

LinksPlatform's Platform.Numbers Class Library

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

```
1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Numbers
4  {
5      /// <summary>
6      /// <para>Represents a set of arithmetic methods.</para>
7      /// <para>Представляет набор арифметических методов.</para>
8      /// </summary>
9      public static class Arithmetic
10     {
11         /// <summary>
12         /// <para>Performing adding the x and y arguments.</para>
13         /// <para>Выполняет сложение аргументов x и y.</para>
14         /// </summary>
15         /// <typeparam name="T">
16         /// <para>The numbers' type.</para>
17         /// <para>Тип чисел.</para>
18         /// </typeparam>
19         /// <param name="x">
20         /// <para>The first term.</para>
21         /// <para>Первое слагаемое.</para>
22         /// </param>
23         /// <param name="y">
24         /// <para>The second term.</para>
25         /// <para>Второе слагаемое.</para>
26         /// </param>
27         /// <returns>
28         /// <para>Sum of x and y.</para>
29         /// <para>Сумма x и y.</para>
30         /// </returns>
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
33
34         /// <summary>
35         /// <para>Performs subtracting y from x.</para>
36         /// <para>Выполняет вычитание y из x.</para>
37         /// </summary>
38         /// <typeparam name="T">
39         /// <para>The numbers' type.</para>
40         /// <para>Тип чисел.</para>
41         /// </typeparam>
42         /// <param name="x">
43         /// <para>Minuend.</para>
44         /// <para>Уменьшаемое.</para>
45         /// </param>
46         /// <param name="y">
47         /// <para>Subtrahend.</para>
48         /// <para>Вычитаемое.</para>
49         /// </param>
50         /// <returns>
51         /// <para>Difference between x and y.</para>
52         /// <para>Разность между x и y.</para>
53         /// </returns>
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
56
57         /// <summary>
58         /// <para>Performs multiplication x by y.</para>
59         /// <para>Выполняет умножение x на y.</para>
60         /// </summary>
61         /// <typeparam name="T">
62         /// <para>The numbers' type.</para>
63         /// <para>Тип чисел.</para>
64         /// </typeparam>
65         /// <param name="x">
66         /// <para>First multiplier.</para>
67         /// <para>Первый множитель.</para>
68         /// </param>
69         /// <param name="y">
70         /// <para>Second multiplier.</para>
71         /// <para>Второй множитель.</para>
72         /// </param>
73         /// <returns>
74         /// <para>Product of x and y.</para>
75         /// <para>Произведение x и y.</para>
76         /// </returns>
```

```

77 [MethodImpl(MethodImplOptions.AggressiveInlining)]
78 public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);
79
80 /// <summary>
81 /// <para>Performs dividing x by y.</para>
82 /// <para>Выполняет деление x на y.</para>
83 /// </summary>
84 /// <typeparam name="T">
85 /// <para>The numbers' type.</para>
86 /// <para>Тип чисел.</para>
87 /// </typeparam>
88 /// <param name="x">
89 /// <para>Dividend.</para>
90 /// <para>Делимое.</para>
91 /// </param>
92 /// <param name="y">
93 /// <para>Divider.</para>
94 /// <para>Делитель.</para>
95 /// </param>
96 /// <returns>
97 /// <para>Quoitent of x and y.</para>
98 /// <para>Частное x и y.</para>
99 /// </returns>
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);
102
103 /// <summary>
104 /// <para>Increasing the number by one.</para>
105 /// <para>Увеличивает число на единицу.</para>
106 /// </summary>
107 /// <typeparam name="T">
108 /// <para>The number's type.</para>
109 /// <para>Тип числа.</para>
110 /// </typeparam>
111 /// <param name="x">
112 /// <para>The number to increase.</para>
113 /// <para>Число для увеличения.</para>
114 /// </param>
115 /// <returns>
116 /// <para>Increase by one number.</para>
117 /// <para>Увеличенное на единицу число.</para>
118 /// </returns>
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
121
122 /// <summary>
123 /// <para>Increases the value of argument by one.</para>
124 /// <para>Увеличивает значение аргумента на единицу.</para>
125 /// </summary>
126 /// <typeparam name="T">
127 /// <para>The number's type.</para>
128 /// <para>Тип числа.</para>
129 /// </typeparam>
130 /// <param name="x">
131 /// <para>The argument to increase.</para>
132 /// <para>Аргумент для увеличения.</para>
133 /// </param>
134 /// <returns>
135 /// <para>Increased argument value.</para>
136 /// <para>Увеличенное значение аргумента.</para>
137 /// </returns>
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);
140
141 /// <summary>
142 /// <para>Decreases number by one.</para>
143 /// <para>Уменьшение числа на единицу.</para>
144 /// </summary>
145 /// <typeparam name="T">
146 /// <para>The number's type.</para>
147 /// <para>Тип числа.</para>
148 /// </typeparam>
149 /// <param name="x">
150 /// <para>The number to reduce.</para>
151 /// <para>Число для уменьшения.</para>
152 /// </param>
153 /// <returns>
154 /// <para>Decreased by one number.</para>

```

```

155     /// <para>Уменьшенное на единицу число.</para>
156     /// </returns>
157     [MethodImpl(MethodImplOptions.AggressiveInlining)]
158     public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
159
160     /// <summary>
161     /// <para>Decreases the value of the argument by one.</para>
162     /// <para>Уменьшает значение аргумента на единицу.</para>
163     /// </summary>
164     /// <typeparam name="T">
165     /// <para>The number's type.</para>
166     /// <para>Тип числа.</para>
167     /// </typeparam>
168     /// <param name="x">
169     /// <para>The argument to reduce.</para>
170     /// <para>Аргумент для уменьшения.</para>
171     /// </param>
172     /// <returns>
173     /// <para>Decreased argument value.</para>
174     /// <para>Уменьшенное значение аргумента.</para>
175     /// </returns>
176     [MethodImpl(MethodImplOptions.AggressiveInlining)]
177     public static T Decrement<T>(ref T x) => x = Arithmetic<T>.Decrement(x);
178 }
179 }

