

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      /// <summary>
8      /// <para>Represents a set of arithmetic methods.</para>
9      /// <para>Представляет набор арифметических методов.</para>
10     /// </summary>
11     public static class Arithmetic
12     {
13         /// <summary>
14         /// <para>Performing adding the x and y arguments.</para>
15         /// <para>Выполняет сложение аргументов x и y.</para>
16         /// </summary>
17         /// <typeparam name="T">
18         /// <para>The numbers type.</para>
19         /// <para>Тип чисел.</para>
20         /// </typeparam>
21         /// <param name="x">
22         /// <para>The first term.</para>
23         /// <para>Первое слагаемое.</para>
24         /// </param>
25         /// <param name="y">
26         /// <para>The second term.</para>
27         /// <para>Второе слагаемое.</para>
28         /// </param>
29         /// <returns>
30         /// <para>Sum of x and y.</para>
31         /// <para>Сумма x и y.</para>
32         /// </returns>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
35
36         /// <summary>
37         /// <para>Performs subtracting y from x.</para>
38         /// <para>Выполняет вычитание y из x.</para>
39         /// </summary>
40         /// <typeparam name="T">
41         /// <para>The numbers type.</para>
42         /// <para>Тип чисел.</para>
43         /// </typeparam>
44         /// <param name="x">
45         /// <para>Minuend.</para>
46         /// <para>Уменьшаемое.</para>
47         /// </param>
48         /// <param name="y">
49         /// <para>Subtrahend.</para>
50         /// <para>Вычитаемое.</para>
51         /// </param>
52         /// <returns>
53         /// <para>Difference between x and y.</para>
54         /// <para>Разность между x и y.</para>
55         /// </returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
58
59         /// <summary>
60         /// <para>Performs multiplication x by y.</para>
61         /// <para>Выполняет умножение x на y.</para>
62         /// </summary>
63         /// <typeparam name="T">
64         /// <para>The numbers type.</para>
65         /// <para>Тип чисел.</para>
66         /// </typeparam>
67         /// <param name="x">
68         /// <para>First multiplier.</para>
69         /// <para>Первый множитель.</para>
70         /// </param>
71         /// <param name="y">
72         /// <para>Second multiplier.</para>
73         /// <para>Второй множитель.</para>
74         /// </param>
75         /// <returns>
76         /// <para>Product of x and y.</para>
```

```

77     /// <para>Произведение x и y.</para>
78     /// </returns>
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);
81
82     /// <summary>
83     /// <para>Performs dividing x by y.</para>
84     /// <para>Выполняет деление x на y.</para>
85     /// </summary>
86     /// <typeparam name="T">
87     /// <para>The numbers type.</para>
88     /// <para>Тип чисел.</para>
89     /// </typeparam>
90     /// <param name="x">
91     /// <para>Dividend.</para>
92     /// <para>Делимое.</para>
93     /// </param>
94     /// <param name="y">
95     /// <para>Divider.</para>
96     /// <para>Делитель.</para>
97     /// </param>
98     /// <returns>
99     /// <para>Quoitent of x and y.</para>
100    /// <para>Частное x и y.</para>
101    /// </returns>
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);
104
105    /// <summary>
106    /// <para>Increasing the number x by one.</para>
107    /// <para>Увеличивает число x на единицу.</para>
108    /// </summary>
109    /// <typeparam name="T">
110    /// <para>The number type.</para>
111    /// <para>Тип числа.</para>
112    /// </typeparam>
113    /// <param name="x">
114    /// <para>The number to increase.</para>
115    /// <para>Число для увеличения.</para>
116    /// </param>
117    /// <returns>
118    /// <para>Increase by one number x.</para>
119    /// <para>Увеличенное на единицу число x.</para>
120    /// </returns>
121    [MethodImpl(MethodImplOptions.AggressiveInlining)]
122    public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
123
124    /// <summary>
125    /// <para>Increases the value of argument x by one.</para>
126    /// <para>Увеличивает значение аргумента x на единицу.</para>
127    /// </summary>
128    /// <typeparam name="T">
129    /// <para>The number type.</para>
130    /// <para>Тип числа.</para>
131    /// </typeparam>
132    /// <param name="x">
133    /// <para>The argument to increase.</para>
134    /// <para>Аргумент для увеличения.</para>
135    /// </param>
136    /// <returns>
137    /// <para>Increased argument x value.</para>
138    /// <para>Увеличенное значение аргумента x.</para>
139    /// </returns>
140    [MethodImpl(MethodImplOptions.AggressiveInlining)]
141    public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);
142
143    /// <summary>
144    /// <para>Decrease number x by one.</para>
145    /// <para>Уменьшает число x на единицу.</para>
146    /// </summary>
147    /// <typeparam name="T">
148    /// <para>The number type.</para>
149    /// <para>Тип числа.</para>
150    /// </typeparam>
151    /// <param name="x">
152    /// <para>The number to reduce.</para>
153    /// <para>Число для уменьшения.</para>
154    /// </param>

```

