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

```csharp
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of arithmetic methods.</para>
    /// <para>Представляет набор арифметических методов.</para>
    /// </summary>
    public static class Arithmetic
    {
        /// <summary>
        /// <para>Performing adding the x and y arguments.</para>
        /// <para>Выполняет сложение аргументов x и y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The first term.</para>
        /// <para>Первое слагаемое.</para>
        /// </param>
        /// <param name="y">
        /// <para>The second term.</para>
        /// <para>Второе слагаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Sum of x and y.</para>
        /// <para>Сумма x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);

        /// <summary>
        /// <para>Performs subtracting y from x.</para>
        /// <para>Выполняет вычитание y из x.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>Minuend.</para>
        /// <para>Уменьшаемое.</para>
        /// </param>
        /// <param name="y">
        /// <para>Subtrahend.</para>
        /// <para>Вычитаемое.</para>
        /// </param>
        /// <returns>
        /// <para>Difference between x and y.</para>
        /// <para>Разность между x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);

        /// <summary>
        /// <para>Performs multiplication x by y.</para>
        /// <para>Выполняет умножение x на y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>First multiplier.</para>
        /// <para>Первый множитель.</para>
        /// </param>
        /// <param name="y">
        /// <para>Second multiplier.</para>
        /// <para>Второй множитель.</para>
        /// </param>
        /// <returns>
        /// <para>Product of x and y.</para>
```

```csharp
        /// <para>Произведение x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Multiply<T>(T x, T y) => Arithmetic<T>.Multiply(x, y);

        /// <summary>
        /// <para>Performs dividing x by y.</para>
        /// <para>Выполняет деление x на y.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>Dividend.</para>
        /// <para>Делимое.</para>
        /// </param>
        /// <param name="y">
        /// <para>Divider.</para>
        /// <para>Делитель.</para>
        /// </param>
        /// <returns>
        /// <para>Quoitent of x and y.</para>
        /// <para>Частное x и y.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Divide<T>(T x, T y) => Arithmetic<T>.Divide(x, y);

        /// <summary>
        /// <para>Increasing the number by one.</para>
        /// <para>Увеличивает число на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to increase.</para>
        /// <para>Число для увеличения.</para>
        /// </param>
        /// <returns>
        /// <para>Increase by one number.</para>
        /// <para>Увеличенное на единицу число.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para>Increases the value of argument by one.</para>
        /// <para>Увеличивает значение аргумента на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The argument to increase.</para>
        /// <para>Аргумент для увеличения.</para>
        /// </param>
        /// <returns>
        /// <para>Increased argument value.</para>
        /// <para>Увеличенное значение аргумента.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Increment<T>(ref T x) => x = Arithmetic<T>.Increment(x);

        /// <summary>
        /// <para>Decreases number by one.</para>
        /// <para>Уменьшение числа на единицу.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="x">
        /// <para>The number to reduce.</para>
        /// <para>Число для уменьшения.</para>
        /// </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>Уменьшеное 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
          arbitrary object types.</para>
9      /// <para>Предоставляет набор методов расширения выполняющих арифметические операции для
          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 =
           Arithmetic<T>.Increment(x);
31
32         /// <summary>
33         /// <para>Decrements the variable passed as an argument by one.</para>
34         /// <para>Уменьшает переданную в качестве аргумента переменную на единицу.</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 reference to the decremented variable.</para>
42         /// <para>Ссылка на уменьшаемую переменную.</para>
43         /// </param>
44         /// <returns>
45         /// <para>The value of the argument decremented by one.</para>
46         /// <para>Уменьшеное значение аргумента на единицу.</para>
```

```
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  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>Represents compiled arithmetic delegates.</para>
14     /// <para>Представляет набор скомпилированных делегатов арифметических операций.</para>
15     /// </summary>
16     public static class Arithmetic<T>
17     {
18         /// <summary>
19         /// <para>A read-only field that represents delegate of the addition function.</para>
20         /// <para>Поле только для чтения, представляющее делегат функции сложения.</para>
21         /// </summary>
22         public static readonly Func<T, T, T> Add = CompileAddDelegate();
23
24         /// <summary>
25         /// <para>A read-only field that represents delegate of the subtraction function.</para>
26         /// <para>Поле только для чтения, представляющее делегат функции вычитания.</para>
27         /// </summary>