```

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

```

1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Numbers
4  {
5      /// <summary>
6      /// <para>Represents a set of extension methods that perform arithmetic operations on
7      ///   ↳ arbitrary object types.</para>
8      /// <para>Представляет набор методов расширения выполняющих арифметические операции для
9      ///   ↳ объектов произвольного типа.</para>
10     /// </summary>
11     public static class ArithmeticExtensions
12     {
13         /// <summary>
14         /// <para>Increments the variable passed as an argument by one.</para>
15         /// <para>Увеличивает переданную в качестве аргумента переменную на единицу.</para>
16         /// </summary>
17         /// <typeparam name="T">
18         /// <para>The number's type.</para>
19         /// <para>Тип числа.</para>
20         /// </typeparam>
21         /// <param name="x">
22         /// <para>The reference to the incremented variable.</para>
23         /// <para>Ссылка на увеличиваемую переменную.</para>
24         /// </param>
25         /// <returns>
26         /// <para>The value of the argument incremented by one.</para>
27         /// <para>Увеличенное значение аргумента на единицу.</para>
28         /// </returns>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public static T Increment<T>(this ref T x) where T : struct => x =
31             ↳ Arithmetic<T>.Increment(x);
32
33         /// <summary>
34         /// <para>Decrements the variable passed as an argument by one.</para>
35         /// <para>Уменьшает переданную в качестве аргумента переменную на единицу.</para>
36         /// </summary>
37         /// <typeparam name="T">
38         /// <para>The number's type.</para>
39         /// <para>Тип числа.</para>
40         /// </typeparam>
41         /// <param name="x">
42         /// <para>The reference to the decremented variable.</para>
43         /// <para>Ссылка на уменьшаемую переменную.</para>
44         /// </param>
45         /// <returns>
46         /// <para>The value of the argument decremented by one.</para>
47         /// <para>Уменьшенное значение аргумента на единицу.</para>
48         /// </returns>
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public static T Decrement<T>(this ref T x) where T : struct => x =
51             ↳ Arithmetic<T>.Decrement(x);

```

```
48     }
49 }
```

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  namespace Platform.Numbers
9  {
10     /// <summary>
11     /// <para>Represents a set of compiled arithmetic operations delegates.</para>
12     /// <para>Представляет набор скомпилированных делегатов арифметических операций.</para>
13     /// </summary>
14     public static class Arithmetic<T>
15     {
16         /// <summary>
17         /// <para>A read-only field that represents a addition function delegate.</para>
18         /// <para>Поле только для чтения, представляющее делегат функции сложения.</para>
19         /// </summary>
20         public static readonly Func<T, T, T> Add = CompileAddDelegate();
21
22         /// <summary>
23         /// <para>A read-only field that represents a subtraction function delegate.</para>
24         /// <para>Поле только для чтения, представляющее делегат функции вычитания.</para>
25         /// </summary>
26         public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
27
28         /// <summary>
29         /// <para>A read-only field that represents a multiplication function delegate.</para>
30         /// <para>Поле только для чтения, представляющее делегат функции умножения.</para>
31         /// </summary>
32         public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
33
34         /// <summary>
35         /// <para>A read-only field that represents a division function delegate.</para>
36         /// <para>Поле только для чтения, представляющее делегат функции деления.</para>
37         /// </summary>
38         public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
39
40         /// <summary>
41         /// <para>A read-only field that represents a increment function delegate.</para>
42         /// <para>Поле только для чтения, представляющее делегат функции инкремента.</para>
43         /// </summary>
44         public static readonly Func<T, T> Increment = CompileIncrementDelegate();
45
46         /// <summary>
47         /// <para>A read-only field that represents a decrement function delegate.</para>
48         /// <para>Поле только для чтения, представляющее делегат функции декремента.</para>
49         /// </summary>
50         public static readonly Func<T, T> Decrement = CompileDecrementDelegate();
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     private static Func<T, T, T> CompileAddDelegate()
53     {
54         return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
55         {
56             Ensure.Always.IsNumeric<T>();
57             emitter.LoadArguments(0, 1);
58             emitter.Add();
59             emitter.Return();
60         });
61     }
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     private static Func<T, T, T> CompileSubtractDelegate()
64     {
65         return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
66         {
67             Ensure.Always.IsNumeric<T>();
68             emitter.LoadArguments(0, 1);
69             emitter.Subtract();
70             emitter.Return();
71         });
72     }
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     private static Func<T, T, T> CompileMultiplyDelegate()
75     {
```

```

76         return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
77         {
78             Ensure.Always.IsNumeric<T>();
79             emitter.LoadArguments(0, 1);
80             emitter.Emit(OpCodes.Mul);
81             emitter.Return();
82         });
83     }
84     [MethodImpl(MethodImplOptions.AggressiveInlining)]
85     private static Func<T, T, T> CompileDivideDelegate()
86     {
87         return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
88         {
89             Ensure.Always.IsNumeric<T>();
90             emitter.LoadArguments(0, 1);
91             if(NumericType<T>.IsSigned)
92             {
93                 emitter.Emit(OpCodes.Div);
94             }
95             else
96             {
97                 emitter.Emit(OpCodes.Div_Un);
98             }
99             emitter.Return();
100         });
101     }
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     private static Func<T, T> CompileIncrementDelegate()
104     {
105         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
106         {
107             Ensure.Always.IsNumeric<T>();
108             emitter.LoadArgument(0);
109             emitter.Increment<T>();
110             emitter.Return();
111         });
112     }
113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
114     private static Func<T, T> CompileDecrementDelegate()
115     {
116         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
117         {
118             Ensure.Always.IsNumeric<T>();
119             emitter.LoadArgument(0);
120             emitter.Decrement<T>();
121             emitter.Return();
122         });
123     }
124 }
125 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Numbers
4 {
5     /// <summary>
6     /// <para>A set of operations on the set bits of a number.</para>
7     /// <para>Набор операций над установленными битами числа.</para>
8     /// </summary>
9     public static class Bit
10     {
11         /// <summary>
12         /// <para>Counts the number of bits set in a number.</para>
13         /// <para>Подсчитывает количество установленных бит в числе.</para>
14         /// </summary>
15         /// <param name="x">
16         /// <para>Bitwise number.</para>
17         /// <para>Число в битовом представлении.</para>
18         /// </param>
19         /// <returns>
20         /// <para>Number of bits set in a number.</para>
21         /// <para>Количество установленных бит в числе.</para>
22         /// </returns>
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static long Count(long x)
25         {
26             long n = 0;

```

