

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
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         private static Func<T, T, T> CompileAddDelegate()
54         {
55             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
56             {
57                 Ensure.Always.IsNumeric<T>();
58                 emitter.LoadArguments(0, 1);
59                 emitter.Add();
60                 emitter.Return();
61             });
62         }
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         private static Func<T, T, T> CompileSubtractDelegate()
66         {
67             return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
68             {
69                 Ensure.Always.IsNumeric<T>();
70                 emitter.LoadArguments(0, 1);
71                 emitter.Subtract();
72                 emitter.Return();
73             });
74         }
75     }
```

```

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

```

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>

```

```

170     /// </returns>
171     [MethodImpl(MethodImplOptions.AggressiveInlining)]
172     public static T ShiftRight<T>(T x, int y) => Bit<T>.ShiftRight(x, y);
173
174     /// <summary>
175     /// <para>Performs a partial write of a specified number of bits from source number to
176     ///     target number.</para>
177     /// <para>Выполняет частичную запись определенного количества бит исходного числа в
178     ///     целевое число.</para>
179     /// </summary>
180     /// <typeparam name="T">
181     /// <para>The numbers' type.</para>
182     /// <para>Тип чисел.</para>
183     /// </typeparam>
184     /// <param name="target">
185     /// <para>The value to which the partial write will be performed.</para>
186     /// <para>Значение в которое будет выполнена частичная запись.</para>
187     /// </param>
188     /// <param name="source">
189     /// <para>Data source for recording.</para>
190     /// <para>Источник данных для записи.</para>
191     /// </param>
192     /// <param name="shift">
193     /// <para>The start position to read from.</para>
194     /// <para>Стартовая позиция чтения.</para>
195     /// </param>
196     /// <param name="limit">
197     /// <para>The number of bits to write from source to target.</para>
198     /// <para>Количество бит, которые нужно записать из source в target.</para>
199     /// </param>
200     /// <returns>
201     /// <para>The target number updated with bits from source number.</para>
202     /// <para>Целевое число с обновленными битами из исходного числа.</para>
203     /// </returns>
204     [MethodImpl(MethodImplOptions.AggressiveInlining)]
205     public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
206     /// <summary>
207     /// <para>Reads a specified number of bits from the number at specified position.</para>
208     /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
209     /// </summary>
210     /// <typeparam name="T">
211     /// <para>The number's type.</para>
212     /// <para>Тип числа.</para>
213     /// </typeparam>
214     /// <param name="target">
215     /// <para>The number from which the partial read will be performed.</para>
216     /// <para>Число из которого будет выполнено частичное чтение.</para>
217     /// </param>
218     /// <param name="shift">
219     /// <para>The start position to read from.</para>
220     /// <para>Стартовая позиция чтения.</para>
221     /// </param>
222     /// <param name="limit">
223     /// <para>The number of bits to read.</para>
224     /// <para>Количество бит, которые нужно считать.</para>
225     /// </param>
226     /// <returns>
227     /// <para>The number consisting of bits read from the source number.</para>
228     /// <para>Число состоящее из считанных из исходного числа бит.</para>
229     /// </returns>
230     [MethodImpl(MethodImplOptions.AggressiveInlining)]
231     public static T PartialRead<T>(T target, int shift, int limit) =>
232     Bit<T>.PartialRead(target, shift, limit);
233 }

```

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>

```

```

84     /// </returns>
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static T PartialRead<T>(this T target, int shift, int limit) =>
87         ↪ Bit<T>.PartialRead(target, shift, limit);
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 }

```