28         public static readonly Func<T, T, T> Subtract = CompileSubtractDelegate();
29
30         /// <summary>
31         /// <para>A read-only field that represents delegate of the multiplication
32            ↪  function.</para>
32         /// <para>Поле только для чтения, представляющее делегат функции умножения.</para>
33         /// </summary>
34         public static readonly Func<T, T, T> Multiply = CompileMultiplyDelegate();
35
36         /// <summary>
37         /// <para>A read-only field that represents delegate of the division function.</para>
38         /// <para>Поле только для чтения, представляющее делегат функции деления.</para>
39         /// </summary>
40         public static readonly Func<T, T, T> Divide = CompileDivideDelegate();
41
42         /// <summary>
43         /// <para>A read-only field that represents delegate of the increment function.</para>
44         /// <para>Поле только для чтения, представляющее делегат функции инкремента.</para>
45         /// </summary>
46         public static readonly Func<T, T> Increment = CompileIncrementDelegate();
47
48         /// <summary>
49         /// <para>A read-only field that represents delegate of the decrement function.</para>
50         /// <para>Поле только для чтения, представляющее делегат функции декремента.</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>>(emiter =>
58             {
59                 Ensure.Always.IsNumeric<T>();
60                 emiter.LoadArguments(0, 1);
61                 emiter.Add();
62                 emiter.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>>(emiter =>
70             {
```

```
71              Ensure.Always.IsNumeric<T>();
72              emiter.LoadArguments(0, 1);
73              emiter.Subtract();
74              emiter.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>>(emiter =>
82          {
83              Ensure.Always.IsNumeric<T>();
84              emiter.LoadArguments(0, 1);
85              emiter.Emit(OpCodes.Mul);
86              emiter.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>>(emiter =>
94          {
95              Ensure.Always.IsNumeric<T>();
96              emiter.LoadArguments(0, 1);
97              if(NumericType<T>.IsSigned)
98              {
99                  emiter.Emit(OpCodes.Div);
100             }
101             else
102             {
103                 emiter.Emit(OpCodes.Div_Un);
104             }
105             emiter.Return();
106         });
107     }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     private static Func<T, T> CompileIncrementDelegate()
111     {
112         return DelegateHelpers.Compile<Func<T, T>>(emiter =>
113         {
114             Ensure.Always.IsNumeric<T>();
115             emiter.LoadArgument(0);
116             emiter.Increment<T>();
117             emiter.Return();
118         });
119     }
120
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     private static Func<T, T> CompileDecrementDelegate()
123     {
124         return DelegateHelpers.Compile<Func<T, T>>(emiter =>
125         {
126             Ensure.Always.IsNumeric<T>();
127             emiter.LoadArgument(0);
128             emiter.Decrement<T>();
129             emiter.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>
```

```csharp
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
       ↪  bits.</para>
132          /// <para>Выполняет побитовый свиг числа влево на указанное количество бит.</para>
133          /// </summary>
134          /// <typeparam name="T">
135          /// <para>The number's type.</para>
136          /// <para>Тип числа.</para>
137          /// </typeparam>
138          /// <param name="x">
139          /// <para>The number on which the left bitwise shift operation will be performed.</para>
140          /// <para>Число над которым будет производиться операция пиботового смещения
       ↪  влево.</para>
141          /// </param>
142          /// <param name="y">
143          /// <para>The number of bits to shift.</para>
144          /// <para>Количество бит на которые выполнить смещение.</para>
145          /// </param>
146          /// <returns>
147          /// <para>The value with discarded high-order bits that are outside the range of the
       ↪  number's type and set low-order empty bit positions to zero.</para>
148          /// <para>Значение с отброшенными старшими битами, которые находятся за пределами
       ↪  диапазона типа числа и устанавливленными пустыми битовыми позициями младших разрядов
       ↪  в ноль.</para>
149          /// </returns>
150          [MethodImpl(MethodImplOptions.AggressiveInlining)]
151          public static T ShiftLeft<T>(T x, int y) => Bit<T>.ShiftLeft(x, y);
152
153          /// <summary>
154          /// <para>Performs a bitwise shift of a number to the right by the specified number of
       ↪  bits.</para>
155          /// <para>Выполняет побитовый свиг числа вправо на указанное количество бит.</para>
156          /// </summary>
157          /// <typeparam name="T">
158          /// <para>The number's type.</para>
159          /// <para>Тип числа.</para>
160          /// </typeparam>
161          /// <param name="x">
162          /// <para>The number on which the right bitwise shift operation will be performed.</para>
163          /// <para>Число над которым будет производиться операция побитового смещения
       ↪  вправо.</para>
```

```csharp
164        /// </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
             ↪  target number.</para>
178        /// <para>Выполняет частичную запись определенного количества бит исходного числа в