```

27     while (x != 0)
28     {
29         n++;
30         x &= x - 1;
31     }
32     return n;
33 }
34
35 /// <summary>
36 /// <para>Searches for the first bit set in a number.</para>
37 /// <para>Ищет первый установленный бит в числе.</para>
38 /// </summary>
39 /// <param name="value">
40 /// <para>Bitwise number.</para>
41 /// <para>Число в битовом представлении.</para>
42 /// </param>
43 /// <returns>
44 /// <para>First bit set.</para>
45 /// <para>Первый установленный бит.</para>
46 /// </returns>
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 public static int GetLowestPosition(ulong value)
49 {
50     if (value == 0)
51     {
52         return -1;
53     }
54     var position = 0;
55     while ((value & 1UL) == 0)
56     {
57         value >>= 1;
58         ++position;
59     }
60     return position;
61 }
62
63 /// <summary>
64 /// <para>Performing bitwise inversion of a number.</para>
65 /// <para>Выполняет побитовую инверсию числа.</para>
66 /// </summary>
67 /// <typeparam name="T">
68 /// <para>The number's type.</para>
69 /// <para>Тип числа.</para>
70 /// </typeparam>
71 /// <param name="x">
72 /// <para>Number to invert.</para>
73 /// <para>Число для инверсии.</para>
74 /// </param>
75 /// <returns>
76 /// <para>Inverse value of the number.</para>
77 /// <para>Обратное значение числа.</para>
78 /// </returns>
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 public static T Not<T>(T x) => Bit<T>.Not(x);
81
82 /// <summary>
83 /// <para>Performing bitwise numbers addition.</para>
84 /// <para>Выполняет побитовое сложение чисел.</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>First term.</para>
92 /// <para>Первое слагаемое.</para>
93 /// </param>
94 /// <param name="y">
95 /// <para>Second term.</para>
96 /// <para>Второе слагаемое.</para>
97 /// </param>
98 /// <returns>
99 /// <para>The logical sum of numbers</para>
100 /// <para>Логическая сумма чисел.</para>
101 /// </returns>
102 [MethodImpl(MethodImplOptions.AggressiveInlining)]
103 public static T Or<T>(T x, T y) => Bit<T>.Or(x, y);
104
105 /// <summary>

```

```

106 /// <para>Performs bitwise numbers multiplication.</para>
107 /// <para>Выполняет побитовое умножение чисел.</para>
108 /// <typeparam name="T">
109 /// <para>The numbers' type.</para>
110 /// <para>Тип чисел.</para>
111 /// </typeparam>
112 /// </summary>
113 /// <param name="x">
114 /// <para>First multiplier.</para>
115 /// <para>Первый множитель.</para>
116 /// </param>
117 /// <param name="y">
118 /// <para>Second multiplier.</para>
119 /// <para>Второй множитель.</para>
120 /// </param>
121 /// <returns>
122 /// <para>Logical product of numbers.</para>
123 /// <para>Логическое произведение чисел.</para>
124 /// </returns>
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public static T And<T>(T x, T y) => Bit<T>.And(x, y);
127
128 /// <summary>
129 /// <para>Performs a bitwise shift of a number to the left by the specified number of
130 /// → bits.</para>
131 /// <para>Выполняет побитовый сдвиг числа влево на указанное количество бит.</para>
132 /// </summary>
133 /// <typeparam name="T">
134 /// <para>The number's type.</para>
135 /// <para>Тип числа.</para>
136 /// </typeparam>
137 /// <param name="x">
138 /// <para>The number on which the left bitwise shift operation will be performed.</para>
139 /// <para>Число над которым будет производиться операция побитового сдвига
140 /// → влево.</para>
141 /// </param>
142 /// <param name="y">
143 /// <para>The number of bits to shift.</para>
144 /// <para>Количество бит на которые выполнить сдвиг.</para>
145 /// </param>
146 /// <returns>
147 /// <para>The value with discarded high-order bits that are outside the range of the
148 /// → number's type and set low-order empty bit positions to zero.</para>
149 /// <para>Значение с отброшенными старшими битами, которые находятся за пределами
150 /// → диапазона типа числа и установленными пустыми битовыми позициями младших разрядов
151 /// → в ноль.</para>
152 /// </returns>
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
155
156 /// <summary>
157 /// <para>Performs a bitwise shift of a number to the right by the specified number of
158 /// → bits.</para>
159 /// <para>Выполняет побитовый сдвиг числа вправо на указанное количество бит.</para>
160 /// </summary>
161 /// <typeparam name="T">
162 /// <para>The number's type.</para>
163 /// <para>Тип числа.</para>
164 /// </typeparam>
165 /// <param name="x">
166 /// <para>The number on which the right bitwise shift operation will be performed.</para>
167 /// <para>Число над которым будет производиться операция побитового сдвига
168 /// → вправо.</para>
169 /// </param>
170 /// <param name="y">
171 /// <para>The number of bits to shift.</para>
172 /// <para>Количество бит на которые выполнить сдвиг.</para>
173 /// </param>
174 /// <returns>
175 /// <para>The value with discarded low-order bits.</para>
176 /// <para>Значение с отброшенными младшими битами.</para>
177 /// </returns>
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
180
181 /// <summary>

```

