./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 And<T>(T x, T y) => Arithmetic<T>.And(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);
    }
}
```

./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);
    }
}
```

./Arithmetic[T].cs

```
using System;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Reflection.Sigil;

// 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> And;
        public static readonly Func<T, T> Increment;
        public static readonly Func<T, T, T> Subtract;
        public static readonly Func<T, T> Decrement;

        static Arithmetic()
        {
            Add = DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Add();
                emiter.Return();
            });
            And = DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.And();
                emiter.Return();
            });
            Increment = DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                emiter.Increment(typeof(T));
                emiter.Return();
            });
            Subtract = DelegateHelpers.Compile<Func<T, T, T>>(emiter =>
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArguments(0, 1);
                emiter.Subtract();
                emiter.Return();
```

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

./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 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);
    }
}
```

./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);
    }
}
```

./Bit[T].cs

```
using System;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Reflection.Sigil;

// 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, int, int, T> PartialWrite;
        public static readonly Func<T, int, int, T> PartialRead;

        static Bit()
        {
            PartialWrite = 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);
                using (var sourceMask = emiter.DeclareLocal<T>())
                using (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();
            });
            PartialRead = 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);
                    using (var sourceMask = emiter.DeclareLocal<T>())
                    using (var targetMask = emiter.DeclareLocal<T>())
                    {
                        emiter.LoadConstant(typeof(T), numberFilledWithOnes);
                        emiter.LoadArgument(limitArgument); // limit
                        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();
                });
            }

            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();
            }
        }
    }
```

./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();
    }
}
```

./Integer[T].cs

```
using System;
using System.Linq;
using System.Reflection;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Reflection.Sigil;
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 = 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.ToArray()));
                }
                if (CachedTypeInfo<T>.IsNullable)
                {
                    emiter.NewObject(typeof(T), CachedTypeInfo<T>.UnderlyingType);
                }
                if (typeof(T) == typeof(Integer))
                {
                    emiter.NewObject(typeof(Integer), typeof(ulong));
                }
                emiter.NewObject(typeof(Integer<T>), typeof(T));
                emiter.Return();
            });
            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();
    }
}
```

./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);
    }
}
```

./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);
    }
}
```

./Math[T].cs

```
using System;
using System.Reflection;
using Platform.Exceptions;
using Platform.Reflection;
using Platform.Reflection.Sigil;

// 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 = DelegateHelpers.Compile<Func<T, T>>(emiter =>
```

```
            {
                Ensure.Always.IsNumeric<T>();
                emiter.LoadArgument(0);
                if (CachedTypeInfo<T>.IsSigned)
                {
                    emiter.Call(typeof(System.Math).GetTypeInfo().GetMethod("Abs", new[] {
                    ↪  typeof(T) }));
                }
                emiter.Return();
            });
            Negate = DelegateHelpers.Compile<Func<T, T>>(emiter =>
            {
                Ensure.Always.IsSigned<T>();
                emiter.LoadArgument(0);
                emiter.Negate();
                emiter.Return();
            });
        }
    }
}
```

Index