             ↪  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) =>
             ↪  Bit<T>.PartialWrite(target, source, shift, limit);
206
207        /// <summary>
208        /// <para>Reads a specified number of bits from the number at specified position.</para>
209        /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
210        /// </summary>
211        /// <typeparam name="T">
212        /// <para>The number's type.</para>
213        /// <para>Тип числа.</para>
214        /// </typeparam>
215        /// <param name="target">
216        /// <para>The number from which the partial read will be performed.</para>
217        /// <para>Число из которого будет выполнено частичное чтение.</para>
218        /// </param>
219        /// <param name="shift">
220        /// <para>The start position to read from.</para>
221        /// <para>Стартовая позиция чтения.</para>
222        /// </param>
223        /// <param name="limit">
224        /// <para>The number of bits to read.</para>
225        /// <para>Количество бит, которые нужно считать.</para>
226        /// </param>
227        /// <returns>
228        /// <para>The number consisting of bits read from the source number.</para>
229        /// <para>Число состоящее из считанных из исходного числа бит.</para>
230        /// </returns>
231        [MethodImpl(MethodImplOptions.AggressiveInlining)]
232        public static T PartialRead<T>(T target, int shift, int limit) =>
             ↪  Bit<T>.PartialRead(target, shift, limit);
233    }
234  }
```

```csharp
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a set of bitwise operation.</para>
    /// <para>Представляет набор битовых операций.</para>
    /// </summary>
    public static class BitwiseExtensions
    {
        /// <summary>
        /// <para>Performs bitwise inversion of a number.</para>
        /// <para>Выполняет побитовую инверсию числа.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The number to invert.</para>
        /// <para>Число для инверсии.</para>
        /// </param>
        /// <returns>
        /// <para>An inverted value of the number.</para>
        /// <para>Обратное значение числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T Not<T>(this ref T target) where T : struct => target = Bit.Not(target);

        /// <summary>
        /// <para>Performs a partial write of a specified number of bits from source number to
        ///   target number.</para>
        /// <para>Выполняет частичную запись определенного количества бит исходного числа в
        ///   целевое число.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The numbers' type.</para>
        /// <para>Тип чисел.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The value to which the partial write will be performed.</para>
        /// <para>Значение в которое будет выполнена частичная запись.</para>
        /// </param>
        /// <param name="source">
        /// <para>Data source for writing.</para>
        /// <para>Источник данных для записи.</para>
        /// </param>
        /// <param name="shift">
        /// <para>The start position to read from.</para>
        /// <para>Стартовая позиция чтения.</para>
        /// </param>
        /// <param name="limit">
        /// <para>The number of bits to write from source to target.</para>
        /// <para>Количество бит, которые нужно записать из source в target.</para>
        /// </param>
        /// <returns>
        /// <para>The target number updated with bits from source number.</para>
        /// <para>Целевое число с обновленными битами из исходного числа.</para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PartialWrite<T>(this ref T target, T source, int shift, int limit) where
          T : struct => target = Bit<T>.PartialWrite(target, source, shift, limit);

        /// <summary>
        /// <para>Reads a specified number of bits from the number at specified position.</para>
        /// <para>Считывает указанное количество бит из числа в указанной позиции.</para>
        /// </summary>
        /// <typeparam name="T">
        /// <para>The number's type.</para>
        /// <para>Тип числа.</para>
        /// </typeparam>
        /// <param name="target">
        /// <para>The number from which the partial read will be performed.</para>
        /// <para>Число из которого будет выполнено частичное чтение.</para>
        /// </param>
```

```csharp
         /// <param name="shift">
         /// <para>The start position to read from.</para>
         /// <para>Стартовая позиция чтения.</para>
         /// </param>
         /// <param name="limit">
         /// <para>The number of bits to read.</para>
         /// <para>Количество бит, которые нужно считать.</para>
         /// </param>
         /// <returns>
         /// <para>The number consisting of bits read from the source number.</para>
         /// <para>Число состоящее из считанных из исходного числа бит.</para>
         /// </returns>
         [MethodImpl(MethodImplOptions.AggressiveInlining)]
         public static T PartialRead<T>(this T target, int shift, int limit) =>
             ↪  Bit<T>.PartialRead(target, shift, limit);
     }
 }
```

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

```csharp
using System;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Reflection;

// ReSharper disable StaticFieldInGenericType
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
     /// <summary>
     /// <para>.</para>
     /// <para>.</para>
     /// </summary>
     public static class Bit<T>
     {
