

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 by one.</para>
107    /// <para>Увеличивает число на единицу.</para>
108    /// </summary>
109    /// <typeparam name="T">
110    /// <para>The number's 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.</para>
119    /// <para>Увеличенное на единицу число.</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 by one.</para>
126    /// <para>Увеличивает значение аргумента на единицу.</para>
127    /// </summary>
128    /// <typeparam name="T">
129    /// <para>The number's 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 value.</para>
138    /// <para>Увеличенное значение аргумента.</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>Decreases number by one.</para>
145    /// <para>Уменьшение числа на единицу.</para>
146    /// </summary>
147    /// <typeparam name="T">
148    /// <para>The number's 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.</para>
157     /// <para>Уменьшенное на единицу число.</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 by one.</para>
164     /// <para>Уменьшает значение аргумента на единицу.</para>
165     /// </summary>
166     /// <typeparam name="T">
167     /// <para>The number's 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 value.</para>
176     /// <para>Уменьшенное значение аргумента.</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     /// <summary>
8     /// <para>Provides a set of extension methods that perform arithmetic operations on
9     ↪ arbitrary object types.</para>
10    /// <para>Предоставляет набор методов расширения выполняющих арифметические операции для
11    ↪ объектов произвольного типа.</para>
12    /// </summary>
13    public static class ArithmeticExtensions
14    {
15        /// <summary>
16        /// <para>Increments the variable passed as an argument by one.</para>
17        /// <para>Увеличивает переданную в качестве аргумента переменную на единицу.</para>
18        /// </summary>
19        /// <typeparam name="T">
20        /// <para>The number's type.</para>
21        /// <para>Тип числа.</para>
22        /// </typeparam>
23        /// <param name="x">
24        /// <para>The reference to the incremented variable.</para>
25        /// <para>Ссылка на увеличиваемую переменную.</para>
26        /// </param>
27        /// <returns>
28        /// <para>The value of the argument incremented by one.</para>
29        /// <para>Увеличенное значение аргумента на единицу.</para>
30        /// </returns>
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public static T Increment<T>(this ref T x) where T : struct => x =
33        ↪ Arithmetic<T>.Increment(x);
34
35        /// <summary>
36        /// <para>Decrements the variable passed as an argument by one.</para>
37        /// <para>Уменьшает переданную в качестве аргумента переменную на единицу.</para>
38        /// </summary>
39        /// <typeparam name="T">
40        /// <para>The number's type.</para>
41        /// <para>Тип числа.</para>
42        /// </typeparam>
43        /// <param name="x">
44        /// <para>The reference to the decremented variable.</para>
45        /// <para>Ссылка на уменьшаемую переменную.</para>
46        /// </param>
47        /// <returns>
48        /// <para>The value of the argument decremented by one.</para>
49        /// <para>Уменьшенное значение аргумента на единицу.</para>
50        /// </returns>
51        [MethodImpl(MethodImplOptions.AggressiveInlining)]
52        public static T Decrement<T>(this ref T x) where T : struct => x =
53        ↪ Arithmetic<T>.Decrement(x);
54    }
55 }

```

```

47     /// </returns>
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static T Decrement<T>(this ref T x) where T : struct => x =
        ↪ Arithmetic<T>.Decrement(x);
50 }
51 }

```

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     /// <summary>
13     /// <para>.</para>
14     /// <para>.</para>
15     /// </summary>
16     public static class Arithmetic<T>
17     {
18         /// <summary>
19         /// <para>.</para>
20         /// <para>.</para>
21         /// </summary>
22         public static readonly Func<T, T, T> Add = CompileAddDelegate();
23
24         /// <summary>
25         /// <para>.</para>
26         /// <para>.</para>
27         /// </summary>
28         public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
29
30         /// <summary>
31         /// <para>.</para>
32         /// <para>.</para>
33         /// </summary>
34         public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
35
36         /// <summary>
37         /// <para>.</para>
38         /// <para>.</para>
39         /// </summary>
40         public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
41
42         /// <summary>
43         /// <para>.</para>
44         /// <para>.</para>
45         /// </summary>
46         public static readonly Func<T, T> Increment = CompileIncrementDelegate();
47
48         /// <summary>
49         /// <para>.</para>
50         /// <para>.</para>
51         /// </summary>
52         public static readonly Func<T, T> Decrement = CompileDecrementDelegate();
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         private static Func<T, T, T> CompileAddDelegate()
56         {
57             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
58             {
59                 Ensure.Always.IsNumeric<T>();
60                 emitter.LoadArguments(0, 1);
61                 emitter.Add();
62                 emitter.Return();
63             });
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         private static Func<T, T, T> CompileSubtractDelegate()
68         {
69             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
70             {
71                 Ensure.Always.IsNumeric<T>();

