

LinksPlatform's Platform.Numbers Class Library

1.1 ./csharp/Platform.Numbers/Arithmetic.cs

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     public static class Arithmetic
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
23
24        [MethodImpl(MethodImplOptions.AggressiveInlining)]
25        public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);
26
27        [MethodImpl(MethodImplOptions.AggressiveInlining)]
28        public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
29
30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        public static T Decrement<T>(ref T x) => x = Arithmetic<T>.Decrement(x);
32    }
33 }
```

1.2 ./csharp/Platform.Numbers/ArithmeticExtensions.cs

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     public static class ArithmeticExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static T Decrement<T>(this ref T x) where T : struct => x =
11            ↪ Arithmetic<T>.Decrement(x);
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public static T Increment<T>(this ref T x) where T : struct => x =
15            ↪ Arithmetic<T>.Increment(x);
16    }
17 }
```

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

```
1 using System;
2 using System.Reflection.Emit;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Reflection;
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 Arithmetic<T>
13     {
14         public static readonly Func<T, T, T> Add = CompileAddDelegate();
15         public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
16         public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
17         public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
18         public static readonly Func<T, T> Increment = CompileIncrementDelegate();
19         public static readonly Func<T, T> Decrement = CompileDecrementDelegate();
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         private static Func<T, T, T> CompileAddDelegate()
23         {
```

```

24     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
25     {
26         Ensure.Always.IsNumeric<T>();
27         emitter.LoadArguments(0, 1);
28         emitter.Add();
29         emitter.Return();
30     });
31 }
32
33 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34 private static Func<T, T, T> CompileSubtractDelegate()
35 {
36     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
37     {
38         Ensure.Always.IsNumeric<T>();
39         emitter.LoadArguments(0, 1);
40         emitter.Subtract();
41         emitter.Return();
42     });
43 }
44
45 [MethodImpl(MethodImplOptions.AggressiveInlining)]
46 private static Func<T, T, T> CompileMultiplyDelegate()
47 {
48     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
49     {
50         Ensure.Always.IsNumeric<T>();
51         emitter.LoadArguments(0, 1);
52         emitter.Emit(OpCodes.Mul);
53         emitter.Return();
54     });
55 }
56
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 private static Func<T, T, T> CompileDivideDelegate()
59 {
60     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
61     {
62         Ensure.Always.IsNumeric<T>();
63         emitter.LoadArguments(0, 1);
64         if(NumericType<T>.IsSigned)
65         {
66             emitter.Emit(OpCodes.Div);
67         }
68         else
69         {
70             emitter.Emit(OpCodes.Div_Un);
71         }
72         emitter.Return();
73     });
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 private static Func<T, T> CompileIncrementDelegate()
78 {
79     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
80     {
81         Ensure.Always.IsNumeric<T>();
82         emitter.LoadArgument(0);
83         emitter.Increment<T>();
84         emitter.Return();
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private static Func<T, T> CompileDecrementDelegate()
90 {
91     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
92     {
93         Ensure.Always.IsNumeric<T>();
94         emitter.LoadArgument(0);
95         emitter.Decrement<T>();
96         emitter.Return();
97     });
98 }
99 }
100 }

```

1.4 ./csharp/Platform.Numbers/Bit.cs

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     public static class Bit
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static long Count(long x)
11        {
12            long n = 0;
13            while (x != 0)
14            {
15                n++;
16                x &= x - 1;
17            }
18            return n;
19        }
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        public static int GetLowestPosition(ulong value)
23        {
24            if (value == 0)
25            {
26                return -1;
27            }
28            var position = 0;
29            while ((value & 1UL) == 0)
30            {
31                value >>= 1;
32                ++position;
33            }
34            return position;
35        }
36
37        [MethodImpl(MethodImplOptions.AggressiveInlining)]
38        public static T Not<T>(T x) => Bit<T>.Not(x);
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
42
43        [MethodImpl(MethodImplOptions.AggressiveInlining)]
44        public static T And<T>(T x, T y) => Bit<T>.And(x, y);
45
46        [MethodImpl(MethodImplOptions.AggressiveInlining)]
47        public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
48
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
51
52        [MethodImpl(MethodImplOptions.AggressiveInlining)]
53        public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
54            ↪ Bit<T>.PartialWrite(target, source, shift, limit);
55
56        [MethodImpl(MethodImplOptions.AggressiveInlining)]
57        public static T PartialRead<T>(T target, int shift, int limit) =>
58            ↪ Bit<T>.PartialRead(target, shift, limit);
59    }
60 }
```

1.5 ./csharp/Platform.Numbers/BitExtensions.cs