         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, T> Not = CompileNotDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, T, T> Or = CompileOrDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, T, T> And = CompileAndDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, int, T> ShiftLeft = CompileShiftLeftDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, int, T> ShiftRight = CompileShiftRightDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, T, int, int, T> PartialWrite =
             ↪  CompilePartialWriteDelegate();

         /// <summary>
         /// <para>.</para>
         /// <para>.</para>
         /// </summary>
         public static readonly Func<T, int, int, T> PartialRead = CompilePartialReadDelegate();

         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```csharp
    private static Func<T, T> CompileNotDelegate()
    {
        return DelegateHelpers.Compile<Func<T, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            emiter.LoadArguments(0);
            emiter.Not();
            emiter.Return();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static Func<T, T, T> CompileOrDelegate()
    {
        return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            emiter.LoadArguments(0, 1);
            emiter.Or();
            emiter.Return();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static Func<T, T, T> CompileAndDelegate()
    {
        return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            emiter.LoadArguments(0, 1);
            emiter.And();
            emiter.Return();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static Func<T, int, T> CompileShiftLeftDelegate()
    {
        return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            emiter.LoadArguments(0, 1);
            emiter.ShiftLeft();
            emiter.Return();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static Func<T, int, T> CompileShiftRightDelegate()
    {
        return DelegateHelpers.Compile<Func<T, int, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            emiter.LoadArguments(0, 1);
            emiter.ShiftRight();
            emiter.Return();
        });
    }

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    private static Func<T, T, int, int, T> CompilePartialWriteDelegate()
    {
        return DelegateHelpers.Compile<Func<T, T, int, int, T>>(emiter =>
        {
            Ensure.Always.IsNumeric<T>();
            var constants = GetConstants();
            var bitsNumber = constants.Item1;
            var numberFilledWithOnes = constants.Item2;
            ushort shiftArgument = 2;
            ushort limitArgument = 3;
            var checkLimit = emiter.DefineLabel();
            var calculateSourceMask = emiter.DefineLabel();
            // Check shift
            emiter.LoadArgument(shiftArgument);
            emiter.LoadConstant(0);
            emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
            // Fix shift
            emiter.LoadConstant(bitsNumber);
```

```csharp
                    emiter.LoadArgument(shiftArgument);
                    emiter.Add();
                    emiter.StoreArgument(shiftArgument);
                    emiter.MarkLabel(checkLimit);
                    // Check limit
                    emiter.LoadArgument(limitArgument);
                    emiter.LoadConstant(0);
                    emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                    // Fix limit
                    emiter.LoadConstant(bitsNumber);
                    emiter.LoadArgument(limitArgument);
                    emiter.Add();
                    emiter.StoreArgument(limitArgument);
                    emiter.MarkLabel(calculateSourceMask);
                    var sourceMask = emiter.DeclareLocal<T>();
                    var targetMask = emiter.DeclareLocal<T>();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.LoadArgument(limitArgument);
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.And();
                    emiter.StoreLocal(sourceMask);
                    emiter.LoadLocal(sourceMask);
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.StoreLocal(targetMask);
                    emiter.LoadArgument(0); // target
                    emiter.LoadLocal(targetMask);
                    emiter.And();
                    emiter.LoadArgument(1); // source
                    emiter.LoadLocal(sourceMask);
                    emiter.And();
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.Or();
                    emiter.Return();
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Func<T, int, int, T> CompilePartialReadDelegate()
        {
            return DelegateHelpers.Compile<Func<T, int, int, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                var constants = GetConstants();
                var bitsNumber = constants.Item1;
                var numberFilledWithOnes = constants.Item2;
                ushort shiftArgument = 1;
                ushort limitArgument = 2;
                var checkLimit = emiter.DefineLabel();
                var calculateSourceMask = emiter.DefineLabel();
                // Check shift
                emiter.LoadArgument(shiftArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(checkLimit); // Skip fix
                // Fix shift
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(shiftArgument);
                emiter.Add();
                emiter.StoreArgument(shiftArgument);
                emiter.MarkLabel(checkLimit);
                // Check limit