```

```

72         emitter.LoadArguments(0, 1);
73         emitter.Subtract();
74         emitter.Return();
75     });
76 }
77
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 private static Func<T, T, T> CompileMultiplyDelegate()
80 {
81     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
82     {
83         Ensure.Always.IsNumeric<T>();
84         emitter.LoadArguments(0, 1);
85         emitter.Emit(OpCodes.Mul);
86         emitter.Return();
87     });
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static Func<T, T, T> CompileDivideDelegate()
92 {
93     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
94     {
95         Ensure.Always.IsNumeric<T>();
96         emitter.LoadArguments(0, 1);
97         if(NumericType<T>.IsSigned)
98         {
99             emitter.Emit(OpCodes.Div);
100         }
101         else
102         {
103             emitter.Emit(OpCodes.Div_Un);
104         }
105         emitter.Return();
106     });
107 }
108
109 [MethodImpl(MethodImplOptions.AggressiveInlining)]
110 private static Func<T, T> CompileIncrementDelegate()
111 {
112     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
113     {
114         Ensure.Always.IsNumeric<T>();
115         emitter.LoadArgument(0);
116         emitter.Increment<T>();
117         emitter.Return();
118     });
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 private static Func<T, T> CompileDecrementDelegate()
123 {
124     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
125     {
126         Ensure.Always.IsNumeric<T>();
127         emitter.LoadArgument(0);
128         emitter.Decrement<T>();
129         emitter.Return();
130     });
131 }
132 }
133 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Numbers
6 {
7     /// <summary>
8     /// <para>A set of operations on the set bits of a number.</para>
9     /// <para>Набор операций над установленными битами числа.</para>
10    /// </summary>
11    public static class Bit
12    {
13        /// <summary>
14        /// <para>Counts the number of bits set in a number.</para>
15        /// <para>Подсчитывает количество установленных бит в числе.</para>

```

```

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

```

```

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

```

```

165 /// <param name="y">
166 /// <para>The number of bits to shift.</para>
167 /// <para>Количество бит на которые выполнить смещение.</para>
168 /// </param>
169 /// <returns>
170 /// <para>The value with discarded low-order bits.</para>
171 /// <para>Значение с отброшенными младшими битами.</para>
172 /// </returns>
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
175
176 /// <summary>
177 /// <para>Performs a partial write of a specified number of bits from source number to
178   ↳ target number.</para>
179 /// <para>Выполняет частичную запись определенного количества бит исходного числа в
180   ↳ целевое число.</para>
181 /// </summary>
182 /// <typeparam name="T">
183 /// <para>The numbers' type.</para>
184 /// <para>Тип чисел.</para>
185 /// </typeparam>
186 /// <param name="target">
187 /// <para>The value to which the partial write will be performed.</para>
188 /// <para>Значение в которое будет выполнена частичная запись.</para>
189 /// </param>
190 /// <param name="source">
191 /// <para>Data source for recording.</para>
192 /// <para>Источник данных для записи.</para>
193 /// </param>
194 /// <param name="shift">
195 /// <para>The start position to read from.</para>
196 /// <para>Стартовая позиция чтения.</para>
197 /// </param>
198 /// <param name="limit">
199 /// <para>The number of bits to write from source to target.</para>
200 /// <para>Количество бит, которые нужно записать из source в target.</para>
201 /// </param>
202 /// <returns>
203 /// <para>The target number updated with bits from source number.</para>
204 /// <para>Целевое число с обновленными битами из исходного числа.</para>
205 /// </returns>
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
208   ↳ 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's 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) =>
236   ↳ Bit<T>.PartialRead(target, shift, limit);
237 }
238 }

