Data.Bits

Copyright	(c) The University of Glasgow 2001
License	BSD-style (see the file libraries/base/LICENSE)
Maintainer	libraries@haskell.org
Stability	experimental
Portability	portable
Safe Haskell	Trustworthy
Language	Haskell2010

This module defines bitwise operations for signed and unsigned integers. Instances of the class `Bits` for the `Int` and `Integer` types are available from this module, and instances for explicitly sized integral types are available from the Data.Int and Data.Word modules.

Documentation

`class Eq a => Bits a where` # Source

The `Bits` class defines bitwise operations over integral types.

- Bits are numbered from 0 with bit 0 being the least significant bit.

Minimal complete definition

`(.&.), (.|.), xor, complement, (shift | shiftL, shiftR), (rotate | rotateL, rotateR), bitSize, bitSizeMaybe, isSigned, testBit, bit, popCount`

Methods

`(.&.) :: a -> a -> a` | infixl 7 | # Source

Bitwise "and"

`(.|.) :: a -> a -> a` | infixl 5 | # Source

Bitwise "or"

`xor :: a -> a -> a` | infixl 6 | # Source

Bitwise "xor"

`complement :: a -> a` # Source

Reverse all the bits in the argument

`shift :: a -> Int -> a` | infixl 8 | # Source

`shift x i` shifts x left by `i` bits if `i` is positive, or right by `-i` bits otherwise. Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the `x` is negative and with 0 otherwise.

An instance can define either this unified `shift` or `shiftL` and `shiftR`, depending on which is more convenient for the type in question.

`rotate :: a -> Int -> a` | infixl 8 | # Source

`rotate x i` rotates x left by `i` bits if `i` is positive, or right by `-i` bits otherwise.

For unbounded types like `Integer`, `rotate` is equivalent to `shift`.

An instance can define either this unified `rotate` or `rotateL` and `rotateR`, depending on which is more convenient for the type in question.

`zeroBits :: a` # Source

`zeroBits` is the value with all bits unset.

The following laws ought to hold (for all valid bit indices *n*):

```
clearBit zeroBits n == zeroBits

setBit   zeroBits n == bit n

testBit  zeroBits n == False

popCount zeroBits   == 0
```

This method uses `clearBit (bit 0) 0` as its default implementation (which ought to be equivalent to `zeroBits` for types which possess a 0th bit).

Since: 4.7.0.0

bit :: Int -> a # Source

`bit i` is a value with the `i`th bit set and all other bits clear.

Can be implemented using `bitDefault` if a is also an instance of `Num`.

See also `zeroBits`.

setBit :: a -> Int -> a # Source

``x `setBit` i`` is the same as `x .|. bit i`

clearBit :: a -> Int -> a # Source

``x `clearBit` i`` is the same as `x .&. complement (bit i)`

complementBit :: a -> Int -> a # Source

``x `complementBit` i`` is the same as ``x `xor` bit i``

testBit :: a -> Int -> Bool # Source

Return `True` if the nth bit of the argument is 1

Can be implemented using `testBitDefault` if a is also an instance of `Num`.

bitSizeMaybe :: a -> Maybe Int # Source

Return the number of bits in the type of the argument. The actual value of the argument is ignored. Returns Nothing for types that do not have a fixed bitsize, like `Integer`.

Since: 4.7.0.0

bitSize :: a -> Int # Source

Deprecated: Use `bitSizeMaybe` or `finiteBitSize` instead

Return the number of bits in the type of the argument. The actual value of the argument is ignored. The function `bitSize` is undefined for types that do not have a fixed bitsize, like `Integer`.

isSigned :: a -> Bool # Source

Return `True` if the argument is a signed type. The actual value of the argument is ignored

shiftL :: a -> Int -> a | infixl 8 | # Source

Shift the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `shiftR` or the unified `shift`, depending on which is more convenient for the type in question.