```

58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 private static Func<T, T> CompileNotDelegate()
60 {
61     return DelegateHelpers.Compile<Func<T, T>>(emitter =>
62     {
63         Ensure.Always.IsNumeric<T>();
64         emitter.LoadArguments(0);
65         emitter.Not();
66         emitter.Return();
67     });
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 private static Func<T, T, T> CompileOrDelegate()
72 {
73     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
74     {
75         Ensure.Always.IsNumeric<T>();
76         emitter.LoadArguments(0, 1);
77         emitter.Or();
78         emitter.Return();
79     });
80 }
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 private static Func<T, T, T> CompileAndDelegate()
84 {
85     return DelegateHelpers.Compile<Func<T, T, T>>(emitter =>
86     {
87         Ensure.Always.IsNumeric<T>();
88         emitter.LoadArguments(0, 1);
89         emitter.And();
90         emitter.Return();
91     });
92 }
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 private static Func<T, int, T> CompileShiftLeftDelegate()
96 {
97     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
98     {
99         Ensure.Always.IsNumeric<T>();
100         emitter.LoadArguments(0, 1);
101         emitter.ShiftLeft();
102         emitter.Return();
103     });
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 private static Func<T, int, T> CompileShiftRightDelegate()
108 {
109     return DelegateHelpers.Compile<Func<T, int, T>>(emitter =>
110     {
111         Ensure.Always.IsNumeric<T>();
112         emitter.LoadArguments(0, 1);
113         emitter.ShiftRight<T>();
114         emitter.Return();
115     });
116 }
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
120 {
121     return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emitter =>
122     {
123         Ensure.Always.IsNumeric<T>();
124         var constants = GetConstants();
125         var bitsNumber = constants.Item1;
126         var numberFilledWithOnes = constants.Item2;
127         ushort shiftArgument = 2;
128         ushort limitArgument = 3;
129         var checkLimit = emitter.DefineLabel();
130         var calculateSourceMask = emitter.DefineLabel();
131         // Check shift
132         emitter.LoadArgument(shiftArgument);
133         emitter.LoadConstant(0);
134         emitter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
135         // Fix shift

```

```

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

```

```

214         emitter.LoadArgument(limitArgument); // limit
215         emitter.ShiftLeft();
216         emitter.Not();
217         emitter.LoadConstant(typeof(T), numberFilledWithOnes);
218         emitter.And();
219         emitter.StoreLocal(sourceMask);
220         emitter.LoadLocal(sourceMask);
221         emitter.LoadArgument(shiftArgument);
222         emitter.ShiftLeft();
223         emitter.StoreLocal(targetMask);
224         emitter.LoadArgument(0); // target
225         emitter.LoadLocal(targetMask);
226         emitter.And();
227         emitter.LoadArgument(shiftArgument);
228         emitter.ShiftRight<T>();
229         emitter.Return();
230     });
231 }
232
233 [MethodImpl(MethodImplOptions.AggressiveInlining)]
234 private static Tuple<int, T> GetConstants()
235 {
236     var type = typeof(T);
237     if (type == typeof(ulong))
238     {
239         return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
240     }
241     if (type == typeof(uint))
242     {
243         return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
244     }
245     if (type == typeof(ushort))
246     {
247         return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
248     }
249     if (type == typeof(byte))
250     {
251         return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
252     }
253     throw new NotSupportedException();
254 }
255 }
256

```

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         /// <remarks>
15         /// <para>Source: https://oeis.org/A000142/list </para>
16         /// <para>Источник: https://oeis.org/A000142/list </para>
17         /// </remarks>
18         private static readonly ulong[] _factorials =
19         {
20             1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
21             479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
22             355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
23         };
24
25         /// <remarks>
26         /// <para>Source: https://oeis.org/A000108/list </para>
27         /// <para>Источник: https://oeis.org/A000108/list </para>
28         /// </remarks>
29         private static readonly ulong[] _catalans =
30         {
31             1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
32             742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
33             6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
34             18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304,

```

14544636039226909, 55534064877048198, 212336130412243110, 812944042149730764,
→ 3116285494907301262, 11959798385860453492