```

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     /// <para>Represents a set of bitwise operation.</para>
9     /// <para>Представляет набор битовых операций.</para>
10    /// </summary>
11    public static class BitwiseExtensions
12    {
13        /// <summary>
14        /// <para>Performs bitwise inversion of a number.</para>
15        /// <para>Выполняет побитовую инверсию 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="target">
22        /// <para>The number to invert.</para>
23        /// <para>Число для инверсии.</para>
24        /// </param>
25        /// <returns>
26        /// <para>An inverted value of the number.</para>
27        /// <para>Обратное значение числа.</para>
28        /// </returns>
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);
31
32        /// <summary>
33        /// <para>Performs a partial write of a specified number of bits from source number to
34        /// ↪ target number.</para>
35        /// <para>Выполняет частичную запись определенного количества бит исходного числа в
36        /// ↪ целевое число.</para>
37        /// </summary>
38        /// <typeparam name="T">
39        /// <para>The numbers' type.</para>
40        /// <para>Тип чисел.</para>
41        /// </typeparam>
42        /// <param name="target">
43        /// <para>The value to which the partial write will be performed.</para>
44        /// <para>Значение в которое будет выполнена частичная запись.</para>
45        /// </param>
46        /// <param name="source">
47        /// <para>Data source for writing.</para>
48        /// <para>Источник данных для записи.</para>
49        /// </param>
50        /// <param name="shift">
51        /// <para>The start position to read from.</para>
52        /// <para>Стартовая позиция чтения.</para>
53        /// </param>
54        /// <param name="limit">
55        /// <para>The number of bits to write from source to target.</para>
56        /// <para>Количество бит, которые нужно записать из source в target.</para>
57        /// </param>
58        /// <returns>
59        /// <para>The target number updated with bits from source number.</para>
60        /// <para>Целевое число с обновленными битами из исходного числа.</para>
61        /// </returns>
62        [MethodImpl(MethodImplOptions.AggressiveInlining)]
63        public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
64        /// ↪ T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);
65
66        /// <summary>
67        /// <para>Reads a specified number of bits from the number at specified position.</para>
68        /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
69        /// </summary>
70        /// <typeparam name="T">
71        /// <para>The number's type.</para>
72        /// <para>Тип числа.</para>
73        /// </typeparam>
74        /// <param name="target">
75        /// <para>The number from which the partial read will be performed.</para>
76        /// <para>Число из которого будет выполнено частичное чтение.</para>
77        /// </param>
78        /// <param name="shift">
79        /// <para>The start position to read from.</para>
80        /// <para>Стартовая позиция чтения.</para>
81        /// </param>
82        /// <param name="limit">
83        /// <para>The number of bits to read from the number.</para>
84        /// <para>Количество бит, которые нужно прочитать из числа.</para>
85        /// </param>
86        /// <returns>
87        /// <para>The number of bits read from the number.</para>
88        /// <para>Число, полученное в результате чтения.</para>
89        /// </returns>
90        [MethodImpl(MethodImplOptions.AggressiveInlining)]
91        public static T PartialRead<T>(this ref T target, int shift, int limit) where
92        /// ↪ T : struct => target = Bit<T>.PartialRead(target, shift, limit);
93    }
94 }

```

