./Platform.Numbers/Arithmetic.cs

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

namespace Platform.Numbers
{
    public static class Arithmetic
    {
        public static T Add<T>(T x, T y) => Arithmetic<T>.Add(x, y);
        public static T Increment<T>(T x) => Arithmetic<T>.Increment(x);
        public static T Subtract<T>(T x, T y) => Arithmetic<T>.Subtract(x, y);
        public static T Subtract<T>(Integer<T> x, Integer<T> y) => Arithmetic<T>.Subtract(x, y);
        public static T Decrement<T>(T x) => Arithmetic<T>.Decrement(x);
    }
}
```

./Platform.Numbers/ArithmeticExtensions.cs

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

namespace Platform.Numbers
{
    public static class ArithmeticExtensions
    {
        public static T Decrement<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Decrement(x);
        public static T Increment<T>(this ref T x) where T : struct => x =
        ↪  Arithmetic<T>.Increment(x);
    }
}
```

./Platform.Numbers/Arithmetic[T].cs

```
using System;
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
{
    public static class Arithmetic<T>
    {
        public static readonly Func<T, T, T> Add;
        public static readonly Func<T, T, T> Subtract;
        public static readonly Func<T, T> Increment;
        public static readonly Func<T, T> Decrement;

        static Arithmetic()
        {
            Add = CompileAddDelegate();
            Subtract = CompileSubtractDelegate();
            Increment = CompileIncrementDelegate();
            Decrement = CompileDecrementDelegate();
        }

        private static Func<T, T, T> CompileAddDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Add();
                emiter.Return();
            });
        }

        private static Func<T, T, T> CompileSubtractDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Subtract();
                emiter.Return();
            });
        }

        private static Func<T, T> CompileIncrementDelegate()
        {
```

```
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                emiter.Increment<T>();
                emiter.Return();
            });
        }

        private static Func<T, T> CompileDecrementDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                emiter.Decrement<T>();
                emiter.Return();
            });
        }
    }
}
```

./Platform.Numbers/Bit.cs

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

namespace Platform.Numbers
{
    public static class Bit
    {
        public static long Count(long x)
        {
            long n = 0;
            while (x != 0)
            {
                n++;
                x &= x - 1;
            }
            return n;
        }

        public static int GetLowestPosition(ulong value)
        {
            if (value == 0)
            {
                return -1;
            }
            var position = 0;
            while ((value & 1UL) == 0)
            {
                value >>= 1;
                ++position;
            }
            return position;
        }

        public static T And<T>(T x, T y) => Bit<T>.And(x, y);

        public static T PartialWrite<T>(T target, T source, int shift, int limit) =>
        ↪  Bit<T>.PartialWrite(target, source, shift, limit);

        public static T PartialRead<T>(T target, int shift, int limit) =>
        ↪  Bit<T>.PartialRead(target, shift, limit);
    }
}
```

./Platform.Numbers/BitExtensions.cs

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

namespace Platform.Numbers
{
    public static class BitwiseExtensions
    {
        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);
        public static T PartialRead<T>(this T target, int shift, int limit) =>
        ↪  Bit<T>.PartialRead(target, shift, limit);
    }
}
```

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

```
using System;
using System.Reflection.Emit;
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
{
    public static class Bit<T>
    {
        public static readonly Func<T, T, T> And;
        public static readonly Func<T, T, int, int, T> PartialWrite;
        public static readonly Func<T, int, int, T> PartialRead;

        static Bit()
        {
            And = CompileAndDelegate();
            PartialWrite = CompilePartialWriteDelegate();
            PartialRead = CompilePartialReadDelegate();
        }

        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();
            });
        }

        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);
                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);
                LoadMaxValueConstant(emiter);
                emiter.LoadArgument(limitArgument);
                emiter.ShiftLeft();
                emiter.Not();
                //emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                LoadMaxValueConstant(emiter);
                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();
            });
        }

        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);
                LoadMaxValueConstant(emiter);
                emiter.LoadArgument(limitArgument); // limit
                emiter.ShiftLeft();
                emiter.Not();
                //emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                LoadMaxValueConstant(emiter);
                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();
            });
        }

        private static void LoadMaxValueConstant(ILGenerator emiter)
        {
            var type = typeof(T);
            if (type == typeof(ulong))
            {
```

```
                emiter.Emit(OpCodes.Ldc_I8, unchecked((long)ulong.MaxValue));
            }
            else if (type == typeof(uint))
            {
                emiter.Emit(OpCodes.Ldc_I4, unchecked((int)uint.MaxValue));
            }
            else if (type == typeof(ushort))
            {
                emiter.Emit(OpCodes.Ldc_I4, unchecked((int)ushort.MaxValue));
            }
            else if (type == typeof(byte))
            {
                emiter.Emit(OpCodes.Ldc_I4_S, unchecked((sbyte)byte.MaxValue));
            }
            else
            {
                throw new NotSupportedException();
            }
        }

        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();
        }
    }
}
```

