create(integer);
56         public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);
57         public static implicit operator Integer<T>(long integer) => To.UInt64(integer);
58         public static implicit operator Integer<T>(uint integer) => new Integer(integer);
59         public static implicit operator Integer<T>(int integer) => To.UInt64(integer);
60         public static implicit operator Integer<T>(ushort integer) => new Integer(integer);
61         public static implicit operator Integer<T>(short integer) => To.UInt64(integer);
62         public static implicit operator Integer<T>(byte integer) => new Integer(integer);
63         public static implicit operator Integer<T>(sbyte integer) => To.UInt64(integer);
64         public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);
65         public static implicit operator long(Integer<T> integer) => To.Int64(integer);
66         public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);
67         public static implicit operator int(Integer<T> integer) => To.Int32(integer);
68         public static implicit operator ushort(Integer<T> integer) => To.UInt16(integer);
69     }
70 }

```

```

83     public static implicit operator short(Integer<T> integer) => To.Int16(integer);
84     public static implicit operator byte(Integer<T> integer) => To.Byte(integer);
85     public static implicit operator sbyte(Integer<T> integer) => To.SByte(integer);
86     public static implicit operator bool(Integer<T> integer) => To.Boolean(integer);
87     public bool Equals(Integer<T> other) => _equalityComparer.Equals(Value, other.Value);
88     public override string ToString() => Value.ToString();
89     private static Func<ulong, Integer<T>> CompileCreateDelegate()
90     {
91         return DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emitter =>
92         {
93             if (typeof(T) != typeof(Integer))
94             {
95                 Ensure.Always.CanBeNumeric<T>();
96             }
97             emitter.LoadArgument(0);
98             if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))
99             {
100                 emitter.Call(typeof(To).GetTypeInfo().GetMethod(typeof(T).Name,
101                     ↪ Types<ulong>.Array));
102             }
103             if (NumericType<T>.IsNullable)
104             {
105                 emitter.NewObject(typeof(T), NumericType<T>.UnderlyingType);
106             }
107             if (typeof(T) == typeof(Integer))
108             {
109                 emitter.NewObject(typeof(Integer), typeof(ulong));
110             }
111             emitter.NewObject(typeof(Integer<T>), typeof(T));
112             emitter.Return();
113         });
114     }
115 }
116 }
117 }
118 }
119 }
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         }
38     }
39 }

```



```

37         if (n < _factorials.Length)
38         {
39             return _factorials[(int)n];
40         }
41         return n * Factorial(n - 1);
42     }
43
44     public static double Catalan(double n)
45     {
46         if (n <= 1)
47         {
48             return 1;
49         }
50         if (n < _catalans.Length)
51         {
52             return _catalans[(int)n];
53         }
54         return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
55     }
56
57     public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
58
59     public static T Abs<T>(T x) => Math<T>.Abs(x);
60
61     public static T Negate<T>(T x) => Math<T>.Negate(x);
62 }
63 }

```

./Platform.Numbers/MathExtensions.cs

```

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

```

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

```

1  using System;
2  using System.Reflection;
3  using Platform.Exceptions;
4  using Platform.Reflection;
5
6  // ReSharper disable StaticFieldInGenericType
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Numbers
10 {
11     public static class Math<T>
12     {
13         public static readonly Func<T, T> Abs;
14         public static readonly Func<T, T> Negate;
15
16         static Math()
17         {
18             Abs = CompileAbsDelegate();
19             Negate = CompileNegateDelegate();
20         }
21
22         private static Func<T, T> CompileAbsDelegate()
23         {
24             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
25             {
26                 Ensure.Always.IsNumeric<T>();
27                 emitter.LoadArgument(0);
28                 if (NumericType<T>.IsSigned)
29                 {
30                     emitter.Call(typeof(System.Math).GetTypeInfo().GetMethod("Abs",
31                                     ↳ Types<T>.Array));
32                 }
33                 emitter.Return();
34             });
35
36             private static Func<T, T> CompileNegateDelegate()
37             {

```

```

38         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
39         {
40             Ensure.Always.IsSigned<T>();
41             emitter.LoadArgument(0);
42             emitter.Negate();
43             emitter.Return();
44         });
45     }
46 }
47 }

```

./Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class ArithmeticExtensionsTests
6      {
7          [Fact]
8          public static void IncrementTest()
9          {
10             var number = 0UL;
11             var returnValue = number.Increment();
12             Assert.Equal(1UL, returnValue);
13             Assert.Equal(1UL, number);
14         }
15
16         [Fact]
17         public static void DecrementTest()
18         {
19             var number = 1UL;
20             var returnValue = number.Decrement();
21             Assert.Equal(0UL, returnValue);
22             Assert.Equal(0UL, number);
23         }
24     }
25 }

```

./Platform.Numbers.Tests/ArithmeticTests.cs

```

1  using System;
2  using Xunit;
3
4  namespace Platform.Numbers.Tests
5  {
6      public static class ArithmeticTests
7      {
8          [Fact]
9          public static void CompiledOperationsTest()
10         {
11             Assert.True(Arithmetic<short>.Add(1, 2) == 3);
12             Assert.True(Arithmetic<short>.Subtract(2, 1) == 1);
13             Assert.True(Arithmetic<byte>.Increment(1) == 2);
14             Assert.True(Arithmetic<ulong>.Decrement(2) == 1);
15             Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
16         }
17     }
18 }