```

78     /// </param>
79     /// <param name="limit">
80     /// <para>The number of bits to read.</para>
81     /// <para>Количество бит, которые нужно считать.</para>
82     /// </param>
83     /// <returns>
84     /// <para>The number consisting of bits read from the source number.</para>
85     /// <para>Число состоящее из считанных из исходного числа бит.</para>
86     /// </returns>
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static T PartialRead<T>(this T target, int shift, int limit) =>
89         ↪ Bit<T>.PartialRead(target, shift, limit);
90 }

```

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     /// <summary>
12     /// <para>.</para>
13     /// <para>.</para>
14     /// </summary>
15     public static class Bit<T>
16     {
17         /// <summary>
18         /// <para>.</para>
19         /// <para>.</para>
20         /// </summary>
21         public static readonly Func<T, T> Not = CompileNotDelegate();
22
23         /// <summary>
24         /// <para>.</para>
25         /// <para>.</para>
26         /// </summary>
27         public static readonly Func<T, T, T> Or = CompileOrDelegate();
28
29         /// <summary>
30         /// <para>.</para>
31         /// <para>.</para>
32         /// </summary>
33         public static readonly Func<T, T, T> And = CompileAndDelegate();
34
35         /// <summary>
36         /// <para>.</para>
37         /// <para>.</para>
38         /// </summary>
39         public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();
40
41         /// <summary>
42         /// <para>.</para>
43         /// <para>.</para>
44         /// </summary>
45         public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();
46
47         /// <summary>
48         /// <para>.</para>
49         /// <para>.</para>
50         /// </summary>
51         public static readonly Func<T, T, int, int, T> PartialWrite =
52             ↪ CompilePartialWriteDelegate();
53
54         /// <summary>
55         /// <para>.</para>
56         /// <para>.</para>
57         /// </summary>
58         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         private static Func<T, T> CompileNotDelegate()
62         {
63             return DelegateHelpers.Compile<Func<T, T>>(emitter =>

```

```

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

```

```

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

```

```

219         emitter.And();
220         emitter.StoreLocal(sourceMask);
221         emitter.LoadLocal(sourceMask);
222         emitter.LoadArgument(shiftArgument);
223         emitter.ShiftLeft();
224         emitter.StoreLocal(targetMask);
225         emitter.LoadArgument(0); // target
226         emitter.LoadLocal(targetMask);
227         emitter.And();
228         emitter.LoadArgument(shiftArgument);
229         emitter.ShiftRight();
230         emitter.Return();
231     });
232 }
233
234 [MethodImpl(MethodImplOptions.AggressiveInlining)]
235 private static Tuple<int, T> GetConstants()
236 {
237     var type = typeof(T);
238     if (type == typeof(ulong))
239     {
240         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
241     }
242     if (type == typeof(uint))
243     {
244         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
245     }
246     if (type == typeof(ushort))
247     {
248         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
249     }
250     if (type == typeof(byte))
251     {
252         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
253     }
254     throw new NotSupportedException();
255 }
256 }
257 }

```

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      /// <summary>
9      /// <para>Represents a collection of algebraic methods.</para>
10     /// <para>Представляет набор алгебраических методов.</para>
11     /// </summary>
12     /// <remarks>Resizable array (FileMappedMemory) for values cache may be used. or cached
13     ↪ oeis.org</remarks>
14     public static class Math
15     {
16         /// <remarks>
17         /// <para>Source: https://oeis.org/A000142/list </para>
18         /// <para>Источник: https://oeis.org/A000142/list </para>
19         /// </remarks>
20         private static readonly ulong[] _factorials =
21         {
22             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
23             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
24             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
25         };
26
27         /// <remarks>
28         /// <para>Source: https://oeis.org/A000108/list </para>
29         /// <para>Источник: https://oeis.org/A000108/list </para>
30         /// </remarks>
31         private static readonly ulong[] _catalans =
32         {
33             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
34             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
35             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
36             18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304,
37             14544636039226909, 55534064877048198, 212336130412243110, 812944042149730764,
38             ↪ 3116285494907301262, 11959798385860453492
39         }
40     }
41 }

```