                emiter.LoadArgument(limitArgument);
                emiter.LoadConstant(0);
                emiter.BranchIfGreaterOrEqual(calculateSourceMask); // Skip fix
                // Fix limit
                emiter.LoadConstant(bitsNumber);
                emiter.LoadArgument(limitArgument);
                emiter.Add();
                emiter.StoreArgument(limitArgument);
                emiter.MarkLabel(calculateSourceMask);
                var sourceMask = emiter.DeclareLocal<T>();
                var targetMask = emiter.DeclareLocal<T>();
                emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                emiter.LoadArgument(limitArgument); // limit
```

```csharp
                    emiter.ShiftLeft();
                    emiter.Not();
                    emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                    emiter.And();
                    emiter.StoreLocal(sourceMask);
                    emiter.LoadLocal(sourceMask);
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftLeft();
                    emiter.StoreLocal(targetMask);
                    emiter.LoadArgument(0); // target
                    emiter.LoadLocal(targetMask);
                    emiter.And();
                    emiter.LoadArgument(shiftArgument);
                    emiter.ShiftRight();
                    emiter.Return();
                });
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Tuple<int, T> GetConstants()
        {
            var type = typeof(T);
            if (type == typeof(ulong))
            {
                return new Tuple<int, T>(64, (T)(object)ulong.MaxValue);
            }
            if (type == typeof(uint))
            {
                return new Tuple<int, T>(32, (T)(object)uint.MaxValue);
            }
            if (type == typeof(ushort))
            {
                return new Tuple<int, T>(16, (T)(object)ushort.MaxValue);
            }
            if (type == typeof(byte))
            {
                return new Tuple<int, T>(8, (T)(object)byte.MaxValue);
            }
            throw new NotSupportedException();
        }
    }
}
```

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

```csharp
using System;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Numbers
{
    /// <summary>
    /// <para>Represents a collection of algebraic methods.</para>
    /// <para>Представляет набор алгебраических методов.</para>
    /// </summary>
    /// <remarks>Resizable array (FileMappedMemory) for values cache may be used. or cached
    ↪   oeis.org</remarks>
    public static class Math
    {
        /// <remarks>
        /// <para>Source: https://oeis.org/A000142/list </para>
        /// <para>Источник: https://oeis.org/A000142/list </para>
        /// </remarks>
        private static readonly ulong[] _factorials =
        {
            1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
            479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
            355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000
        };

        /// <remarks>
        /// <para>Source: https://oeis.org/A000108/list </para>
        /// <para>Источник: https://oeis.org/A000108/list </para>
        /// </remarks>
        private static readonly ulong[] _catalans =
        {
            1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012,
            742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190,
            6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,
```

```csharp
                    18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304,
                    14544636039226909, 5553064877048198, 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.");
                }
```

```
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
          ↪ arbitrary object types.</para>
9       /// <para>Предоставляет набор методов расширения выполняющих математические операции для
          ↪ объектов произвольного типа.</para>
10      /// </summary>
11      public static class MathExtensions
12      {
13          /// <summary>
14          /// <para>Takes a module from a number.</para>
15          /// <para>Берёт модуль от числа.</para>
16          /// </summary>
17          /// <typeparam name="T">
18          /// <para>The number's type.</para>
```

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

```csharp
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>>(emiter =>
33             {
34                 Ensure.Always.IsNumeric<T>();
35                 emiter.LoadArgument(0);
36                 if (NumericType<T>.IsSigned)
37                 {
38                     emiter.Call(typeof(System.Math).GetMethod("Abs", Types<T>.Array));
39                 }
40                 emiter.Return();
41             });
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
45        private static Func<T, T> CompileNegateDelegate()
46        {
47            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
48            {
49                emiter.LoadArgument(0);
50                emiter.Negate();
51                emiter.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         }
```

```csharp
        [Fact]
        public static void ByteBitwiseOperationsTest()
        {
            Assert.True(Bit<byte>.Not(2) == unchecked((byte)~2));
            Assert.True(Bit<byte>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<byte>.And(1, 2) == (1 & 2));
            Assert.True(Bit<byte>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<byte>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt16BitwiseOperationsTest()