`unsafeShiftL :: a -> Int -> a` # Source

Shift the argument left by the specified number of bits. The result is undefined for negative shift amounts and shift amounts greater or equal to the `bitSize`.

Defaults to `shiftL` unless defined explicitly by an instance.

Since: 4.5.0.0

`shiftR :: a -> Int -> a` | infixl 8 | # Source

Shift the first argument right by the specified number of bits. The result is undefined for negative shift amounts and shift amounts greater or equal to the `bitSize`.

Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the x is negative and with 0 otherwise.

An instance can define either this and `shiftL` or the unified `shift`, depending on which is more convenient for the type in question.

`unsafeShiftR :: a -> Int -> a` # Source

Shift the first argument right by the specified number of bits, which must be non-negative an smaller than the number of bits in the type.

Right shifts perform sign extension on signed number types; i.e. they fill the top bits with 1 if the x is negative and with 0 otherwise.

Defaults to `shiftR` unless defined explicitly by an instance.

Since: 4.5.0.0

`rotateL :: a -> Int -> a` | infixl 8 | # Source

Rotate the argument left by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateR` or the unified `rotate`, depending on which is more convenient for the type in question.

`rotateR :: a -> Int -> a` | infixl 8 | # Source

Rotate the argument right by the specified number of bits (which must be non-negative).

An instance can define either this and `rotateL` or the unified `rotate`, depending on which is more convenient for the type in question.

`popCount :: a -> Int` # Source

Return the number of set bits in the argument. This number is known as the population count or the Hamming weight.

Can be implemented using `popCountDefault` if a is also an instance of `Num`.

Since: 4.5.0.0

Instances

`Bits Bool` # Source

`Bits Int` # Source

`Bits Int8` # Source

`Bits Int16` # Source

`Bits Int32` # Source

`Bits Int64` # Source

`Bits Integer` # Source

Bits Word	# Source
Bits Word8	# Source
Bits Word16	# Source
Bits Word32	# Source
Bits Word64	# Source
Bits CUIntMax	# Source
Bits CIntMax	# Source
Bits CUIntPtr	# Source
Bits CIntPtr	# Source
Bits CSigAtomic	# Source
Bits CWchar	# Source
Bits CSize	# Source
Bits CPtrdiff	# Source
Bits CULLong	# Source
Bits CLLong	# Source
Bits CULong	# Source
Bits CLong	# Source
Bits CUInt	# Source
Bits CInt	# Source
Bits CUShort	# Source
Bits CShort	# Source
Bits CUChar	# Source
Bits CSChar	# Source
Bits CChar	# Source
Bits IntPtr	# Source
Bits WordPtr	# Source
Bits Fd	# Source
Bits CRLim	# Source
Bits CTcflag	# Source
Bits CUid	# Source
Bits CNlink	# Source
Bits CGid	# Source
Bits CSsize	# Source
Bits CPid	# Source
Bits COff	# Source
Bits CMode	# Source
Bits CIno	# Source
Bits CDev	# Source

`Bits Natural` # Source

`Bits a => Bits (Identity a)` # Source

`Bits a => Bits (Const k a b)` # Source

`class Bits b => FiniteBits b where` # Source

The `FiniteBits` class denotes types with a finite, fixed number of bits.

Since: 4.7.0.0

Minimal complete definition

`finiteBitSize`

Methods

`finiteBitSize` `:: b -> Int` # Source

Return the number of bits in the type of the argument. The actual value of the argument is ignored. Moreover, `finiteBitSize` is total, in contrast to the deprecated `bitSize` function it replaces.

```
finiteBitSize = bitSize
bitSizeMaybe = Just . finiteBitSize
```

Since: 4.7.0.0

`countLeadingZeros` `:: b -> Int` # Source

Count number of zero bits preceding the most significant set bit.

```
countLeadingZeros (zeroBits :: a) = finiteBitSize (zeroBits :: a)
```

`countLeadingZeros` can be used to compute log base 2 via

```
logBase2 x = finiteBitSize x - 1 - countLeadingZeros x
```

Note: The default implementation for this method is intentionally naive. However, the instances provided for the primitive integral types are implemented using CPU specific machine instructions.