```

37 };
38
39 /// <summary>
40 /// <para>Represents the limit for calculating the catanal number, supported by the <see
  ↳ cref="ulong"/> type.</para>
41 /// <para>Представляет предел расчёта катаналового числа, поддерживаемый <see
  ↳ cref="ulong"/> типом.</para>
42 /// </summary>
43 public static readonly ulong MaximumFactorialNumber = 20;
44
45 /// <summary>
46 /// <para>Represents the limit for calculating the factorial number, supported by the
  ↳ <see cref="ulong"/> type.</para>
47 /// <para>Представляет предел расчёта факториала числа, поддерживаемый <see
  ↳ cref="ulong"/> типом.</para>
48 /// </summary>
49 public static readonly ulong MaximumCatalanIndex = 36;
50
51 /// <summary>
52 /// <para>Returns the product of all positive integers less than or equal to the number
  ↳ specified as an argument.</para>
53 /// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
  ↳ в качестве аргумента числу.</para>
54 /// </summary>
55 /// <param name="n">
56 /// <para>The maximum positive number that will participate in factorial's
  ↳ product.</para>
57 /// <para>Максимальное положительное число, которое будет участвовать в произведении
  ↳ факториала.</para>
58 /// </param>
59 /// <returns>
60 /// <para>The product of all positive integers less than or equal to the number
  ↳ specified as an argument.</para>
61 /// <para>Произведение всех положительных чисел меньше или равных указанному, в качестве
  ↳ аргумента, числу.</para>
62 /// </returns>
63 public static ulong Factorial(ulong n)
64 {
65     if (n >= 0 && n <= MaximumFactorialNumber)
66     {
67         return _factorials[n];
68     }
69     else
70     {
71         throw new ArgumentOutOfRangeException($"Only numbers from 0 to
  ↳ {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
  ↳ length.");
72     }
73 }
74
75 /// <summary>
76 /// <para>Returns the Catalan Number with the number specified as an argument.</para>
77 /// <para>Возвращает Число Катанала с номером, указанным в качестве аргумента.</para>
78 /// </summary>
79 /// <param name="n">
80 /// <para>The number of the Catalan number.</para>
81 /// <para>Номер Числа Катанала.</para>
82 /// </param>
83 /// <returns>
84 /// <para>The Catalan Number with the number specified as an argument.</para>
85 /// <para>Число Катанала с номером, указанным в качестве аргумента.</para>
86 /// </returns>
87 public static ulong Catalan(ulong n)
88 {
89     if (n >= 0 && n <= MaximumCatalanIndex)
90     {
91         return _catalans[n];
92     }
93     else
94     {
95         throw new ArgumentOutOfRangeException($"Only numbers from 0 to
  ↳ {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
  ↳ length.");
96     }
97 }
98
99 /// <summary>

```

```

100     /// <para>Checks if a number is a power of two.</para>
101     /// <para>Проверяет, является ли число степенью двойки.</para>
102     /// </summary>
103     /// <param name="x">
104     /// <para>The number to check.</para>
105     /// <para>Число для проверки.</para>
106     /// </param>
107     /// <returns>
108     /// <para>True if the number is a power of two otherwise false.</para>
109     /// <para>True, если число является степенью двойки, иначе - false.</para>
110     /// </returns>
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
113
114     /// <summary>
115     /// <para>Takes a module from a number.</para>
116     /// <para>Берёт модуль от числа.</para>
117     /// </summary>
118     /// <typeparam name="T">
119     /// <para>The number's type.</para>
120     /// <para>Тип числа.</para>
121     /// </typeparam>
122     /// <param name="x">
123     /// <para>The number from which to take the absolute value.</para>
124     /// <para>Число, от которого необходимо взять абсолютное значение.</para>
125     /// </param>
126     /// <returns>
127     /// <para>The absolute value of the number.</para>
128     /// <para>Абсолютное значение числа.</para>
129     /// </returns>
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static T Abs<T>(T x) => Math<T>.Abs(x);
132
133     /// <summary>
134     /// <para>Makes a number negative.</para>
135     /// <para>Делает число отрицательным.</para>
136     /// </summary>
137     /// <typeparam name="T">
138     /// <para>The number's type.</para>
139     /// <para>Тип числа.</para>
140     /// </typeparam>
141     /// <param name="x">
142     /// <para>The number to be made negative.</para>
143     /// <para>Число которое нужно сделать отрицательным.</para>
144     /// </param>
145     /// <returns>
146     /// <para>A negative number.</para>
147     /// <para>Отрицательное число.</para>
148     /// </returns>
149     [MethodImpl(MethodImplOptions.AggressiveInlining)]
150     public static T Negate<T>(T x) => Math<T>.Negate(x);
151 }
152 }

```

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
9      /// ↳ arbitrary object types.</para>
10     /// <para>Предоставляет набор методов расширения выполняющих математические операции для
11     /// ↳ объектов произвольного типа.</para>
12     /// </summary>
13     public static class MathExtensions
14     {
15         /// <summary>
16         /// <para>Takes a module from a number.</para>
17         /// <para>Берёт модуль от числа.</para>
18         /// </summary>
19         /// <typeparam name="T">
20         /// <para>The number's type.</para>
21         /// <para>Тип числа.</para>
22         /// </typeparam>
23         /// <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>
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     /// <summary>
12     /// <para>.</para>
13     /// <para>.</para>
14     /// </summary>
15     public static class Math<T>
16     {
17         /// <summary>
18         /// <para>.</para>
19         /// <para>.</para>
20         /// </summary>
21         public static readonly Func<T, T> Abs = CompileAbsDelegate();
22
23         /// <summary>
24         /// <para>.</para>
25         /// <para>.</para>
26         /// </summary>
27         public static readonly Func<T, T> Negate = CompileNegateDelegate();
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         private static Func<T, T> CompileAbsDelegate()
31         {
32             return DelegateHelpers.Compile<Func<T, T>>(emitter =>
33             {
34                 Ensure.Always.IsNumeric<T>();
35                 emitter.LoadArgument(0);
36                 if (NumericType<T>.IsSigned)
37                 {
38                     emitter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
39                 }
40                 emitter.Return();
41             });
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private static Func<T, T> CompileNegateDelegate()
46         {
47             return DelegateHelpers.Compile<Func<T, T>>(emitter =>