```

155     /// <returns>
156     /// <para>Decreased by one number x.</para>
157     /// <para>Уменьшенное на единицу число x.</para>
158     /// </returns>
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
161
162     /// <summary>
163     /// <para>Decreases the value of the argument x by one.</para>
164     /// <para>Уменьшает значение аргумента x на единицу.</para>
165     /// </summary>
166     /// <typeparam name="T">
167     /// <para>The number type.</para>
168     /// <para>Тип числа.</para>
169     /// </typeparam>
170     /// <param name="x">
171     /// <para>The argument to reduce.</para>
172     /// <para>Аргумент для уменьшения.</para>
173     /// </param>
174     /// <returns>
175     /// <para>Decreased argument x value.</para>
176     /// <para>Уменьшенное значение аргумента x.</para>
177     /// </returns>
178     [MethodImpl(MethodImplOptions.AggressiveInlining)]
179     public static T Decrement<T>(ref T x) => x = Arithmetic<T>.Decrement(x);
180 }
181 }

```

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 }

```

```

32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 private static Func<T, T, T> CompileSubtractDelegate()
34 {
35     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
36     {
37         Ensure.Always.IsNumeric<T>();
38         emitter.LoadArguments(0, 1);
39         emitter.Subtract();
40         emitter.Return();
41     });
42 }
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 private static Func<T, T, T> CompileMultiplyDelegate()
46 {
47     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
48     {
49         Ensure.Always.IsNumeric<T>();
50         emitter.LoadArguments(0, 1);
51         emitter.Emit(OpCodes.Mul);
52         emitter.Return();
53     });
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 private static Func<T, T, T> CompileDivideDelegate()
58 {
59     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
60     {
61         Ensure.Always.IsNumeric<T>();
62         emitter.LoadArguments(0, 1);
63         if(NumericType<T>.IsSigned)
64         {
65             emitter.Emit(OpCodes.Div);
66         }
67         else
68         {
69             emitter.Emit(OpCodes.Div_Un);
70         }
71         emitter.Return();
72     });
73 }
74
75 [MethodImpl(MethodImplOptions.AggressiveInlining)]
76 private static Func<T, T> CompileIncrementDelegate()
77 {
78     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
79     {
80         Ensure.Always.IsNumeric<T>();
81         emitter.LoadArgument(0);
82         emitter.Increment<T>();
83         emitter.Return();
84     });
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 private static Func<T, T> CompileDecrementDelegate()
89 {
90     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
91     {
92         Ensure.Always.IsNumeric<T>();
93         emitter.LoadArgument(0);
94         emitter.Decrement<T>();
95         emitter.Return();
96     });
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
6 namespace Platform.Numbers
7 {
8