```
1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     public static class BitwiseExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
14            ↪ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
15
16        [MethodImpl(MethodImplOptions.AggressiveInlining)]
17    }
```

```

16         public static T PartialRead<T>(this T target, int shift, int limit) =>
17             ↪ Bit<T>.PartialRead(target, shift, limit);
18     }
19 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
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 = CompileNotDelegate();
14         public static readonly Func<T, T, T> Or = CompileOrDelegate();
15         public static readonly Func<T, T, T> And = CompileAndDelegate();
16         public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
17         public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
18         public static readonly Func<T, T, int, int, T> PartialWrite =
19             ↪ CompilePartialWriteDelegate();
20         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         private static Func<T, T> CompileNotDelegate()
24         {
25             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
26             {
27                 Ensure.Always.IsNumeric<T>();
28                 emitter.LoadArguments(0);
29                 emitter.Not();
30                 emitter.Return();
31             });
32         }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         private static Func<T, T, T> CompileOrDelegate()
36         {
37             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
38             {
39                 Ensure.Always.IsNumeric<T>();
40                 emitter.LoadArguments(0, 1);
41                 emitter.Or();
42                 emitter.Return();
43             });
44         }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         private static Func<T, T, T> CompileAndDelegate()
48         {
49             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
50             {
51                 Ensure.Always.IsNumeric<T>();
52                 emitter.LoadArguments(0, 1);
53                 emitter.And();
54                 emitter.Return();
55             });
56         }
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         private static Func<T, int, T> CompileShiftLeftDelegate()
60         {
61             return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
62             {
63                 Ensure.Always.IsNumeric<T>();
64                 emitter.LoadArguments(0, 1);
65                 emitter.ShiftLeft();
66                 emitter.Return();
67             });
68         }
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         private static Func<T, int, T> CompileShiftRightDelegate()
72         {
73             return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>

```

```

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

```

```

151     ushort limitArgument = 2;
152     var checkLimit = emitter.DefineLabel();
153     var calculateSourceMask = emitter.DefineLabel();
154     // Check shift
155     emitter.LoadArgument(shiftArgument);
156     emitter.LoadConstant(0);
157     emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
158     // Fix shift
159     emitter.LoadConstant(bitsNumber);
160     emitter.LoadArgument(shiftArgument);
161     emitter.Add();
162     emitter.StoreArgument(shiftArgument);
163     emitter.MarkLabel(checkLimit);
164     // Check limit
165     emitter.LoadArgument(limitArgument);
166     emitter.LoadConstant(0);
167     emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
168     // Fix limit
169     emitter.LoadConstant(bitsNumber);
170     emitter.LoadArgument(limitArgument);
171     emitter.Add();
172     emitter.StoreArgument(limitArgument);
173     emitter.MarkLabel(calculateSourceMask);
174     var sourceMask = emitter.DeclareLocal<T>();
175     var targetMask = emitter.DeclareLocal<T>();
176     emitter.LoadConstant(typeof(T), numberFilledWithOnes);
177     emitter.LoadArgument(limitArgument); // limit
178     emitter.ShiftLeft();
179     emitter.Not();
180     emitter.LoadConstant(typeof(T), numberFilledWithOnes);
181     emitter.And();
182     emitter.StoreLocal(sourceMask);
183     emitter.LoadLocal(sourceMask);
184     emitter.LoadArgument(shiftArgument);
185     emitter.ShiftLeft();
186     emitter.StoreLocal(targetMask);
187     emitter.LoadArgument(0); // target
188     emitter.LoadLocal(targetMask);
189     emitter.And();
190     emitter.LoadArgument(shiftArgument);
191     emitter.ShiftRight();
192     emitter.Return();
193 });
194 }
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 private static Tuple<int, T> GetConstants()
198 {
199     var type = typeof(T);
200     if (type == typeof(ulong))
201     {
202         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
203     }
204     if (type == typeof(uint))
205     {
206         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
207     }
208     if (type == typeof(ushort))
209     {
210         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
211     }
212     if (type == typeof(byte))
213     {
214         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
215     }
216     throw new NotSupportedException();
217 }
218 }
219 }

```

1.7 ./csharp/Platform.Numbers/Math.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Runtime.CompilerServices;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7