```



```

48         {
49             emitter.LoadArgument(0);
50             emitter.Negate();
51             emitter.Return();
52         });
53     }
54 }
55 }

```

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

```

```

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

```

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

```

1 using Xunit;
2
3 namespace Platform.Numbers.Tests
4 {
5     public static class MathExtensionsTests
6     {
7         [Fact]
8         public static void AbsTest()
9         {
10             var number = -1L;
11             var returnValue = number.Abs();
12             Assert.Equal(1L, returnValue);
13             Assert.Equal(1L, number);
14         }
15
16         [Fact]
17         public static void NegateTest()
18         {
19             var number = 2L;
20             var returnValue = number.Negate();
21             Assert.Equal(-2L, returnValue);
22             Assert.Equal(-2L, number);
23         }
24
25         [Fact]
26         public static void UnsignedNegateTest()
27         {
28             var number = 2UL;

```

```

29         var returnValue = number.Negate();
30         Assert.Equal(18446744073709551614, returnValue);
31         Assert.Equal(18446744073709551614, number);
32     }
33 }
34 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class MathTests
6      {
7          [Fact]
8          public static void CompiledOperationsTest()
9          {
10             Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
11         }
12     }
13 }

```

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

```

1  using Xunit;
2
3  namespace Platform.Numbers.Tests
4  {
5      public static class SystemTests
6      {
7          [Fact]
8          public static void PossiblePackTwoValuesIntoOneTest()
9          {
10             uint value = 0;
11
12             // Set one to first bit
13             value |= 1;
14
15             Assert.True(value == 1);
16
17             // Set zero to first bit
18             value &= 0xFFFFFFFF;
19
20             // Get first bit
21             uint read = value & 1;
22
23             Assert.True(read == 0);
24
25             uint firstValue = 1;
26             uint secondValue = 1543;
27
28             // Pack (join) two values at the same time
29             value = (secondValue << 1) | firstValue;
30
31             uint unpackagedFirstValue = value & 1;
32             uint unpackagedSecondValue = (value & 0xFFFFFFFF) >> 1;
33
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         }
59     }
60 }

```

```

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     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, 17
- ./csharp/Platform.Numbers.Tests/ArithmeticTests.cs, 17
- ./csharp/Platform.Numbers.Tests/BitTests.cs, 17
- ./csharp/Platform.Numbers.Tests/MathExtensionsTests.cs, 20
- ./csharp/Platform.Numbers.Tests/MathTests.cs, 21
- ./csharp/Platform.Numbers.Tests/SystemTests.cs, 21
- ./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