};

```
/// <summary>
/// <para>Represents the limit for calculating the catanal number, supported by the <see
→ cref="ulong"/> type.</para>
/// <para>Представляет предел расчёта катаналового числа, поддерживаемый <see
→ cref="ulong"/> типом.</para>
/// </summary>
public static readonly ulong MaximumFactorialNumber = 20;
```

```
/// <summary>
/// <para>Represents the limit for calculating the factorial number, supported by the
→ <see cref="ulong"/> type.</para>
/// <para>Представляет предел расчёта факториала числа, поддерживаемый <see
→ cref="ulong"/> типом.</para>
/// </summary>
public static readonly ulong MaximumCatalanIndex = 36;
```

```
/// <summary>
/// <para>Returns the product of all positive integers less than or equal to the number
→ specified as an argument.</para>
/// <para>Возвращает произведение всех положительных чисел меньше или равных указанному
→ в качестве аргумента числу.</para>
/// </summary>
/// <param name="n">
/// <para>The maximum positive number that will participate in factorial's
→ product.</para>
/// <para>Максимальное положительное число, которое будет участвовать в произведении
→ факториала.</para>
/// </param>
/// <returns>
/// <para>The product of all positive integers less than or equal to the number
→ specified as an argument.</para>
/// <para>Произведение всех положительных чисел меньше или равных указанному, в качестве
→ аргумента, числу.</para>
/// </returns>
```

```
public static ulong Factorial(ulong n)
{
    if (n >= 0 && n <= MaximumFactorialNumber)
    {
        return _factorials[n];
    }
    else
    {
        throw new ArgumentOutOfRangeException($"Only numbers from 0 to
→ {MaximumFactorialNumber} are supported by unsigned integer with 64 bits
→ length.");
    }
}
```

```
/// <summary>
/// <para>Returns the Catalan Number with the number specified as an argument.</para>
/// <para>Возвращает Число Катанала с номером, указанным в качестве аргумента.</para>
/// </summary>
/// <param name="n">
/// <para>The number of the Catalan number.</para>
/// <para>Номер Числа Катанала.</para>
/// </param>
/// <returns>
/// <para>The Catalan Number with the number specified as an argument.</para>
/// <para>Число Катанала с номером, указанным в качестве аргумента.</para>
/// </returns>
```

```
public static ulong Catalan(ulong n)
{
    if (n >= 0 && n <= MaximumCatalanIndex)
    {
        return _catalans[n];
    }
    else
    {
        throw new ArgumentOutOfRangeException($"Only numbers from 0 to
→ {MaximumCatalanIndex} are supported by unsigned integer with 64 bits
→ length.");
    }
}
```

```

96
97     /// <summary>
98     /// <para>Checks if a number is a power of two.</para>
99     /// <para>Проверяет, является ли число степенью двойки.</para>
100    /// </summary>
101    /// <param name="x">
102    /// <para>The number to check.</para>
103    /// <para>Число для проверки.</para>
104    /// </param>
105    /// <returns>
106    /// <para>True if the number is a power of two otherwise false.</para>
107    /// <para>True, если число является степенью двойки, иначе - false.</para>
108    /// </returns>
109    [MethodImpl(MethodImplOptions.AggressiveInlining)]
110    public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;
111
112    /// <summary>
113    /// <para>Takes a module from a number.</para>
114    /// <para>Берёт модуль от числа.</para>
115    /// </summary>
116    /// <typeparam name="T">
117    /// <para>The number's type.</para>
118    /// <para>Тип числа.</para>
119    /// </typeparam>
120    /// <param name="x">
121    /// <para>The number from which to take the absolute value.</para>
122    /// <para>Число, от которого необходимо взять абсолютное значение.</para>
123    /// </param>
124    /// <returns>
125    /// <para>The absolute value of the number.</para>
126    /// <para>Абсолютное значение числа.</para>
127    /// </returns>
128    [MethodImpl(MethodImplOptions.AggressiveInlining)]
129    public static T Abs<T>(T x) => Math<T>.Abs(x);
130
131    /// <summary>
132    /// <para>Makes a number negative.</para>
133    /// <para>Делает число отрицательным.</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 to be made negative.</para>
141    /// <para>Число которое нужно сделать отрицательным.</para>
142    /// </param>
143    /// <returns>
144    /// <para>A negative number.</para>
145    /// <para>Отрицательное число.</para>
146    /// </returns>
147    [MethodImpl(MethodImplOptions.AggressiveInlining)]
148    public static T Negate<T>(T x) => Math<T>.Negate(x);
149 }
150 }

```

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">

```

```

20     /// <para>The number from which to take the absolute value.</para>
21     /// <para>Число от которого необходимо взять абсолютное значение.</para>
22     /// </param>
23     /// <returns>
24     /// <para>The absolute value of a number.</para>
25     /// <para>Абсолютное значение числа.</para>
26     /// </returns>
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
29
30     /// <summary>
31     /// <para>Makes a number negative.</para>
32     /// <para>Делает число отрицательным.</para>
33     /// </summary>
34     /// <typeparam name="T">
35     /// <para>The number's type.</para>
36     /// <para>Тип числа.</para>
37     /// </typeparam>
38     /// <param name="x">
39     /// <para>The number to be made negative.</para>
40     /// <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()

```



```

44     {
45         return DelegateHelpers.Compile<Func<T, T>>(emitter =>
46         {
47             emitter.LoadArgument(0);
48             emitter.Negate();
49             emitter.Return();
50         });
51     }
52 }
53 }

```

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