```

```

9  /// <summary>
10 /// <para>A set of operations on the set bits of a number.</para>
11 /// <para>Набор операций над установленными битами числа.</para>
12 /// </summary>
13
14 public static class Bit
15 {
16     /// <summary>
17     /// <para>Counts the number of bits set in a number.</para>
18     /// <para>Подсчитывает количество установленных бит в числе.</para>
19     /// </summary>
20     /// <param name="x">
21     /// <para>Bitwise number.</para>
22     /// <para>Число в битовом представлении.</para>
23     /// </param>
24     /// <returns>
25     /// <para>Number of bits set in a number.</para>
26     /// <para>Количество установленных бит в числе.</para>
27     /// </returns>
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public static long Count(long x)
30     {
31         long n = 0;
32         while (x != 0)
33         {
34             n++;
35             x &= x - 1;
36         }
37         return n;
38     }
39
40     /// <summary>
41     /// <para>Searches for the first bit set in a number.</para>
42     /// <para>Ищет первый установленный бит в числе.</para>
43     /// </summary>
44     /// <param name="value">
45     /// <para>Bitwise number.</para>
46     /// <para>Число в битовом представлении.</para>
47     /// </param>
48     /// <returns>
49     /// <para>First bit set.</para>
50     /// <para>Первый установленный бит.</para>
51     /// </returns>
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public static int GetLowestPosition(ulong value)
54     {
55         if (value == 0)
56         {
57             return -1;
58         }
59         var position = 0;
60         while ((value & 1UL) == 0)
61         {
62             value >>= 1;
63             ++position;
64         }
65         return position;
66     }
67
68     /// <summary>
69     /// <para>Performing bitwise inversion of a number.</para>
70     /// <para>Выполняет побитовую инверсию числа.</para>
71     /// </summary>
72     /// <typeparam name="T">
73     /// <para>The number type.</para>
74     /// <para>Тип числа.</para>
75     /// </typeparam>
76     /// <param name="x">
77     /// <para>Number to invert.</para>
78     /// <para>Число для инверсии.</para>
79     /// </param>
80     /// <returns>
81     /// <para>Inverse value of the number.</para>
82     /// <para>Обратное значение числа.</para>
83     /// </returns>
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     public static T Not<T>(T x) => Bit<T>.Not(x);
86
87     /// <summary>

```

```

88     /// <para>Performing bitwise numbers addition.</para>
89     /// <para>Выполняет побитовое сложение чисел.</para>
90     /// </summary>
91     /// <typeparam name="T">
92     /// <para>The numbers type.</para>
93     /// <para>Тип чисел.</para>
94     /// </typeparam>
95     /// <param name="x">
96     /// <para>First term.</para>
97     /// <para>Первое слагаемое.</para>
98     /// </param>
99     /// <param name="y">
100    /// <para>Second term.</para>
101    /// <para>Второе слагаемое.</para>
102    /// </param>
103    /// <returns>
104    /// <para>The logical sum of numbers</para>
105    /// <para>Логическая сумма чисел.</para>
106    /// </returns>
107    [MethodImpl(MethodImplOptions.AggressiveInlining)]
108    public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
109
110    /// <summary>
111    /// <para>Performs bitwise numbers multiplication.</para>
112    /// <para>Выполняет побитовое умножение чисел.</para>
113    /// <typeparam name="T">
114    /// <para>The numbers type.</para>
115    /// <para>Тип чисел.</para>
116    /// </typeparam>
117    /// </summary>
118    /// <param name="x">
119    /// <para>First multiplier.</para>
120    /// <para>Первый множитель.</para>
121    /// </param>
122    /// <param name="y">
123    /// <para>Second multiplier.</para>
124    /// <para>Второй множитель.</para>
125    /// </param>
126    /// <returns>
127    /// <para>Logical product of numbers.</para>
128    /// <para>Логическое произведение чисел.</para>
129    /// </returns>
130    [MethodImpl(MethodImplOptions.AggressiveInlining)]
131    public static T And<T>(T x, T y) => Bit<T>.And(x, y);
132
133    /// <summary>
134    /// <para>Performs a bitwise shift of a number to the left by the specified number of
135    /// → bits.</para>
136    /// <para>Выполняет побитовый сдвиг числа влево на указанное количество бит.</para>
137    /// </summary>
138    /// <typeparam name="T">
139    /// <para>The number type.</para>
140    /// <para>Тип числа.</para>
141    /// </typeparam>
142    /// <param name="x">
143    /// <para>The number on which the left bitwise shift operation will be performed.</para>
144    /// <para>Число над которым будет производиться операция побитового сдвига влево.</para>
145    /// </param>
146    /// <param name="y">
147    /// <para>The number of bits to shift.</para>
148    /// <para>Количество бит на которые выполнить сдвиг.</para>
149    /// </param>
150    /// <returns>
151    /// <para>The value with discarded high-order bits that are outside the range of the
152    /// → number type and set low-order empty bit positions to zero.</para>
153    /// <para>Значение с отброшенными старшими битами, которые находятся за пределами
154    /// → диапазона типа числа и установленными пустыми битовыми позициями младших разрядов
155    /// → в ноль.</para>
156    /// </returns>
157    [MethodImpl(MethodImplOptions.AggressiveInlining)]
158    public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
159
160    /// <summary>
161    /// <para>Performs a bitwise shift of a number to the right by the specified number of
162    /// → bits.</para>
163    /// <para>Выполняет побитовый сдвиг числа вправо на указанное количество бит.</para>
164    /// </summary>