        {
            Assert.True(Bit<ushort>.Not(2) == unchecked((ushort)~2));
            Assert.True(Bit<ushort>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<ushort>.And(1, 2) == (1 & 2));
            Assert.True(Bit<ushort>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<ushort>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt32BitwiseOperationsTest()
        {
            Assert.True(Bit<uint>.Not(2) == unchecked((uint)~2));
            Assert.True(Bit<uint>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<uint>.And(1, 2) == (1 & 2));
            Assert.True(Bit<uint>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<uint>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void UInt64BitwiseOperationsTest()
        {
            Assert.True(Bit<ulong>.Not(2) == unchecked((ulong)~2));
            Assert.True(Bit<ulong>.Or(1, 2) == (1 | 2));
            Assert.True(Bit<ulong>.And(1, 2) == (1 & 2));
            Assert.True(Bit<ulong>.ShiftLeft(1, 2) == (1 << 2));
            Assert.True(Bit<ulong>.ShiftRight(1, 2) == (1 >> 2));
        }

        [Fact]
        public static void PartialReadWriteTest()
        {
            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
                uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                Assert.True(firstValue == unpackagedFirstValue);
                Assert.True(secondValue == unpackagedSecondValue);

                // Using universal functions:
                Assert.True(PartialRead(value, 0, 1) == firstValue);
                Assert.True(PartialRead(value, 1, -1) == secondValue);

                firstValue = 0;
                secondValue = 6892;

                value = PartialWrite(value, firstValue, 0, 1);
                value = PartialWrite(value, secondValue, 1, -1);

                Assert.True(PartialRead(value, 0, 1) == firstValue);
                Assert.True(PartialRead(value, 1, -1) == secondValue);
            }

            {
                uint firstValue = 1;
                uint secondValue = 1543;

                // Pack (join) two values at the same time
                uint value = secondValue << 1 | firstValue;

                uint unpackagedFirstValue = value & 1;
```

```csharp
                        uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                        Assert.True(firstValue == unpackagedFirstValue);
                        Assert.True(secondValue == unpackagedSecondValue);

                        // Using universal functions:
                        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
                        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);

                        firstValue = 0;
                        secondValue = 6892;

                        value = Bit.PartialWrite(value, firstValue, 0, 1);
                        value = Bit.PartialWrite(value, secondValue, 1, -1);

                        Assert.True(Bit.PartialRead(value, 0, 1) == firstValue);
                        Assert.True(Bit.PartialRead(value, 1, -1) == secondValue);
                }

                {
                        uint firstValue = 1;
                        uint secondValue = 1543;

                        // Pack (join) two values at the same time
                        uint value = secondValue << 1 | firstValue;

                        uint unpackagedFirstValue = value & 1;
                        uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

                        Assert.True(firstValue == unpackagedFirstValue);
                        Assert.True(secondValue == unpackagedSecondValue);

                        // Using universal functions:
                        var readMasksAndShiftFor0And1 = GetReadMaskAndShift(0, 1);
                        var readMasksAndShiftFor1AndMinus1 = GetReadMaskAndShift(1, -1);
                        var writeMasksAndShiftFor0And1 = GetWriteMasksAndShift(0, 1);
                        var writeMasksAndShiftFor1AndMinus1 = GetWriteMasksAndShift(1, -1);

                        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);

                        firstValue = 0;
                        secondValue = 6892;

                        value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
                        value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);

                        Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
                        Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
                }
        }

        // TODO: Can be optimized using precalculation of TargetMask and SourceMask
        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = ~(sourceMask << shift);
            return target & targetMask | (source & sourceMask) << shift;
        }

        private static uint PartialRead(uint target, int shift, int limit)
        {
            if (shift < 0)
            {
                shift = 32 + shift;
            }
            if (limit < 0)
            {
                limit = 32 + limit;
            }
            var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
            var targetMask = sourceMask << shift;
```

```csharp
                return (target & targetMask) >> shift;
            }

            private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
            {
                if (shift < 0)
                {
                    shift = 32 + shift;
                }
                if (limit < 0)