```

175     /// <para>Performs a partial write of a specified number of bits from source number to
    ↪ target number.</para>
176     /// <para>Выполняет частичную запись определенного количества бит исходного числа в
    ↪ целевое число.</para>
177     /// </summary>
178     /// <typeparam name="T">
179     /// <para>The numbers' type.</para>
180     /// <para>Тип чисел.</para>
181     /// </typeparam>
182     /// <param name="target">
183     /// <para>The value to which the partial write will be performed.</para>
184     /// <para>Значение в которое будет выполнена частичная запись.</para>
185     /// </param>
186     /// <param name="source">
187     /// <para>Data source for recording.</para>
188     /// <para>Источник данных для записи.</para>
189     /// </param>
190     /// <param name="shift">
191     /// <para>The start position to read from.</para>
192     /// <para>Стартовая позиция чтения.</para>
193     /// </param>
194     /// <param name="limit">
195     /// <para>The number of bits to write from source to target.</para>
196     /// <para>Количество бит, которые нужно записать из source в target.</para>
197     /// </param>
198     /// <returns>
199     /// <para>The target number updated with bits from source number.</para>
200     /// <para>Целевое число с обновленными битами из исходного числа.</para>
201     /// </returns>
202     [MethodImpl(MethodImplOptions.AggressiveInlining)]
203     public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
    ↪ Bit<T>.PartialWrite(target, source, shift, limit);
204
205     /// <summary>
206     /// <para>Reads a specified number of bits from the number at specified position.</para>
207     /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
208     /// </summary>
209     /// <typeparam name="T">
210     /// <para>The number's type.</para>
211     /// <para>Тип числа.</para>
212     /// </typeparam>
213     /// <param name="target">
214     /// <para>The number from which the partial read will be performed.</para>
215     /// <para>Число из которого будет выполнено частичное чтение.</para>
216     /// </param>
217     /// <param name="shift">
218     /// <para>The start position to read from.</para>
219     /// <para>Стартовая позиция чтения.</para>
220     /// </param>
221     /// <param name="limit">
222     /// <para>The number of bits to read.</para>
223     /// <para>Количество бит, которые нужно считать.</para>
224     /// </param>
225     /// <returns>
226     /// <para>The number consisting of bits read from the source number.</para>
227     /// <para>Число состоящее из считанных из исходного числа бит.</para>
228     /// </returns>
229     [MethodImpl(MethodImplOptions.AggressiveInlining)]
230     public static T PartialRead<T>(T target, int shift, int limit) =>
    ↪ Bit<T>.PartialRead(target, shift, limit);
231 }
232 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Numbers
4 {
5     /// <summary>
6     /// <para>Represents a set of bitwise operation.</para>
7     /// <para>Представляет набор битовых операций.</para>
8     /// </summary>
9     public static class BitwiseExtensions
10     {
11         /// <summary>
12         /// <para>Performs bitwise inversion of a number.</para>
13         /// <para>Выполняет побитовую инверсию числа.</para>
14         /// </summary>

```



```

15  /// <typeparam name="T">
16  /// <para>The number's type.</para>
17  /// <para>Тип числа.</para>
18  /// </typeparam>
19  /// <param name="target">
20  /// <para>The number to invert.</para>
21  /// <para>Число для инверсии.</para>
22  /// </param>
23  /// <returns>
24  /// <para>An inverted value of the number.</para>
25  /// <para>Обратное значение числа.</para>
26  /// </returns>
27  [MethodImpl(MethodImplOptions.AggressiveInlining)]
28  public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
29
30  /// <summary>
31  /// <para>Performs a partial write of a specified number of bits from source number to
32  ↪ target number.</para>
33  /// <para>Выполняет частичную запись определенного количества бит исходного числа в
34  ↪ целевое число.</para>
35  /// </summary>
36  /// <typeparam name="T">
37  /// <para>The numbers' type.</para>
38  /// <para>Тип чисел.</para>
39  /// </typeparam>
40  /// <param name="target">
41  /// <para>The value to which the partial write will be performed.</para>
42  /// <para>Значение в которое будет выполнена частичная запись.</para>
43  /// </param>
44  /// <param name="source">
45  /// <para>Data source for writing.</para>
46  /// <para>Источник данных для записи.</para>
47  /// </param>
48  /// <param name="shift">
49  /// <para>The start position to read from.</para>
50  /// <para>Стартовая позиция чтения.</para>
51  /// </param>
52  /// <param name="limit">
53  /// <para>The number of bits to write from source to target.</para>
54  /// <para>Количество бит, которые нужно записать из source в target.</para>
55  /// </param>
56  /// <returns>
57  /// <para>The target number updated with bits from source number.</para>
58  /// <para>Целевое число с обновленными битами из исходного числа.</para>
59  /// </returns>
60  [MethodImpl(MethodImplOptions.AggressiveInlining)]
61  public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
62  ↪ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
63
64  /// <summary>
65  /// <para>Reads a specified number of bits from the number at specified position.</para>
66  /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
67  /// </summary>
68  /// <typeparam name="T">
69  /// <para>The number's type.</para>
70  /// <para>Тип числа.</para>
71  /// </typeparam>
72  /// <param name="target">
73  /// <para>The number from which the partial read will be performed.</para>
74  /// <para>Число из которого будет выполнено частичное чтение.</para>
75  /// </param>
76  /// <param name="shift">
77  /// <para>The start position to read from.</para>
78  /// <para>Стартовая позиция чтения.</para>
79  /// </param>
80  /// <param name="limit">
81  /// <para>The number of bits to read.</para>
82  /// <para>Количество бит, которые нужно считать.</para>
83  /// </param>
84  /// <returns>
85  /// <para>The number consisting of bits read from the source number.</para>
86  /// <para>Число состоящее из считанных из исходного числа бит.</para>
87  /// </returns>
88  [MethodImpl(MethodImplOptions.AggressiveInlining)]
89  public static T PartialRead<T>(this T target, int shift, int limit) =>
90  ↪ Bit<T>.PartialRead(target, shift, limit);
91
92  }

```