./Platform.Numbers/Integer.cs

```
using System;
using Platform.Converters;

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

namespace Platform.Numbers
{
    public struct Integer : IEquatable<Integer>
    {
        public readonly ulong Value;
        public Integer(ulong value) => Value = value;
        public static implicit operator Integer(ulong integer) => new Integer(integer);
        public static implicit operator Integer(long integer) => To.UInt64(integer);
        public static implicit operator Integer(uint integer) => new Integer(integer);
        public static implicit operator Integer(int integer) => To.UInt64(integer);
        public static implicit operator Integer(ushort integer) => new Integer(integer);
        public static implicit operator Integer(short integer) => To.UInt64(integer);
        public static implicit operator Integer(byte integer) => new Integer(integer);
        public static implicit operator Integer(sbyte integer) => To.UInt64(integer);
        public static implicit operator Integer(bool integer) => To.UInt64(integer);
        public static implicit operator ulong(Integer integer) => integer.Value;
        public static implicit operator long(Integer integer) => To.Int64(integer.Value);
        public static implicit operator uint(Integer integer) => To.UInt32(integer.Value);
        public static implicit operator int(Integer integer) => To.Int32(integer.Value);
        public static implicit operator ushort(Integer integer) => To.UInt16(integer.Value);
        public static implicit operator short(Integer integer) => To.Int16(integer.Value);
        public static implicit operator byte(Integer integer) => To.Byte(integer.Value);
        public static implicit operator sbyte(Integer integer) => To.SByte(integer.Value);
        public static implicit operator bool(Integer integer) => To.Boolean(integer.Value);
        public bool Equals(Integer other) => Value == other.Value;
        public override string ToString() => Value.ToString();
    }
}
```

./Platform.Numbers/Integer[T].cs

```
using System;
using System.Reflection;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Converters;

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

namespace Platform.Numbers
{
    public struct Integer<T> : IEquatable<Integer<T>>
    {
        private static readonly EqualityComparer<T> _equalityComparer =
        ↪  EqualityComparer<T>.Default;
        private static readonly Func<ulong, Integer<T>> _create;

        public static readonly T Zero;
        public static readonly T One;
        public static readonly T Two;

        public readonly T Value;

        static Integer()
        {
            _create = CompileCreateDelegate();
            try
            {
                Zero = default;
                One = Arithmetic.Increment(Zero);
                Two = Arithmetic.Increment(One);
            }
            catch (Exception exception)
            {
                exception.Ignore();
            }
        }

        public Integer(T value) => Value = value;

        public static implicit operator Integer(Integer<T> integer)
        {
            if (typeof(T) == typeof(Integer))
            {
                return (Integer)(object)integer.Value;
            }
            return Convert.ToUInt64(integer.Value);
        }

        public static implicit operator ulong(Integer<T> integer) => ((Integer)integer).Value;

        public static implicit operator T(Integer<T> integer) => integer.Value;

        public static implicit operator Integer<T>(T integer) => new Integer<T>(integer);

        public static implicit operator Integer<T>(ulong integer) => _create(integer);

        public static implicit operator Integer<T>(Integer integer) => _create(integer.Value);

        public static implicit operator Integer<T>(long integer) => To.UInt64(integer);

        public static implicit operator Integer<T>(uint integer) => new Integer(integer);

        public static implicit operator Integer<T>(int integer) => To.UInt64(integer);

        public static implicit operator Integer<T>(ushort integer) => new Integer(integer);

        public static implicit operator Integer<T>(short integer) => To.UInt64(integer);

        public static implicit operator Integer<T>(byte integer) => new Integer(integer);

        public static implicit operator Integer<T>(sbyte integer) => To.UInt64(integer);

        public static implicit operator Integer<T>(bool integer) => To.UInt64(integer);

        public static implicit operator long(Integer<T> integer) => To.Int64(integer);

        public static implicit operator uint(Integer<T> integer) => To.UInt32(integer);