                {
                    limit = 32 + limit;
                }
                var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
                var targetMask = ~(sourceMask << shift);
                return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
            }

            private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
            {
                if (shift < 0)
                {
                    shift = 32 + shift;
                }
                if (limit < 0)
                {
                    limit = 32 + limit;
                }
                var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
                var targetMask = sourceMask << shift;
                return new Tuple<uint, int>(targetMask, shift);
            }

            private static uint PartialWrite(uint target, uint targetMask, uint source, uint
              sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;

            private static uint PartialWrite(uint target, uint source, Tuple<uint, uint, int>
              masksAndShift) => PartialWrite(target, masksAndShift.Item1, source,
              masksAndShift.Item2, masksAndShift.Item3);

            private static uint PartialRead(uint target, uint targetMask, int shift) => (target &
              targetMask) >> shift;

            private static uint PartialRead(uint target, Tuple<uint, int> masksAndShift) =>
              PartialRead(target, masksAndShift.Item1, masksAndShift.Item2);

            [Fact]
            public static void BugWithLoadingConstantOf8Test()
            {
                Bit<byte>.PartialWrite(0, 1, 5, -5);
            }
        }
    }
```

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

```csharp
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class MathExtensionsTests
    {
        [Fact]
        public static void AbsTest()
        {
            var number = -1L;
            var returnValue = number.Abs();
            Assert.Equal(1L, returnValue);
            Assert.Equal(1L, number);
        }

        [Fact]
        public static void NegateTest()
        {
            var number = 2L;
            var returnValue = number.Negate();
            Assert.Equal(-2L, returnValue);
            Assert.Equal(-2L, number);
        }

```

```csharp
          [Fact]
          public static void UnsignedNegateTest()
          {
              var number = 2UL;
              var returnValue = number.Negate();
              Assert.Equal(18446744073709551614, returnValue);
              Assert.Equal(18446744073709551614, number);
          }
      }
  }
```

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

```csharp
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class MathTests
    {
        [Fact]
        public static void CompiledOperationsTest()
        {
            Assert.True(Math.Abs(Arithmetic<double>.Subtract(3D, 2D) - 1D) < 0.01);
        }
    }
}
```

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

```csharp
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class SystemTests
    {
        [Fact]
        public static void PossiblePackTwoValuesIntoOneTest()
        {
            uint value = 0;

            // Set one to first bit
            value |= 1;

            Assert.True(value == 1);

            // Set zero to first bit
            value &= 0xFFFFFFFE;

            // Get first bit
            uint read = value & 1;

            Assert.True(read == 0);

            uint firstValue = 1;
            uint secondValue = 1543;

            // Pack (join) two values at the same time
            value = (secondValue << 1) | firstValue;

            uint unpackagedFirstValue = value & 1;
            uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;

            Assert.True(firstValue == unpackagedFirstValue);
            Assert.True(secondValue == unpackagedSecondValue);

            // Using universal functions:

            Assert.True(PartialRead(value, 0, 1) == firstValue);
            Assert.True(PartialRead(value, 1, -1) == secondValue);

            firstValue = 0;
            secondValue = 6892;

            value = PartialWrite(value, firstValue, 0, 1);
            value = PartialWrite(value, secondValue, 1, -1);

            Assert.True(PartialRead(value, 0, 1) == firstValue);
            Assert.True(PartialRead(value, 1, -1) == secondValue);
        }

        private static uint PartialWrite(uint target, uint source, int shift, int limit)
        {
```

```csharp
                    if (shift < 0)
                    {
                        shift = 32 + shift;
                    }
                    if (limit < 0)
                    {
                        limit = 32 + limit;
                    }
                    var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
                    var targetMask = ~(sourceMask << shift);
                    return (target & targetMask) | ((source & sourceMask) << shift);
                }

        private static uint PartialRead(uint target, int shift, int limit)
        {
                    if (shift < 0)
                    {
                        shift = 32 + shift;
                    }
                    if (limit < 0)
                    {
                        limit = 32 + limit;
                    }
                    var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
                    var targetMask = sourceMask << shift;
                    return (target & targetMask) >> shift;
                }
        }
}
```

Index