```
88 }
```

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
8 namespace Platform.Numbers
9 {
10     /// <summary>
11     /// <para>Represents a set of compiled bit operations delegates.</para>
12     /// <para>Представляет набор скомпилированных делегатов битовых операций.</para>
13     /// </summary>
14     public static class Bit<T>
15     {
16         /// <summary>
17         /// <para>A read-only field that represents a bitwise inversion function delegate.</para>
18         /// <para>Поле только для чтения, представляющее делегат функции побитовой инверсии
19         → числа.</para>
20         /// </summary>
21         public static readonly Func<T, T> Not = CompileNotDelegate();
22
23         /// <summary>
24         /// <para>A read-only field that represents a logic addition function delegate.</para>
25         /// <para>Поле только для чтения, представляющее делегат функции логического
26         → сложения.</para>
27         /// </summary>
28         public static readonly Func<T, T, T> Or = CompileOrDelegate();
29
30         /// <summary>
31         /// <para>A read-only field that represents a logic multiplication function
32         → delegate.</para>
33         /// <para>Поле только для чтения, представляющее делегат функции логического
34         → умножения.</para>
35         /// </summary>
36         public static readonly Func<T, T, T> And = CompileAndDelegate();
37
38         /// <summary>
39         /// <para>A read-only field that represents a bitwise left shift function
40         → delegate.</para>
41         /// <para>Поле только для чтения, представляющее делегат функции побитового сдвига числа
42         → влево.</para>
43         /// </summary>
44         public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
45
46         /// <summary>
47         /// <para>A read-only field that represents a bitwise right shift function
48         → delegate.</para>
49         /// <para>Поле только для чтения, представляющее делегат функции побитового сдвига числа
50         → вправо.</para>
51         /// </summary>
52         public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
53
54         /// <summary>
55         /// <para>A read-only field that represents a bitwise number representation partial
56         → rewrite function delegate.</para>
57         /// <para>Поле только для чтения, представляющее делегат функции частичной побитовой
58         → перезаписи представления числа.</para>
59         /// </summary>
60         public static readonly Func<T, T, int, int, T> PartialWrite =
61         → CompilePartialWriteDelegate();
62
63         /// <summary>
64         /// <para>A read-only field that represents a bitwise number representation partial read
65         → function delegate.</para>
66         /// <para>Поле только для чтения, представляющее делегат функции частичного побитового
67         → считывания числа.</para>
68         /// </summary>
69         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
70     }
71
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     private static Func<T, T> CompileNotDelegate()
74     {
75         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
76         {
77             Ensure.Always.IsNumeric<T>();
78         });
79     }
80 }
```

```

63         emitter.LoadArguments(0);
64         emitter.Not();
65         emitter.Return();
66     });
67 }
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 private static Func<T, T, T> CompileOrDelegate()
70 {
71     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
72     {
73         Ensure.Always.IsNumeric<T>();
74         emitter.LoadArguments(0, 1);
75         emitter.Or();
76         emitter.Return();
77     });
78 }
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 private static Func<T, T, T> CompileAndDelegate()
81 {
82     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
83     {
84         Ensure.Always.IsNumeric<T>();
85         emitter.LoadArguments(0, 1);
86         emitter.And();
87         emitter.Return();
88     });
89 }
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static Func<T, int, T> CompileShiftLeftDelegate()
92 {
93     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
94     {
95         Ensure.Always.IsNumeric<T>();
96         emitter.LoadArguments(0, 1);
97         emitter.ShiftLeft();
98         emitter.Return();
99     });
100 }
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 private static Func<T, int, T> CompileShiftRightDelegate()
103 {
104     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
105     {
106         Ensure.Always.IsNumeric<T>();
107         emitter.LoadArguments(0, 1);
108         emitter.ShiftRight<T>();
109         emitter.Return();
110     });
111 }
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
114 {
115     return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
116     {
117         Ensure.Always.IsNumeric<T>();
118         var constants = GetConstants();
119         var bitsNumber = constants.Item1;
120         var numberFilledWithOnes = constants.Item2;
121         ushort shiftArgument = 2;
122         ushort limitArgument = 3;
123         var checkLimit = emitter.DefineLabel();
124         var calculateSourceMask = emitter.DefineLabel();
125         // Check shift
126         emitter.LoadArgument(shiftArgument);
127         emitter.LoadConstant(0);
128         emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
129         // Fix shift
130         emitter.LoadConstant(bitsNumber);
131         emitter.LoadArgument(shiftArgument);
132         emitter.Add();
133         emitter.StoreArgument(shiftArgument);
134         emitter.MarkLabel(checkLimit);
135         // Check limit
136         emitter.LoadArgument(limitArgument);
137         emitter.LoadConstant(0);
138         emitter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
139         // Fix limit
140         emitter.LoadConstant(bitsNumber);

```

```

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

```

```

219         emitter.And();
220         emitter.LoadArgument(shiftArgument);
221         emitter.ShiftRight<T>();
222         emitter.Return();
223     });
224 }
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 private static Tuple<int, T> GetConstants()
227 {
228     var type = typeof(T);
229     if (type == typeof(ulong))
230     {
231         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
232     }
233     if (type == typeof(uint))
234     {
235         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
236     }
237     if (type == typeof(ushort))
238     {
239         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
240     }
241     if (type == typeof(byte))
242     {
243         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
244     }
245     throw new NotSupportedException();
246 }
247 }
248 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Numbers
5  {
6      /// <summary>
7      /// <para>Represents a set of math methods.</para>
8      /// <para>Представляет набор математических методов.</para>
9      /// </summary>
10     /// <remarks>Resizable array (FileMappedMemory) for values cache may be used. or cached
11     ↪ oeis.org</remarks>
12     public static class Math
13     {
14         private static readonly ulong[] _factorials =
15         {
16             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
17             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
18             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
19         };
20         private static readonly ulong[] _catalans =
21         {
22             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
23             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
24             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
25             18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304,
26             14544636039226909, 55534064877048198, 212336130412243110, 812944042149730764,
27             ↪ 3116285494907301262, 11959798385860453492
28         };
29
30         /// <summary>
31         /// <para>Represents the limit for calculating the catanal number, supported by the <see
32         ↪ cref="ulong"/> type.</para>
33         /// <para>Представляет предел расчёта катаналового числа, поддерживаемый <see
34         ↪ cref="ulong"/> типом.</para>
35         /// </summary>
36         public static readonly ulong MaximumFactorialNumber = 20;
37
38         /// <summary>
39         /// <para>Represents the limit for calculating the factorial number, supported by the
40         ↪ <see cref="ulong"/> type.</para>
41         /// <para>Представляет предел расчёта факториала числа, поддерживаемый <see
42         ↪ cref="ulong"/> типом.</para>
43         /// </summary>
44         public static readonly ulong MaximumCatalanIndex = 36;
45
46         /// <summary>