```

```
        public static implicit operator int(Integer<T> integer) => To.Int32(integer);

        public static implicit operator ushort(Integer<T> integer) => To.UInt16(integer);

        public static implicit operator short(Integer<T> integer) => To.Int16(integer);

        public static implicit operator byte(Integer<T> integer) => To.Byte(integer);

        public static implicit operator sbyte(Integer<T> integer) => To.SByte(integer);

        public static implicit operator bool(Integer<T> integer) => To.Boolean(integer);

        public bool Equals(Integer<T> other) => _equalityComparer.Equals(Value, other.Value);

        public override string ToString() => Value.ToString();

        private static Func<ulong, Integer<T>> CompileCreateDelegate()
        {
            return DelegateHelpers.Compile<Func<ulong, Integer<T>>>(emiter =>
            {
                if (typeof(T) != typeof(Integer))
                {
                    Ensure.Always.CanBeNumeric<T>();
                }
                emiter.LoadArgument(0);
                if (typeof(T) != typeof(ulong) && typeof(T) != typeof(Integer))
                {
                    emiter.Call(typeof(To).GetTypeInfo().GetMethod(typeof(T).Name,
                     ↪  Types<ulong>.Array));
                }
                if (NumericType<T>.IsNullable)
                {
                    emiter.NewObject(typeof(T), NumericType<T>.UnderlyingType);
                }
                if (typeof(T) == typeof(Integer))
                {
                    emiter.NewObject(typeof(Integer), typeof(ulong));
                }
                emiter.NewObject(typeof(Integer<T>), typeof(T));
                emiter.Return();
            });
        }
    }
}
```

./Platform.Numbers/Math.cs

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

namespace Platform.Numbers
{
    /// <remarks>
    /// Resizable array (FileMappedMemory) for values cache may be used. or cached oeis.org
    /// </remarks>
    public static class Math
    {
        /// <remarks>
        /// Source: https://oeis.org/A000142/list
        /// </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>
        /// Source: https://oeis.org/A000108/list
        /// </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,
            18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304
        };

        public static double Factorial(double n)
        {
            if (n <= 1)
```

```
            {
                return 1;
            }
            if (n < _factorials.Length)
            {
                return _factorials[(int)n];
            }
            return n * Factorial(n - 1);
        }

        public static double Catalan(double n)
        {
            if (n <= 1)
            {
                return 1;
            }
            if (n < _catalans.Length)
            {
                return _catalans[(int)n];
            }
            return Factorial(2 * n) / (Factorial(n + 1) * Factorial(n));
        }

        public static bool IsPowerOfTwo(ulong x) => (x & x - 1) == 0;

        public static T Abs<T>(T x) => Math<T>.Abs(x);

        public static T Negate<T>(T x) => Math<T>.Negate(x);
    }
}
```

./Platform.Numbers/MathExtensions.cs

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

namespace Platform.Numbers
{
    public static class MathExtensions
    {
        public static T Abs<T>(this ref T x) where T : struct => x = Math<T>.Abs(x);
        public static T Negate<T>(this ref T x) where T : struct => x = Math<T>.Negate(x);
    }
}
```

./Platform.Numbers/Math[T].cs

```
using System;
using System.Reflection;
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
{
    public static class Math<T>
    {
        public static readonly Func<T, T> Abs;
        public static readonly Func<T, T> Negate;

        static Math()
        {
            Abs = CompileAbsDelegate();
            Negate = CompileNegateDelegate();
        }

        private static Func<T, T> CompileAbsDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                if (NumericType<T>.IsSigned)
                {
                    emiter.Call(typeof(System.Math).GetTypeInfo().GetMethod("Abs",
                    ↪  Types<T>.Array));
                }
                emiter.Return();
            });
        }
```

```

        private static Func<T, T> CompileNegateDelegate()
        {
            return DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsSigned<T>();
                emiter.LoadArgument(0);
                emiter.Negate();
                emiter.Return();
            });
        }
    }
}
```

./Platform.Numbers.Tests/ArithmeticExtensionsTests.cs

```
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class ArithmeticExtensionsTests
    {
        [Fact]
        public static void IncrementTest()
        {
            var number = 0UL;
            var returnValue = number.Increment();
            Assert.Equal(1UL, returnValue);
            Assert.Equal(1UL, number);
        }

        [Fact]
        public static void DecrementTest()
        {
            var number = 1UL;
            var returnValue = number.Decrement();
            Assert.Equal(0UL, returnValue);
            Assert.Equal(0UL, number);
        }
    }
}
```

./Platform.Numbers.Tests/ArithmeticTests.cs

```
using System;
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class ArithmeticTests
    {
        [Fact]
        public static void CompiledOperationsTest()
        {
            Assert.True(Arithmetic<short>.Add(1, 2) == 3);
            Assert.True(Arithmetic<byte>.Increment(1) == 2);
            Assert.True(Arithmetic<ulong>.Decrement(2) == 1);
            Assert.Throws<NotSupportedException>(() => Arithmetic<string>.Subtract("1", "2"));
        }
    }
}
```

./Platform.Numbers.Tests/BitTests.cs

```
using System;
using Xunit;

namespace Platform.Numbers.Tests
{
    public static class BitTests
    {
        [Theory]
        [InlineData(00, -1)] // 0000 0000 (none, -1)
        [InlineData(01, 00)] // 0000 0001 (first, 0)
        [InlineData(08, 03)] // 0000 1000 (forth, 3)
        [InlineData(88, 03)] // 0101 1000 (forth, 3)
        public static void GetLowestBitPositionTest(ulong value, int expectedPosition)
        {
            Assert.True(Bit.GetLowestPosition(value) == expectedPosition);
        }