```

```

8 namespace Platform.Numbers
9 {
10     /// <remarks>
11     /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
12     /// </remarks>
13     public static class Math
14     {
15         /// <remarks>
16         /// <para>Source: https://oeis.org/A000142/list </para>
17         /// <para>Источник: https://oeis.org/A000142/list </para>
18         /// </remarks>
19         private static readonly long[] _factorials =
20         {
21             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
22             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
23             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000,
24         };
25
26         /// <remarks>
27         /// <para>Source: https://oeis.org/A000108/list </para>
28         /// <para>Источник: https://oeis.org/A000108/list </para>
29         /// </remarks>
30         private static readonly long[] _catalans =
31         {
32             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
33             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
34             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324,
35             ↪ 4861946401452,
36             18367353072152, 69533550916004, 263747951750360, 1002242216651368,
37             ↪ 3814986502092304
38         };
39
40         /// <summary>
41         /// <para>Generate the factorial of the value "n".</para>
42         /// <para>Генерация факториала из значения переменной "n".</para>
43         /// </summary>
44         /// <param name="n"><para>Factorial generation value.</para><para>Значение генерации
45         ↪ факториала.</para></param>
46         /// <returns><para>Result of factorial calculation.</para><para>Результат подсчета
47         ↪ факториала.</para></returns>
48         public static long Factorial(long n)
49         {
50             if (n <= 1)
51             {
52                 return 1;
53             }
54
55             if (n < _factorials.Length)
56             {
57                 return _factorials[n];
58             }
59
60             return n * Factorial(n - 1);
61         }
62
63         public static long FactorialEratosthenesSieve(long n)
64         {
65             if (n < 0)
66                 return 0;
67             if (n == 0)
68                 return 1;
69             if (n < _factorials.Length)
70                 return _factorials[n];
71
72             bool[] u = new bool[n + 1]; // маркеры для решета Эратосфена
73             List<Tuple<long, long>> p = new List<Tuple<long, long>>(); // множители и их
74             ↪ показатели степеней
75             for (int i = 2; i <= n; ++i)
76                 if (!u[i]) // если i - очередное простое число
77                 {
78                     // считаем показатель степени в разложении
79                     long k = n / i;
80                     long c = 0;
81                     while (k > 0)
82                     {
83                         c += k;
84                         k /= i;
85                     }
86                     u[i] = true;
87                     p.Add(new Tuple<long, long>(i, c));
88                 }
89
90             return p.Count == 0 ? 1 : p[p.Count - 1].Item2;
91         }
92     }
93 }

```

```

82     }
83     // запоминаем множитель и его показатель степени
84     p.Add(new Tuple<long, long>(i, c));
85     // просеиваем составные числа через решето
86     int j = 2;
87     while (i * j <= n)
88     {
89         u[i * j] = true;
90         ++j;
91     }
92 }
93 // вычисляем факториал
94 int r = 1;
95 for (int i = p.Count() - 1; i >= 0; --i)
96     r *= (int)System.Math.Pow(p[i].Item1, p[i].Item2);
97 return r;
98 }
99
100
101 /// <summary>
102 /// <para>Generating the Catalan Number of the value "n".</para>
103 /// <para>Генерация числа Каталана из значения переменной "n".</para>
104 /// </summary>
105 /// <param name="n"><para>Catalan Number generation value.</para><para>Значение
106   ↳ генерации Числа Каталана.</para></param>
107 /// <returns><para>Result of Catalan Number calculation.</para><para>Результат подсчета
108   ↳ Числа Каталана.</para></returns>
109 public static long Catalan(int n)
110 {
111     if (n <= 1)
112     {
113         return 1;
114     }
115     if (n < _catalans.Length)
116     {
117         return _catalans[(int)n];
118     }
119     return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
120 }
121
122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static T Abs<T>(T x) => Math<T>.Abs(x);
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static T Negate<T>(T x) => Math<T>.Negate(x);
130 }
131 }