```

```

160 /// <typeparam name="T">
161 /// <para>The number type.</para>
162 /// <para>Тип числа.</para>
163 /// </typeparam>
164 /// <param name="x">
165 /// <para>The number on which the right bitwise shift operation will be performed.</para>
166 /// <para>Число над которым будет производиться операция побитового сдвига
    → вправо.</para>
167 /// </param>
168 /// <param name="y">
169 /// <para>The number of bits to shift.</para>
170 /// <para>Количество бит на которые выполнить сдвиг.</para>
171 /// </param>
172 /// <returns>
173 /// <para>The value with discarded low-order bits.</para>
174 /// <para>Значение с отброшенными младшими битами.</para>
175 /// </returns>
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
178
179 /// <summary>
180 /// <para>Performs a partial write of a specified number of bits from source number to
    → target number.</para>
181 /// <para>Выполняет частичную запись определенного количества бит исходного числа в
    → целевое число.</para>
182 /// </summary>
183 /// <typeparam name="T">
184 /// <para>The numbers type.</para>
185 /// <para>Тип чисел.</para>
186 /// </typeparam>
187 /// <param name="target">
188 /// <para>The value to which the partial write will be performed.</para>
189 /// <para>Значение в которое будет выполнена частичная запись.</para>
190 /// </param>
191 /// <param name="source">
192 /// <para>Data source for recording.</para>
193 /// <para>Источник данных для записи.</para>
194 /// </param>
195 /// <param name="shift">
196 /// <para>The start position to read from.</para>
197 /// <para>Стартовая позиция чтения.</para>
198 /// </param>
199 /// <param name="limit">
200 /// <para>The number of bits to write from source to target.</para>
201 /// <para>Количество бит, которые нужно записать из source в target.</para>
202 /// </param>
203 /// <returns>
204 /// <para>The target number updated with bits from source number.</para>
205 /// <para>Целевое число с обновленными битами из исходного числа.</para>
206 /// </returns>
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
    → Bit<T>.PartialWrite(target, source, shift, limit);
209
210 /// <summary>
211 /// <para>Reads a specified number of bits from the number at specified position.</para>
212 /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
213 /// </summary>
214 /// <typeparam name="T">
215 /// <para>The number type.</para>
216 /// <para>Тип числа.</para>
217 /// </typeparam>
218 /// <param name="target">
219 /// <para>The number from which the partial read will be performed.</para>
220 /// <para>Число из которого будет выполнено частичное чтение.</para>
221 /// </param>
222 /// <param name="shift">
223 /// <para>The start position to read from.</para>
224 /// <para>Стартовая позиция чтения.</para>
225 /// </param>
226 /// <param name="limit">
227 /// <para>The number of bits to read.</para>
228 /// <para>Количество бит, которые нужно считать.</para>
229 /// </param>
230 /// <returns>
231 /// <para>The number consisting of bits read from the source number.</para>
232 /// <para>Число состоящее из считанных из исходного числа бит.</para>
233 /// </returns>