```

```

41  /// <para>Returns the product of all positive integers less than or equal to the number
    ↳ specified as an argument.</para>
42  /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
    ↳ в качестве аргумента числу.</para>
43  /// </summary>
44  /// <param name="n">
45  /// <para>The maximum positive number that will participate in factorial's
    ↳ product.</para>
46  /// <para>Максимальное положительное число, которое будет участвовать в произведении
    ↳ факториала.</para>
47  /// </param>
48  /// <returns>
49  /// <para>The product of all positive integers less than or equal to the number
    ↳ specified as an argument.</para>
50  /// <para>Произведение всех положительных чисел меньше или равных указанному, в качестве
    ↳ аргумента, числу.</para>
51  /// </returns>
52  public static ulong Factorial(ulong n)
53  {
54      if (n >= 0 && n <= MaximumFactorialNumber)
55      {
56          return _factorials[n];
57      }
58      else
59      {
60          throw new ArgumentOutOfRangeException($"Only numbers from 0 to
            ↳ {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
            ↳ length.");
61      }
62  }
63
64  /// <summary>
65  /// <para>Returns the Catalan Number with the number specified as an argument.</para>
66  /// <para>Возвращает Число Катанала с номером, указанным в качестве аргумента.</para>
67  /// </summary>
68  /// <param name="n">
69  /// <para>The number of the Catalan number.</para>
70  /// <para>Номер Числа Катанала.</para>
71  /// </param>
72  /// <returns>
73  /// <para>The Catalan Number with the number specified as an argument.</para>
74  /// <para>Число Катанала с номером, указанным в качестве аргумента.</para>
75  /// </returns>
76  public static ulong Catalan(ulong n)
77  {
78      if (n >= 0 && n <= MaximumCatalanIndex)
79      {
80          return _catalans[n];
81      }
82      else
83      {
84          throw new ArgumentOutOfRangeException($"Only numbers from 0 to
            ↳ {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
            ↳ length.");
85      }
86  }
87
88  /// <summary>
89  /// <para>Checks if a number is a power of two.</para>
90  /// <para>Проверяет, является ли число степенью двойки.</para>
91  /// </summary>
92  /// <param name="x">
93  /// <para>The number to check.</para>
94  /// <para>Число для проверки.</para>
95  /// </param>
96  /// <returns>
97  /// <para>True if the number is a power of two otherwise false.</para>
98  /// <para>True, если число является степенью двойки, иначе - false.</para>
99  /// </returns>
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
102
103 /// <summary>
104 /// <para>Takes a module from a number.</para>
105 /// <para>Берёт модуль от числа.</para>
106 /// </summary>
107 /// <typeparam name="T">

```

```

108     /// <para>The number's type.</para>
109     /// <para>Тип числа.</para>
110     /// </typeparam>
111     /// <param name="x">
112     /// <para>The number from which to take the absolute value.</para>
113     /// <para>Число, от которого необходимо взять абсолютное значение.</para>
114     /// </param>
115     /// <returns>
116     /// <para>The absolute value of the number.</para>
117     /// <para>Абсолютное значение числа.</para>
118     /// </returns>
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     public static T Abs<T>(T x) => Math<T>.Abs(x);
121
122     /// <summary>
123     /// <para>Makes a number negative.</para>
124     /// <para>Делает число отрицательным.</para>
125     /// </summary>
126     /// <typeparam name="T">
127     /// <para>The number's type.</para>
128     /// <para>Тип числа.</para>
129     /// </typeparam>
130     /// <param name="x">
131     /// <para>The number to be made negative.</para>
132     /// <para>Число которое нужно сделать отрицательным.</para>
133     /// </param>
134     /// <returns>
135     /// <para>A negative number.</para>
136     /// <para>Отрицательное число.</para>
137     /// </returns>
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]
139     public static T Negate<T>(T x) => Math<T>.Negate(x);
140 }
141 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Numbers
4 {
5     /// <summary>
6     /// <para>Represents a set of extension methods that perform mathematical operations on
7     /// ↳ arbitrary object types.</para>
8     /// <para>Представляет набор методов расширения выполняющих математические операции для
9     /// ↳ 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's 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's 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>

```

```

41     /// </param>
42     /// <returns>
43     /// <para>Negative number.</para>
44     /// <para>Отрицательное число.</para>
45     /// </returns>
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
48 }
49 }

```

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  namespace Platform.Numbers
8  {
9      /// <summary>
10     /// <para>Represents a set of compiled math operations delegates.</para>
11     /// <para>Представляет набор скомпилированных делегатов математических операций.</para>
12     /// </summary>
13     public static class Math<T>
14     {
15         /// <summary>
16         /// <para>A read-only field that represents a number modulus calculation function
17         ↪ delegate.</para>
18         /// <para>Поле только для чтения, которое представляет делегат функции вычисления модуля
19         ↪ числа.</para>
20         /// </summary>
21         public static readonly Func<T, T> Abs = CompileAbsDelegate();
22
23         /// <summary>
24         /// <para>A read-only field that represents a number negation function delegate.</para>
25         /// <para>Поле только для чтения, которое представляет делегат функции отрицания
26         ↪ числа.</para>
27         /// </summary>
28         public static readonly Func<T, T> Negate = CompileNegateDelegate();
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         private static Func<T, T> CompileAbsDelegate()
32         {
33             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
34             {
35                 Ensure.Always.IsNumeric<T>();
36                 emitter.LoadArgument(0);
37                 if (NumericType<T>.IsSigned)
38                 {
39                     emitter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
40                 }
41                 emitter.Return();
42             });
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         private static Func<T, T> CompileNegateDelegate()
47         {
48             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
49             {
50                 emitter.LoadArgument(0);
51                 emitter.Negate();
52                 emitter.Return();
53             });
54         }
55     }
56 }

```

1.10 ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the arithmetic extensions tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public static class ArithmeticExtensionsTests

```



```

12 {
13     /// <summary>
14     /// <para>
15     /// Tests that increment test.
16     /// </para>
17     /// <para></para>
18     /// </summary>
19     [Fact]
20     public static void IncrementTest()
21     {
22         var number = 0UL;
23         var returnValue = number.Increment();
24         Assert.Equal(1UL, returnValue);
25         Assert.Equal(1UL, number);
26     }
27
28     /// <summary>
29     /// <para>
30     /// Tests that decrement test.
31     /// </para>
32     /// <para></para>
33     /// </summary>
34     [Fact]
35     public static void DecrementTest()
36     {
37         var number = 1UL;
38         var returnValue = number.Decrement();
39         Assert.Equal(0UL, returnValue);
40         Assert.Equal(0UL, number);
41     }
42 }
43 }

```

1.11 ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs

```

1 using System;
2 using Xunit;
3
4 namespace Platform.Numbers.Tests
5 {
6     /// <summary>
7     /// <para>
8     /// Represents the arithmetic tests.
9     /// </para>
10    /// <para></para>
11    /// </summary>
12    public static class ArithmeticTests
13    {
14        /// <summary>
15        /// <para>
16        /// Tests that compiled operations test.
17        /// </para>
18        /// <para></para>
19        /// </summary>
20        [Fact]
21        public static void CompiledOperationsTest()
22        {
23            Assert.Equal(3, Arithmetic.Add(1, 2));
24            Assert.Equal(1, Arithmetic.Subtract(2, 1));
25            Assert.Equal(8, Arithmetic.Multiply(2, 4));
26            Assert.Equal(4, Arithmetic.Divide(8, 2));
27            Assert.Equal(2, Arithmetic.Increment(1));
28            Assert.Equal(1UL, Arithmetic.Decrement(2UL));
29            Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
30        }
31    }
32 }