```

```
        [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;
                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);
```

```
99
100                 firstValue = 0;
101                 secondValue = 6892;
102
103                 value = PartialWrite(value, firstValue, writeMasksAndShiftFor0And1);
104                 value = PartialWrite(value, secondValue, writeMasksAndShiftFor1AndMinus1);
105
106                 Assert.True(PartialRead(value, readMasksAndShiftFor0And1) == firstValue);
107                 Assert.True(PartialRead(value, readMasksAndShiftFor1AndMinus1) == secondValue);
108             }
109         }
110
111         // TODO: Can be optimized using precalculation of TargetMask and SourceMask
112         private static uint PartialWrite(uint target, uint source, int shift, int limit)
113         {
114             if (shift < 0)
115             {
116                 shift = 32 + shift;
117             }
118             if (limit < 0)
119             {
120                 limit = 32 + limit;
121             }
122             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
123             var targetMask = ~(sourceMask << shift);
124             return target & targetMask | (source & sourceMask) << shift;
125         }
126
127         private static uint PartialRead(uint target, int shift, int limit)
128         {
129             if (shift < 0)
130             {
131                 shift = 32 + shift;
132             }
133             if (limit < 0)
134             {
135                 limit = 32 + limit;
136             }
137             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
138             var targetMask = sourceMask << shift;
139             return (target & targetMask) >> shift;
140         }
141
142         private static Tuple<uint, uint, int> GetWriteMasksAndShift(int shift, int limit)
143         {
144             if (shift < 0)
145             {
146                 shift = 32 + shift;
147             }
148             if (limit < 0)
149             {
150                 limit = 32 + limit;
151             }
152             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
153             var targetMask = ~(sourceMask << shift);
154             return new Tuple<uint, uint, int>(targetMask, sourceMask, shift);
155         }
156
157         private static Tuple<uint, int> GetReadMaskAndShift(int shift, int limit)
158         {
159             if (shift < 0)
160             {
161                 shift = 32 + shift;
162             }
163             if (limit < 0)
164             {
165                 limit = 32 + limit;
166             }
167             var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
168             var targetMask = sourceMask << shift;
169             return new Tuple<uint, int>(targetMask, shift);
170         }
171
172         private static uint PartialWrite(uint target, uint targetMask, uint source, uint
            ↪  sourceMask, int shift) => target & targetMask | (source & sourceMask) << shift;
173
174         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);
    }
}
```

./Platform.Numbers.Tests/MathExtensionsTests.cs

```
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);
        }
    }
}
```

./Platform.Numbers.Tests/MathTests.cs

```
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);
        }
    }
}
```

./Platform.Numbers.Tests/SystemTests.cs

```
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;
```

```
28
29              // Pack (join) two values at the same time
30              value = (secondValue << 1) | firstValue;
31
32              uint unpackagedFirstValue = value & 1;
33              uint unpackagedSecondValue = (value & 0xFFFFFFFE) >> 1;
34
35              Assert.True(firstValue == unpackagedFirstValue);
36              Assert.True(secondValue == unpackagedSecondValue);
37
38              // Using universal functions:
39
40              Assert.True(PartialRead(value, 0, 1) == firstValue);
41              Assert.True(PartialRead(value, 1, -1) == secondValue);
42
43              firstValue = 0;
44              secondValue = 6892;
45
46              value = PartialWrite(value, firstValue, 0, 1);
47              value = PartialWrite(value, secondValue, 1, -1);
48
49              Assert.True(PartialRead(value, 0, 1) == firstValue);
50              Assert.True(PartialRead(value, 1, -1) == secondValue);
51          }
52
53          private static uint PartialWrite(uint target, uint source, int shift, int limit)
54          {
55              if (shift < 0)
56              {
57                  shift = 32 + shift;
58              }
59              if (limit < 0)
60              {
61                  limit = 32 + limit;
62              }
63              var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
64              var targetMask = ~(sourceMask << shift);
65              return (target & targetMask) | ((source & sourceMask) << shift);
66          }
67
68          private static uint PartialRead(uint target, int shift, int limit)
69          {
70              if (shift < 0)
71              {
72                  shift = 32 + shift;
73              }
74              if (limit < 0)
75              {
76                  limit = 32 + limit;
77              }
78              var sourceMask = ~(uint.MaxValue << limit) & uint.MaxValue;
79              var targetMask = sourceMask << shift;
80              return (target & targetMask) >> shift;
81          }
82      }
83  }
```

Index