```

```

234     [MethodImpl(MethodImplOptions.AggressiveInlining)]
235     public static T PartialRead<T>(T target, int shift, int limit) =>
        ↳ Bit<T>.PartialRead(target, shift, limit);
236 }
237 }

```

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      /// <summary>
8      /// <returns> bit operations </returns>
9      /// </summary>
10     /* bit operations */
11     public static class BitwiseExtensions
12     {
13
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
            ↳ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static T PartialRead<T>(this T target, int shift, int limit) =>
            ↳ Bit<T>.PartialRead(target, shift, limit);
23     }
24 }

```

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 =
            ↳ CompilePartialWriteDelegate();
19         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         private static Func<T, T> CompileNotDelegate()
23         {
24             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
25             {
26                 Ensure.Always.IsNumeric<T>();
27                 emitter.LoadArguments(0);
28                 emitter.Not();
29                 emitter.Return();
30             });
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         private static Func<T, T, T> CompileOrDelegate()
35         {
36             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
37             {
38                 Ensure.Always.IsNumeric<T>();
39                 emitter.LoadArguments(0, 1);
40                 emitter.Or();
41                 emitter.Return();
42             });
43         }

```



```

44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 private static Func<T, T, T> CompileAndDelegate()
46 {
47     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
48     {
49         Ensure.Always.IsNumeric<T>();
50         emitter.LoadArguments(0, 1);
51         emitter.And();
52         emitter.Return();
53     });
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 private static Func<T, int, T> CompileShiftLeftDelegate()
58 {
59     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
60     {
61         Ensure.Always.IsNumeric<T>();
62         emitter.LoadArguments(0, 1);
63         emitter.ShiftLeft();
64         emitter.Return();
65     });
66 }
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 private static Func<T, int, T> CompileShiftRightDelegate()
70 {
71     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
72     {
73         Ensure.Always.IsNumeric<T>();
74         emitter.LoadArguments(0, 1);
75         emitter.ShiftRight();
76         emitter.Return();
77     });
78 }
79
80 [MethodImpl(MethodImplOptions.AggressiveInlining)]
81 private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
82 {
83     return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
84     {
85         Ensure.Always.IsNumeric<T>();
86         var constants = GetConstants();
87         var bitsNumber = constants.Item1;
88         var numberFilledWithOnes = constants.Item2;
89         ushort shiftArgument = 2;
90         ushort limitArgument = 3;
91         var checkLimit = emitter.DefineLabel();
92         var calculateSourceMask = emitter.DefineLabel();
93         // Check shift
94         emitter.LoadArgument(shiftArgument);
95         emitter.LoadConstant(0);
96         emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
97         // Fix shift
98         emitter.LoadConstant(bitsNumber);
99         emitter.LoadArgument(shiftArgument);
100         emitter.Add();
101         emitter.StoreArgument(shiftArgument);
102         emitter.MarkLabel(checkLimit);
103         // Check limit
104         emitter.LoadArgument(limitArgument);
105         emitter.LoadConstant(0);
106         emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
107         // Fix limit
108         emitter.LoadConstant(bitsNumber);
109         emitter.LoadArgument(limitArgument);
110         emitter.Add();
111         emitter.StoreArgument(limitArgument);
112         emitter.MarkLabel(calculateSourceMask);
113         var sourceMask = emitter.DeclareLocal<T>();
114         var targetMask = emitter.DeclareLocal<T>();
115         emitter.LoadConstant(typeof(T), numberFilledWithOnes);
116         emitter.LoadArgument(limitArgument);
117         emitter.ShiftLeft();
118         emitter.Not();
119         emitter.LoadConstant(typeof(T), numberFilledWithOnes);
120         emitter.And();
121