```

1.8 ./csharp/Platform.Numbers/MathExtensions.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     public static class MathExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
14     }
15 }

```

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

```

1 using System;
2 using System.Runtime.CompilerServices;
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 = CompileAbsDelegate();
14         public static readonly Func<T, T> Negate = CompileNegateDelegate();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         private static Func<T, T> CompileAbsDelegate()
18         {
19             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
20             {
21                 Ensure.Always.IsNumeric<T>();
22                 emitter.LoadArgument(0);
23                 if (NumericType<T>.IsSigned)
24                 {
25                     emitter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
26                 }
27                 emitter.Return();
28             });
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         private static Func<T, T> CompileNegateDelegate()
33         {
34             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
35             {
36                 emitter.LoadArgument(0);
37                 emitter.Negate();
38                 emitter.Return();
39             });
40         }
41     }
42 }

```

1.10 ./csharp/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 }

```

1.11 ./csharp/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.Equal(3, Arithmetic.Add(1, 2));
12            Assert.Equal(1, Arithmetic.Subtract(2, 1));
13            Assert.Equal(8, Arithmetic.Multiply(2, 4));
14            Assert.Equal(4, Arithmetic.Divide(8, 2));
15            Assert.Equal(2, Arithmetic.Increment(1));
16            Assert.Equal(1UL, Arithmetic.Decrement(2UL));
17        }
18    }
19 }

```

```

17         Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
18     }
19 }
20 }

```

1.12 ./csharp/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         }
75     }
76 }

```

```

74 // Using universal functions:
75 Assert.True(PartialRead(value, 0, 1) == firstValue);
76 Assert.True(PartialRead(value, 1, -1) == secondValue);
77
78 firstValue = 0;
79 secondValue = 6892;
80
81 value = PartialWrite(value, firstValue, 0, 1);
82 value = PartialWrite(value, secondValue, 1, -1);
83
84 Assert.True(PartialRead(value, 0, 1) == firstValue);
85 Assert.True(PartialRead(value, 1, -1) == secondValue);
86 }
87
88 {
89     uint firstValue = 1;
90     uint secondValue = 1543;
91
92     // Pack (join) two values at the same time
93     uint value = secondValue << 1 | firstValue;
94
95     uint unpackagedFirstValue = value & 1;
96     uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
97
98     Assert.True(firstValue == unpackagedFirstValue);
99     Assert.True(secondValue == unpackagedSecondValue);
100
101     // Using universal functions:
102     Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
103     Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
104
105     firstValue = 0;
106     secondValue = 6892;
107
108     value = Bit.PartialWrite(value, firstValue, 0, 1);
109     value = Bit.PartialWrite(value, secondValue, 1, -1);
110
111     Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
112     Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
113 }
114
115 {
116     uint firstValue = 1;
117     uint secondValue = 1543;
118
119     // Pack (join) two values at the same time
120     uint value = secondValue << 1 | firstValue;
121
122     uint unpackagedFirstValue = value & 1;
123     uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
124
125     Assert.True(firstValue == unpackagedFirstValue);
126     Assert.True(secondValue == unpackagedSecondValue);
127
128     // Using universal functions:
129     var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
130     var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
131     var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
132     var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
133
134     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
135     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
136
137     firstValue = 0;
138     secondValue = 6892;
139
140     value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
141     value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
142
143     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
144     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
145 }
146
147 }
148
149 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
150 private static uint PartialWrite(uint target, uint source, int shift, int limit)
151 {
152     if (shift < 0)
153     {
154         shift = 32 + shift;
155     }
156 }