```

./Platform.Numbers.Tests/BitTests.cs

```

1  using System;
2  using Xunit;
3
4  namespace Platform.Numbers.Tests
5  {
6      public static class BitTests
7      {
8          [Theory]
9          [InlineData(00, -1)] // 0000 0000 (none, -1)
10         [InlineData(01, 00)] // 0000 0001 (first, 0)
11         [InlineData(08, 03)] // 0000 1000 (forth, 3)
12         [InlineData(88, 03)] // 0101 1000 (forth, 3)
13         public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
14         {
15             Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
16         }
17
18         [Fact]
19         public static void ByteBitwiseOperationsTest()

```

```

20 {
21     Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
22     Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
23     Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
24     Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
25     Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
26 }
27
28 [Fact]
29 public static void UInt16BitwiseOperationsTest()
30 {
31     Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
32     Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
33     Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
34     Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
35     Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
36 }
37
38 [Fact]
39 public static void UInt32BitwiseOperationsTest()
40 {
41     Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
42     Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
43     Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
44     Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
45     Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
46 }
47
48 [Fact]
49 public static void UInt64BitwiseOperationsTest()
50 {
51     Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
52     Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
53     Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
54     Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
55     Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
56 }
57
58 [Fact]
59 public static void PartialReadWriteTest()
60 {
61     {
62         uint firstValue = 1;
63         uint secondValue = 1543;
64
65         // Pack (join) two values at the same time
66         uint value = secondValue << 1 | firstValue;
67
68         uint unpackagedFirstValue = value & 1;
69         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
70
71         Assert.True(firstValue == unpackagedFirstValue);
72         Assert.True(secondValue == unpackagedSecondValue);
73
74         // Using universal functions:
75
76         Assert.True(PartialRead(value, 0, 1) == firstValue);
77         Assert.True(PartialRead(value, 1, -1) == secondValue);
78
79         firstValue = 0;
80         secondValue = 6892;
81
82         value = PartialWrite(value, firstValue, 0, 1);
83         value = PartialWrite(value, secondValue, 1, -1);
84
85         Assert.True(PartialRead(value, 0, 1) == firstValue);
86         Assert.True(PartialRead(value, 1, -1) == secondValue);
87     }
88
89     {
90         uint firstValue = 1;
91         uint secondValue = 1543;
92
93         // Pack (join) two values at the same time
94         uint value = secondValue << 1 | firstValue;
95
96         uint unpackagedFirstValue = value & 1;
97         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
98
99         Assert.True(firstValue == unpackagedFirstValue);

```

```

100     Assert.True(secondValue == unpackagedSecondValue);
101
102     // Using universal functions:
103
104     Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
105     Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
106
107     firstValue = 0;
108     secondValue = 6892;
109
110     value = Bit.PartialWrite(value, firstValue, 0, 1);
111     value = Bit.PartialWrite(value, secondValue, 1, -1);
112
113     Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
114     Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
115 }
116
117 {
118     uint firstValue = 1;
119     uint secondValue = 1543;
120
121     // Pack (join) two values at the same time
122     uint value = secondValue << 1 | firstValue;
123
124     uint unpackagedFirstValue = value & 1;
125     uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
126
127     Assert.True(firstValue == unpackagedFirstValue);
128     Assert.True(secondValue == unpackagedSecondValue);
129
130     // Using universal functions:
131
132     var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
133     var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
134     var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
135     var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
136
137     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
138     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
139
140     firstValue = 0;
141     secondValue = 6892;
142
143     value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
144     value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
145
146     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
147     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
148 }
149 }
150
151 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
152 private static uint PartialWrite(uint target, uint source, int shift, int limit)
153 {
154     if (shift < 0)
155     {
156         shift = 32 + shift;
157     }
158     if (limit < 0)
159     {
160         limit = 32 + limit;
161     }
162     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
163     var targetMask = ~(sourceMask << shift);
164     return target & targetMask | (source & sourceMask) << shift;
165 }
166
167 private static uint PartialRead(uint target, int shift, int limit)
168 {
169     if (shift < 0)
170     {
171         shift = 32 + shift;
172     }
173     if (limit < 0)
174     {
175         limit = 32 + limit;
176     }
177     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
178     var targetMask = sourceMask << shift;
179     return (target & targetMask) >> shift;

```

```

180     }
181
182     private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
183     {
184         if (shift < 0)
185         {
186             shift = 32 + shift;
187         }
188         if (limit < 0)
189         {
190             limit = 32 + limit;
191         }
192         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
193         var targetMask = ~(sourceMask << shift);
194         return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
195     }
196
197     private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
198     {
199         if (shift < 0)
200         {
201             shift = 32 + shift;
202         }
203         if (limit < 0)
204         {
205             limit = 32 + limit;
206         }
207         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
208         var targetMask = sourceMask << shift;
209         return new Tuple<uint, int>(targetMask, shift);
210     }
211
212     private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
213
214     private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
215
216     private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
217
218     private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
219
220     [Fact]
221     public static void BugWithLoadingConstantOf8Test()
222     {
223         Bit<byte>.PartialWrite(0, 1, 5, -5);
224     }
225 }
226 }

```

./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 }
```

```
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, 10
- ./Platform.Numbers.Tests/ArithmeticTests.cs, 10
- ./Platform.Numbers.Tests/BitTests.cs, 10
- ./Platform.Numbers.Tests/MathExtensionsTests.cs, 13
- ./Platform.Numbers.Tests/MathTests.cs, 14
- ./Platform.Numbers.Tests/SystemTests.cs, 14
- ./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, 6
- ./Platform.Numbers/Integer[T].cs, 6
- ./Platform.Numbers/Math.cs, 8
- ./Platform.Numbers/MathExtensions.cs, 9
- ./Platform.Numbers/Math[T].cs, 9