```

```

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

```

```

93     {
94         uint firstValue = 1;
95         uint secondValue = 1543;
96
97         // Pack (join) two values at the same time
98         uint value = secondValue << 1 | firstValue;
99
100        uint unpackagedFirstValue = value & 1;
101        uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
102
103        Assert.True(firstValue == unpackagedFirstValue);
104        Assert.True(secondValue == unpackagedSecondValue);
105
106        // Using universal functions:
107        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
108        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
109
110        firstValue = 0;
111        secondValue = 6892;
112
113        value = Bit.PartialWrite(value, firstValue, 0, 1);
114        value = Bit.PartialWrite(value, secondValue, 1, -1);
115
116        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
117        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
118    }
119
120    {
121        uint firstValue = 1;
122        uint secondValue = 1543;
123
124        // Pack (join) two values at the same time
125        uint value = secondValue << 1 | firstValue;
126
127        uint unpackagedFirstValue = value & 1;
128        uint unpackagedSecondValue = (value & 0xFFFFFFF0) >> 1;
129
130        Assert.True(firstValue == unpackagedFirstValue);
131        Assert.True(secondValue == unpackagedSecondValue);
132
133        // Using universal functions:
134        var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
135        var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
136        var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
137        var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);
138
139        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
140        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
141
142        firstValue = 0;
143        secondValue = 6892;
144
145        value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
146        value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
147
148        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
149        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
150    }
151 }
152
153 // TODO: Can be optimized using precalculation of TargetMask and SourceMask
154 private static uint PartialWrite(uint target, uint source, int shift, int limit)
155 {
156     if (shift < 0)
157     {
158         shift = 32 + shift;
159     }
160     if (limit < 0)
161     {
162         limit = 32 + limit;
163     }
164     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
165     var targetMask = ~(sourceMask << shift);
166     return target & targetMask | (source & sourceMask) << shift;
167 }
168
169 private static uint PartialRead(uint target, int shift, int limit)
170 {
171     if (shift < 0)
172     {

```

```

173         shift = 32 + shift;
174     }
175     if (limit < 0)
176     {
177         limit = 32 + limit;
178     }
179     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
180     var targetMask = sourceMask << shift;
181     return (target & targetMask) >> shift;
182 }
183
184 private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
185 {
186     if (shift < 0)
187     {
188         shift = 32 + shift;
189     }
190     if (limit < 0)
191     {
192         limit = 32 + limit;
193     }
194     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
195     var targetMask = ~(sourceMask << shift);
196     return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
197 }
198
199 private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
200 {
201     if (shift < 0)
202     {
203         shift = 32 + shift;
204     }
205     if (limit < 0)
206     {
207         limit = 32 + limit;
208     }
209     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
210     var targetMask = sourceMask << shift;
211     return new Tuple<uint, int>(targetMask, shift);
212 }
213
214 private static uint PartialWrite(uint target, uint targetMask, uint source, uint
    ↪ sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
215
216 private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
    ↪ masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
    ↪ masksAndShift.Item2, masksAndShift.Item3);
217
218 private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
    ↪ targetMask) >> shift;
219
220 private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
    ↪ PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);
221
222 [Fact]
223 public static void BugWithLoadingConstantOf8Test()
224 {
225     Bit<byte>.PartialWrite(0, 1, 5, -5);
226 }
227 }
228 }

```

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 unpackedFirstValue = value & 1;
32             uint unpackedSecondValue = (value & 0xFFFFFFFF) >> 1;
33
34             Assert.True(firstValue == unpackedFirstValue);
35             Assert.True(secondValue == unpackedSecondValue);
36
37             // Using universal functions:
38
39             Assert.True(PartialRead(value, 0, 1) == firstValue);
40             Assert.True(PartialRead(value, 1, -1) == secondValue);
41
42             firstValue = 0;
43             secondValue = 6892;
44
45             value = PartialWrite(value, firstValue, 0, 1);

```

```

46     value = PartialWrite(value, secondValue, 1, -1);
47
48     Assert.True(PartialRead(value, 0, 1) == firstValue);
49     Assert.True(PartialRead(value, 1, -1) == secondValue);
50 }
51
52 private static uint PartialWrite(uint target, uint source, int shift, int limit)
53 {
54     if (shift < 0)
55     {
56         shift = 32 + shift;
57     }
58     if (limit < 0)
59     {
60         limit = 32 + limit;
61     }
62     var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
63     var targetMask = ~(sourceMask << shift);
64     return (target & targetMask) | ((source & sourceMask) << shift);
65 }
66
67 private static uint PartialRead(uint target, int shift, int limit)
68 {
69     if (shift < 0)
70     {
71         shift = 32 + shift;
72     }
73     if (limit < 0)
74     {
75         limit = 32 + limit;
76     }
77     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