Since: 4.8.0.0

`countTrailingZeros` `:: b -> Int` # Source

Count number of zero bits following the least significant set bit.

```
countTrailingZeros (zeroBits :: a) = finiteBitSize (zeroBits :: a)
countTrailingZeros . negate = countTrailingZeros
```

The related find-first-set operation can be expressed in terms of `countTrailingZeros` as follows

```
findFirstSet x = 1 + countTrailingZeros x
```

Note: The default implementation for this method is intentionally naive. However, the instances provided for the primitive integral types are implemented using CPU specific machine instructions.

Since: 4.8.0.0

Instances

`FiniteBits Bool` # Source

`FiniteBits Int` # Source

`FiniteBits Int8` # Source

FiniteBits Int16 # Source

FiniteBits Int32 # Source

FiniteBits Int64 # Source

FiniteBits Word # Source

FiniteBits Word8 # Source

FiniteBits Word16 # Source

FiniteBits Word32 # Source

FiniteBits Word64 # Source

FiniteBits CUIntMax # Source

FiniteBits CIntMax # Source

FiniteBits CUIntPtr # Source

FiniteBits CIntPtr # Source

FiniteBits CSigAtomic # Source

FiniteBits CWchar # Source

FiniteBits CSize # Source

FiniteBits CPtrdiff # Source

FiniteBits CULLong # Source

FiniteBits CLLong # Source

FiniteBits CULong # Source

FiniteBits CLong # Source

FiniteBits CUInt # Source

FiniteBits CInt # Source

FiniteBits CUShort # Source

FiniteBits CShort # Source

FiniteBits CUChar # Source

FiniteBits CSChar # Source

FiniteBits CChar # Source

FiniteBits IntPtr # Source

FiniteBits WordPtr # Source

FiniteBits Fd # Source

FiniteBits CRLim # Source

FiniteBits CTcflag # Source

FiniteBits CUid # Source

FiniteBits CNlink # Source

FiniteBits CGid # Source

FiniteBits CSsize # Source

FiniteBits CPid # Source

FiniteBits COff # Source

FiniteBits CMode # Source

`FiniteBits CIno` # Source

`FiniteBits CDev` # Source

`FiniteBits a => FiniteBits (Identity a)` # Source

`FiniteBits a => FiniteBits (Const k a b)` # Source

`bitDefault` `:: (Bits a, Num a) => Int -> a` # Source

Default implementation for `bit`.

Note that: `bitDefault i = 1 shiftL i`

Since: 4.6.0.0

`testBitDefault` `:: (Bits a, Num a) => a -> Int -> Bool` # Source

Default implementation for `testBit`.

Note that: `testBitDefault x i = (x .&. bit i) /= 0`

Since: 4.6.0.0

`popCountDefault` `:: (Bits a, Num a) => a -> Int` # Source

Default implementation for `popCount`.

This implementation is intentionally naive. Instances are expected to provide an optimized implementation for their size.

Since: 4.6.0.0

`toIntegralSized` `:: (Integral a, Integral b, Bits a, Bits b) => a -> Maybe b` # Source

Attempt to convert an `Integral` type a to an `Integral` type b using the size of the types as measured by `Bits` methods.

A simpler version of this function is:

```
toIntegral :: (Integral a, Integral b) => a -> Maybe b
toIntegral x
  | toInteger x == y = Just (fromInteger y)
  | otherwise        = Nothing
  where
    y = toInteger x
```

This version requires going through `Integer`, which can be inefficient. However, `toIntegralSized` is optimized to allow GHC to statically determine the relative type sizes (as measured by `bitSizeMaybe` and `isSigned`) and avoid going through `Integer` for many types. (The implementation uses `fromIntegral`, which is itself optimized with rules for `base` types but may go through `Integer` for some type pairs.)

Since: 4.8.0.0