```

1.12 ./csharp/Platform.Numbers.Tests/BitTests.cs

```

1 using System;
2 using Platform.Reflection;
3 using Xunit;
4
5 namespace Platform.Numbers.Tests
6 {
7     /// <summary>
8     /// <para>
9     /// Represents the bit tests.
10    /// </para>

```

```

11  /// <para></para>
12  /// </summary>
13  public static class BitTests
14  {
15      /// <summary>
16      /// <para>
17      /// Tests that get lowest bit position test.
18      /// </para>
19      /// <para></para>
20      /// </summary>
21      /// <param name="value">
22      /// <para>The value.</para>
23      /// <para></para>
24      /// </param>
25      /// <param name="expectedPosition">
26      /// <para>The expected position.</para>
27      /// <para></para>
28      /// </param>
29      [Theory]
30      [InlineData(00, -1)] // 0000 0000 (none, -1)
31      [InlineData(01, 00)] // 0000 0001 (first, 0)
32      [InlineData(08, 03)] // 0000 1000 (forth, 3)
33      [InlineData(88, 03)] // 0101 1000 (forth, 3)
34      public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
35      {
36          Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
37      }
38
39      /// <summary>
40      /// <para>
41      /// Tests that byte bitwise operations test.
42      /// </para>
43      /// <para></para>
44      /// </summary>
45      [Fact]
46      public static void ByteBitwiseOperationsTest()
47      {
48          Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
49          Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
50          Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
51          Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
52          Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
53          Assert.Equal(NumericType<byte>.MaxValue >> 1,
54                      ↪ Bit<byte>.ShiftRight(NumericType<byte>.MaxValue, 1));
55      }
56
57      /// <summary>
58      /// <para>
59      /// Tests that u int 16 bitwise operations test.
60      /// </para>
61      /// <para></para>
62      /// </summary>
63      [Fact]
64      public static void UInt16BitwiseOperationsTest()
65      {
66          Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
67          Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
68          Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
69          Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
70          Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
71          Assert.Equal(NumericType<ushort>.MaxValue >> 1,
72                      ↪ Bit<ushort>.ShiftRight(NumericType<ushort>.MaxValue, 1));
73      }
74
75      /// <summary>
76      /// <para>
77      /// Tests that u int 32 bitwise operations test.
78      /// </para>
79      /// <para></para>
80      /// </summary>
81      [Fact]
82      public static void UInt32BitwiseOperationsTest()
83      {
84          Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
85          Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
86          Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
87          Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
88          Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));

```

```

87     Assert.Equal(NumericType<uint>.MaxValue >> 1,
88         ↪ Bit<uint>.ShiftRight(NumericType<uint>.MaxValue, 1));
89 }
90
91 /// <summary>
92 /// <para>
93 /// Tests that u int 64 bitwise operations test.
94 /// </para>
95 /// </summary>
96 [Fact]
97 public static void UInt64BitwiseOperationsTest()
98 {
99     Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
100    Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
101    Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
102    Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
103    Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
104    Assert.Equal(NumericType<ulong>.MaxValue >> 1,
105        ↪ Bit<ulong>.ShiftRight(NumericType<ulong>.MaxValue, 1));
106 }
107
108 /// <summary>
109 /// <para>
110 /// Tests that partial read write test.
111 /// </para>
112 /// </summary>
113 [Fact]
114 public static void PartialReadWriteTest()
115 {
116     {
117         uint firstValue = 1;
118         uint secondValue = 1543;
119
120         // Pack (join) two values at the same time
121         uint value = secondValue << 1 | firstValue;
122
123         uint unpackagedFirstValue = value & 1;
124         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
125
126         Assert.True(firstValue == unpackagedFirstValue);
127         Assert.True(secondValue == unpackagedSecondValue);
128
129         // Using universal functions:
130         Assert.True(PartialRead(value, 0, 1) == firstValue);
131         Assert.True(PartialRead(value, 1, -1) == secondValue);
132
133         firstValue = 0;
134         secondValue = 6892;
135
136         value = PartialWrite(value, firstValue, 0, 1);
137         value = PartialWrite(value, secondValue, 1, -1);
138
139         Assert.True(PartialRead(value, 0, 1) == firstValue);
140         Assert.True(PartialRead(value, 1, -1) == secondValue);
141     }
142
143     {
144         uint firstValue = 1;
145         uint secondValue = 1543;
146
147         // Pack (join) two values at the same time
148         uint value = secondValue << 1 | firstValue;
149
150         uint unpackagedFirstValue = value & 1;
151         uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
152
153         Assert.True(firstValue == unpackagedFirstValue);
154         Assert.True(secondValue == unpackagedSecondValue);
155
156         // Using universal functions:
157         Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
158         Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
159
160         firstValue = 0;
161         secondValue = 6892;
162
163         value = Bit.PartialWrite(value, firstValue, 0, 1);
164         value = Bit.PartialWrite(value, secondValue, 1, -1);

```

```

165
166     Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
167     Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
168 }
169
170 {
171     uint firstValue = 1;
172     uint secondValue = 1543;
173
174     // Pack (join) two values at the same time
175     uint value = secondValue << 1 | firstValue;
176
177     uint unpackagedFirstValue = value & 1;
178     uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
179
180     Assert.True(firstValue == unpackagedFirstValue);
181     Assert.True(secondValue == unpackagedSecondValue);
182
183     // Using universal functions:
184     var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
185     var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
186     var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
187     var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
188
189     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
190     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
191
192     firstValue = 0;
193     secondValue = 6892;
194
195     value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
196     value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
197
198     Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
199     Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
200 }
201
202
203 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
204 private static uint PartialWrite(uint target, uint source, int shift, int limit)
205 {
206     if (shift < 0)
207     {
208         shift = 32 + shift;
209     }
210     if (limit < 0)
211     {
212         limit = 32 + limit;
213     }
214     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
215     var targetMask = ~(sourceMask << shift);
216     return target & targetMask | (source & sourceMask) << shift;
217 }
218 private static uint PartialRead(uint target, int shift, int limit)
219 {
220     if (shift < 0)
221     {
222         shift = 32 + shift;
223     }
224     if (limit < 0)
225     {
226         limit = 32 + limit;
227     }
228     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
229     var targetMask = sourceMask << shift;
230     return (target & targetMask) >> shift;
231 }
232 private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
233 {
234     if (shift < 0)
235     {
236         shift = 32 + shift;
237     }
238     if (limit < 0)
239     {
240         limit = 32 + limit;
241     }
242     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
243     var targetMask = ~(sourceMask << shift);