```

```

155         if (limit < 0)
156         {
157             limit = 32 + limit;
158         }
159         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
160         var targetMask = ~(sourceMask << shift);
161         return target & targetMask | (source & sourceMask) << shift;
162     }
163
164     private static uint PartialRead(uint target, int shift, int limit)
165     {
166         if (shift < 0)
167         {
168             shift = 32 + shift;
169         }
170         if (limit < 0)
171         {
172             limit = 32 + limit;
173         }
174         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
175         var targetMask = sourceMask << shift;
176         return (target & targetMask) >> shift;
177     }
178
179     private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
180     {
181         if (shift < 0)
182         {
183             shift = 32 + shift;
184         }
185         if (limit < 0)
186         {
187             limit = 32 + limit;
188         }
189         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
190         var targetMask = ~(sourceMask << shift);
191         return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
192     }
193
194     private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
195     {
196         if (shift < 0)
197         {
198             shift = 32 + shift;
199         }
200         if (limit < 0)
201         {
202             limit = 32 + limit;
203         }
204         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
205         var targetMask = sourceMask << shift;
206         return new Tuple<uint, int>(targetMask, shift);
207     }
208
209     private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
210
211     private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
212
213     private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
214
215     private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
216
217     [Fact]
218     public static void BugWithLoadingConstantOf8Test()
219     {
220         Bit<byte>.PartialWrite(0, 1, 5, -5);
221     }
222 }
223 }

```

1.13 ./csharp/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         [Fact]
26         public static void UnsignedNegateTest()
27         {
28             var number = 2UL;
29             var returnValue = number.Negate();
30             Assert.Equal(18446744073709551614, returnValue);
31             Assert.Equal(18446744073709551614, number);
32         }
33     }
34 }

```

1.14 ./csharp/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 }

```

1.15 ./csharp/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             value |= 1;
14
15             Assert.True(value == 1);
16
17             // Set zero to first bit
18             value &= 0xFFFFFFFF;
19
20             // Get first bit
21             uint read = value & 1;
22
23             Assert.True(read == 0);
24
25             uint firstValue = 1;
26             uint secondValue = 1543;
27
28             // Pack (join) two values at the same time
29             value = (secondValue << 1) | firstValue;
30
31             uint unpackagedFirstValue = value & 1;
32             uint unpackagedSecondValue = (value & 0xFFFFFFFF) >> 1;

```

```

33     Assert.True(firstValue == unpackagedFirstValue);
34     Assert.True(secondValue == unpackagedSecondValue);
35
36     // Using universal functions:
37
38     Assert.True(PartialRead(value, 0, 1) == firstValue);
39     Assert.True(PartialRead(value, 1, -1) == secondValue);
40
41     firstValue = 0;
42     secondValue = 6892;
43
44     value = PartialWrite(value, firstValue, 0, 1);
45     value = PartialWrite(value, secondValue, 1, -1);
46
47     Assert.True(PartialRead(value, 0, 1) == firstValue);
48     Assert.True(PartialRead(value, 1, -1) == secondValue);
49 }
50
51 private static uint PartialWrite(uint target, uint source, int shift, int limit)
52 {
53     if (shift < 0)
54     {
55         shift = 32 + shift;
56     }
57     if (limit < 0)
58     {
59         limit = 32 + limit;
60     }
61     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
62     var targetMask = ~(sourceMask << shift);
63     return (target & targetMask) | ((source & sourceMask) << shift);
64 }
65
66 private static uint PartialRead(uint target, int shift, int limit)
67 {
68     if (shift < 0)
69     {
70         shift = 32 + shift;
71     }
72     if (limit < 0)
73     {
74         limit = 32 + limit;
75     }
76     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
77     var targetMask = sourceMask << shift;
78     return (target & targetMask) >> shift;
79 }
80 }
81 }
82 }

```

Index

- ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs, 9
- ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs, 9
- ./csharp/Platform.Numbers.Tests/BitTests.cs, 10
- ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs, 12
- ./csharp/Platform.Numbers.Tests/MathTests.cs, 13
- ./csharp/Platform.Numbers.Tests/SystemTests.cs, 13
- ./csharp/Platform.Numbers/Arithmetic.cs, 1
- ./csharp/Platform.Numbers/ArithmeticExtensions.cs, 1
- ./csharp/Platform.Numbers/Arithmetic[T].cs, 1
- ./csharp/Platform.Numbers/Bit.cs, 2
- ./csharp/Platform.Numbers/BitExtensions.cs, 3
- ./csharp/Platform.Numbers/Bit[T].cs, 4
- ./csharp/Platform.Numbers/Math.cs, 6
- ./csharp/Platform.Numbers/MathExtensions.cs, 8
- ./csharp/Platform.Numbers/Math[T].cs, 8