```

```

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.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Numbers
7  {
8      /// <remarks>
9      /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
10     /// </remarks>
11     public static class Math
12     {
13         /// <remarks>
14         /// <para>Source: https://oeis.org/A000142/list </para>
15         /// <para>Источник: https://oeis.org/A000142/list </para>
16         /// </remarks>
17
18         private static readonly ulong[] _factorials =
19         {
20             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
21             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
22             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
23         };
24
25         /// <remarks>
26         /// <para>Source: https://oeis.org/A000108/list </para>
27         /// <para>Источник: https://oeis.org/A000108/list </para>
28         /// </remarks>
29         private static readonly ulong[] _catalans =
30         {
31             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
32             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
33             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324,
34             ↪ 4861946401452,
35             18367353072152, 69533550916004, 263747951750360, 1002242216651368,
36             ↪ 3814986502092304,
37             14544636039226909, 55534064877048198, 212336130412243110, 812944042149730764,
38             ↪ 3116285494907301262, 11959798385860453492
39         };
40
41         public static readonly ulong MaximumFactorialNumber = 20;
42
43         public static readonly ulong MaximumCatalanIndex = 36;
44
45         /// <summary>
46         /// <para>Returns the product of all positive integers less than or equal to the number
47         ↪ specified as an argument.</para>
48         /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
49         ↪ в качестве аргумента числу.</para>
50         /// </summary>
51         /// <param name="n"><para>The maximum positive number that will participate in
52         ↪ factorial's product.</para><para>Максимальное положительное число, которое будет
53         ↪ участвовать в произведение факториала.</para></param>
54         /// <returns><para>The product of all positive integers less than or equal to the number
55         ↪ specified as an argument.</para><para>Произведение всех положительных чисел меньше
56         ↪ или равных указанному в качестве аргумента числу.</para></returns>
57         public static ulong Factorial(ulong n)

```

```

49 {
50     if (n >= 0 && n <= MaximumFactorialNumber)
51     {
52         return _factorials[n];
53     }
54     else
55     {
56         throw new ArgumentOutOfRangeException($"Only numbers from 0 to
           ↳ {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
           ↳ length.");
57     }
58 }
59
60 /// <summary>
61 /// <para>Returns the Catalan Number with the number specified as an argument.</para>
62 /// <para>Возвращает Каталановое число с номером указанным в качестве аргумента.</para>
63 /// </summary>
64 /// <param name="n"><para>The number of Catalan number.</para><para>Номер Каталанового
   ↳ числа.</para></param>
65 /// <returns><para>The Catalan Number with the number specified as an
   ↳ argument.</para><para>Каталановое число с номером указанным в качестве
   ↳ аргумента.</para></returns>
66 public static ulong Catalan(ulong n)
67 {
68     if (n >= 0 && n <= MaximumCatalanIndex)
69     {
70         return _catalans[n];
71     }
72     else
73     {
74         throw new ArgumentOutOfRangeException($"Only numbers from 0 to
           ↳ {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
           ↳ length.");
75     }
76 }
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public static T Abs<T>(T x) => Math<T>.Abs(x);
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static T Negate<T>(T x) => Math<T>.Negate(x);
86 }
87 }