```

```

244         return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
245     }
246     private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
247     {
248         if (shift < 0)
249         {
250             shift = 32 + shift;
251         }
252         if (limit < 0)
253         {
254             limit = 32 + limit;
255         }
256         var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
257         var targetMask = sourceMask << shift;
258         return new Tuple<uint, int>(targetMask, shift);
259     }
260     private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
261     private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
262     private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
263     private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
264
265     /// <summary>
266     /// <para>
267     /// Tests that bug with loading constant of 8 test.
268     /// </para>
269     /// <para></para>
270     /// </summary>
271     [Fact]
272     public static void BugWithLoadingConstantOf8Test()
273     {
274         Bit<byte>.PartialWrite(0, 1, 5, -5);
275     }
276 }
277 }

```

1.13 ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the math extensions tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public static class MathExtensionsTests
12     {
13         /// <summary>
14         /// <para>
15         /// Tests that abs test.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         [Fact]
20         public static void AbsTest()
21         {
22             var number = -1L;
23             var returnValue = number.Abs();
24             Assert.Equal(1L, returnValue);
25             Assert.Equal(1L, number);
26         }
27
28         /// <summary>
29         /// <para>
30         /// Tests that negate test.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         [Fact]
35         public static void NegateTest()
36         {
37             var number = 2L;

```

```

38         var returnValue = number.Negate();
39         Assert.Equal(-2L, returnValue);
40         Assert.Equal(-2L, number);
41     }
42
43     /// <summary>
44     /// <para>
45     /// Tests that unsigned negate test.
46     /// </para>
47     /// <para></para>
48     /// </summary>
49     [Fact]
50     public static void UnsignedNegateTest()
51     {
52         var number = 2UL;
53         var returnValue = number.Negate();
54         Assert.Equal(18446744073709551614, returnValue);
55         Assert.Equal(18446744073709551614, number);
56     }
57 }
58 }

```

1.14 ./csharp/Platform.Numbers.Tests/MathTests.cs

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the math tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public static class MathTests
12     {
13         /// <summary>
14         /// <para>
15         /// Tests that compiled operations test.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         [Fact]
20         public static void CompiledOperationsTest()
21         {
22             Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
23         }
24     }
25 }

```

1.15 ./csharp/Platform.Numbers.Tests/SystemTests.cs

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      /// <summary>
6      /// <para>
7      /// Represents the system tests.
8      /// </para>
9      /// <para></para>
10     /// </summary>
11     public static class SystemTests
12     {
13         /// <summary>
14         /// <para>
15         /// Tests that possible pack two values into one test.
16         /// </para>
17         /// <para></para>
18         /// </summary>
19         [Fact]
20         public static void PossiblePackTwoValuesIntoOneTest()
21         {
22             uint value = 0;
23
24             // Set one to first bit
25             value |= 1;
26
27             Assert.True(value == 1);
28         }
29     }
30 }

```

```

29 // Set zero to first bit
30 value &= 0xFFFFFFFF;
31
32 // Get first bit
33 uint read = value & 1;
34
35 Assert.True(read == 0);
36
37 uint firstValue = 1;
38 uint secondValue = 1543;
39
40 // Pack (join) two values at the same time
41 value = (secondValue << 1) | firstValue;
42
43 uint unpackagedFirstValue = value & 1;
44 uint unpackagedSecondValue = (value & 0xFFFFFFFF) >> 1;
45
46 Assert.True(firstValue == unpackagedFirstValue);
47 Assert.True(secondValue == unpackagedSecondValue);
48
49 // Using universal functions:
50
51 Assert.True(PartialRead(value, 0, 1) == firstValue);
52 Assert.True(PartialRead(value, 1, -1) == secondValue);
53
54 firstValue = 0;
55 secondValue = 6892;
56
57 value = PartialWrite(value, firstValue, 0, 1);
58 value = PartialWrite(value, secondValue, 1, -1);
59
60 Assert.True(PartialRead(value, 0, 1) == firstValue);
61 Assert.True(PartialRead(value, 1, -1) == secondValue);
62 }
63 private static uint PartialWrite(uint target, uint source, int shift, int limit)
64 {
65     if (shift < 0)
66     {
67         shift = 32 + shift;
68     }
69     if (limit < 0)
70     {
71         limit = 32 + limit;
72     }
73     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
74     var targetMask = ~(sourceMask << shift);
75     return (target & targetMask) | ((source & sourceMask) << shift);
76 }
77 private static uint PartialRead(uint target, int shift, int limit)
78 {
79     if (shift < 0)
80     {
81         shift = 32 + shift;
82     }
83     if (limit < 0)
84     {
85         limit = 32 + limit;
86     }
87     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
88     var targetMask = sourceMask << shift;
89     return (target & targetMask) >> shift;
90 }
91 }
92 }

```

Index

- ./csharp/Platform.Numbers.Tests/ArithmeticExtensionsTests.cs, 16
- ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs, 17
- ./csharp/Platform.Numbers.Tests/BitTests.cs, 17
- ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs, 21
- ./csharp/Platform.Numbers.Tests/MathTests.cs, 22
- ./csharp/Platform.Numbers.Tests/SystemTests.cs, 22
- ./csharp/Platform.Numbers/Arithmetic.cs, 1
- ./csharp/Platform.Numbers/ArithmeticExtensions.cs, 3
- ./csharp/Platform.Numbers/Arithmetic[T].cs, 4
- ./csharp/Platform.Numbers/Bit.cs, 5
- ./csharp/Platform.Numbers/BitExtensions.cs, 8
- ./csharp/Platform.Numbers/Bit[T].cs, 10
- ./csharp/Platform.Numbers/Math.cs, 13
- ./csharp/Platform.Numbers/MathExtensions.cs, 15
- ./csharp/Platform.Numbers/Math[T].cs, 16