```

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     /// <summary>
8     /// <para>Provides a set of extension methods that perform mathematical operations on
   ↳ arbitrary object types.</para>
9     /// <para>Предоставляет набор методов расширения выполняющих математические операции для
   ↳ объектов произвольного типа.</para>
10    /// </summary>
11    public static class MathExtensions
12    {
13        /// <summary>
14        /// <para>Takes a module from a number.</para>
15        /// <para>Берёт модуль от числа.</para>
16        /// </summary>
17        /// <typeparam name="T">
18        /// <para>The number type.</para>
19        /// <para>Тип числа.</para>
20        /// </typeparam>
21        /// <param name="x">
22        /// <para>The number from which to take the absolute value.</para>
23        /// <para>Число от которого необходимо взять абсолютное значение.</para>
24        /// </param>
25        /// <returns>
26        /// <para>The absolute value of a number.</para>
27        /// <para>Абсолютное значение числа.</para>
28        /// </returns>

```

```

29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
31
32     /// <summary>
33     /// <para>Makes a number negative.</para>
34     /// <para>Делает число отрицательным.</para>
35     /// </summary>
36     /// <typeparam name="T">
37     /// <para>The number type.</para>
38     /// <para>Тип числа.</para>
39     /// </typeparam>
40     /// <param name="x">
41     /// <para>The number to be made negative.</para>
42     /// <para>Число которое нужно сделать отрицательным.</para>
43     /// </param>
44     /// <returns>
45     /// <para>Negative number.</para>
46     /// <para>Отрицательное число.</para>
47     /// </returns>
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
50 }
51 }

```

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            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         // 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     uint unpackagedFirstValue = value & 1;
122     uint unpackagedSecondValue = (value & 0xFFFFFFF) >> 1;
123
124     Assert.True(firstValue == unpackagedFirstValue);
125     Assert.True(secondValue == unpackagedSecondValue);
126
127     // Using universal functions:
128     var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
129     var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
130     var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
131     var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
132
133     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
134     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
135
136     firstValue = 0;
137     secondValue = 6892;
138
139     value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
140     value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
141
142     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
143     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
144 }
145
146 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
147 private static uint PartialWrite(uint target, uint source, int shift, int limit)
148 {
149     if (shift < 0)
150     {
151         shift = 32 + shift;
152     }
153     if (limit < 0)
154     {
155         limit = 32 + limit;
156     }
157     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
158     var targetMask = ~(sourceMask << shift);
159     return target & targetMask | (source & sourceMask) << shift;
160 }
161
162 private static uint PartialRead(uint target, int shift, int limit)
163 {
164     if (shift < 0)
165     {
166         shift = 32 + shift;
167     }
168     if (limit < 0)
169     {
170         limit = 32 + limit;
171     }
172     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
173     var targetMask = sourceMask << shift;
174     return (target & targetMask) >> shift;
175 }
176
177 private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
178 {
179     if (shift < 0)
180     {
181         shift = 32 + shift;
182     }
183     if (limit < 0)
184     {
185         limit = 32 + limit;
186     }
187     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
188     var targetMask = ~(sourceMask << shift);
189     return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
190 }
191
192 private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
193 {
194     if (shift < 0)
195     {
196         shift = 32 + shift;
197     }
198 }
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     }

```

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

```
77         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
78         var targetMask = sourceMask << shift;
79         return (target & targetMask) >> shift;
80     }
81 }
82 }
```

Index

- ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs, 13
- ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs, 14
- ./csharp/Platform.Numbers.Tests/BitTests.cs, 14
- ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs, 17
- ./csharp/Platform.Numbers.Tests/MathTests.cs, 17
- ./csharp/Platform.Numbers.Tests/SystemTests.cs, 18
- ./csharp/Platform.Numbers/Arithmetic.cs, 1
- ./csharp/Platform.Numbers/ArithmeticExtensions.cs, 3
- ./csharp/Platform.Numbers/Arithmetic[T].cs, 3
- ./csharp/Platform.Numbers/Bit.cs, 4
- ./csharp/Platform.Numbers/BitExtensions.cs, 8
- ./csharp/Platform.Numbers/Bit[T].cs, 8
- ./csharp/Platform.Numbers/Math.cs, 11
- ./csharp/Platform.Numbers/MathExtensions.cs, 12
- ./csharp/Platform.Numbers/Math[T].cs, 13